

Non-intrusive constraint solver enhancements

Rémi Douence and Narendra Jussien
École des Mines de Nantes
La Chantrerie – 4, rue Alfred Kastler
BP 20722
F-44307 Nantes Cedex 3, France

Abstract

Constraint solvers rely on two simple mechanisms: enumeration and propagation. However, modern solvers integrate many optimizations and their actual implementations can be quite complex.

In this paper, we advocate for non-intrusive constraint solver enhancements. First, a minimal solver is implemented. Second, different enhancements (here explanation and dynamic backtracking [7] capabilities) of this minimal solver are implemented with the help of a new programming paradigm: aspect oriented programming. This new approach allow us to non-intrusively enhance the minimal solver: it remains unchanged.

1 Introduction

Constraint solvers are useful tools that can provide solutions to very complex problems. They rely mainly on two simple mechanisms: constraint propagation and enumeration. However, modern solvers (Ilog Solver, Chip from Cosytec, gnuProlog from INRIA, `choco` [17]) integrate many optimizations and their actual implementations can be quite complex.

In this paper, for the sake of software engineering and efficiency, we advocate for non-intrusive constraint solver enhancements. First, a minimal solver is implemented (described in section 4). Second, different enhancements (here explanation and dynamic backtracking capabilities, see section 5) of this minimal solver are implemented with the help of a new programming paradigm: aspect oriented programming [16]. This new approach allow us to non-intrusively enhance the minimal solver: it remains unchanged.

Before dwelling into details, let us first present our motivations in section 2 and the new programming paradigm known as aspect oriented programming (AOP from here on) in section 3.

2 Motivations

The motivation of this work comes from experimentating explanation-based constraint programming [10]. Although explanations have been proved very useful in lots of situations, they are often not offered in classical constraint programming systems. Adding them to an existing system can be a complex and tedious task. Before describing our solution to this problem, let us first make a quick survey on explanations for constraint programming.

2.1 Definition

A **contradiction explanation** (*a.k.a.* **nogood** [18]) is a subset of the current constraint 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.

$$C \vdash \neg (C' \wedge v_1 = a_1 \wedge \dots \wedge v_k = a_k) \quad (1)$$

In a contradiction explanation composed of at least one decision constraint, a variable v_j is selected and the previous formula is rewritten as¹:

$$C \vdash C' \wedge \bigwedge_{i \in [1..k] \setminus j} (v_i = a_i) \rightarrow v_j \neq a_j$$

The left hand side of the implication constitutes an **eliminating explanation** for the removal of value a_j from the domain of variable v_j and is noted $\text{expl}(v_j \neq a_j)$.

Classical CSP solvers use domain-reduction techniques (removal of values). Recording eliminating explanations is sufficient to compute contradiction explanations. Indeed, a contradiction is identified when the domain of a variable v_j is emptied. A contradiction explanation can easily be computed with the eliminating explanations associated with each removed value:

$$C \vdash \neg \left(\bigwedge_{a \in d(v_j)} \text{expl}(v_j \neq a) \right)$$

There exist generally several eliminating explanations for the removal of a given value. Recording all of them leads to an exponential space complexity. Another technique relies on *forgetting* (erasing) eliminating explanations that are no longer relevant² to the current variable assignment. By doing so, the space complexity remains polynomial.

¹A contradiction explanation that does not contain such a constraint denotes an over-constrained problem.

²A nogood is said to be relevant if all the decision constraints in it are still valid in the current search state [2].

2.2 Using explanations

Explanations can be used in several ways [13, 11, 14]. Debugging purposes pop to the mind: to explain **clearly** failures, to explain differences between intended and observed behavior for a given problem (*e.g.* why is value 4 not assigned to variable x ?).

Explanations can also be used to determine direct or indirect effects of a given constraint on the domains of the variables of the problem, and for dynamic constraint removal. This is the case with the justification system used in [3] for solving dynamic CSP. This justification system is actually a partial explanation system. Moreover, being able to explain failure and to dynamically remove a constraint facilitates the building of dynamic over-constrained problem solver [12].

Less direct applications are possible as well, in particular using explanations to guide the search. Indeed, classical backtracking-based searches only proceed when encountering failures (by backtracking to the last choice point). Contradiction explanation can be used to improve standard backtracking and to exploit information gathered to improve the search: to provide intelligent backtracking [8], to replace standard backtracking with a jump-based approach *à la Dynamic Backtracking* [7, 13], or even to develop new local searches on partial instantiations [14].

For example, Dynamic Backtracking can be implemented by exploiting explanations upon any failure. The contradiction explanation (the conflict) gives the value assignment to be undone and eliminating explanations help undoing that assignment with a minimal re-execution [13].

2.3 Computing explanations

Minimal (*w.r.t.* inclusion) explanations are the most *interesting* ones. They allow very precise information on emerging dependencies among variables and constraints, dependencies identified during the search. Unfortunately, computing such explanations can be quite time-consuming [9]. A good tradeoff between size and computability is the use of the knowledge that is *inside* the solver. Indeed, constraint solvers always know (although not often explicitly) why they remove values from the domains of variables³. Precise and interesting eliminating explanations can be computed by explicitly stating such information.

To achieve this behavior, the obvious action is to alter the code of the solver itself [11]. However, this intrusive behavior can become unbearable:

- modifications need to be made a numerous points in the solver code (namely, around each domain modification);
- complex propagation algorithms may be quite difficult to edit (no operational documentation, etc.)

³For example, in binary CSP, a value a is removed by propagation when all its supports on a given constraint c have been removed *i.e.* $\text{expl}(v_i \neq a) = \bigcup_{b \in \text{supp}(c, v_i, a)} \text{expl}(v_j \neq b)$

2.4 Related work

There are many works on computing conflicts *i.e.* contradiction explanations. On one hand, intrusive techniques include using (A)TMS systems [4] or adaptations [7] including the PaLM system [11]. Non-intrusive techniques have recently arisen: for example, QUICKXPLAIN [9] following works from [1] and [5] iteratively tests the consistency of subsets of constraints in order to compute a minimal contradiction explanation for a given inconsistent set of constraints.

However, computing inference explanation (*i.e.* eliminating explanations) has always, as far as we know, been done through intrusive algorithms. Consider for example, [19] which introduces explanations while solving logic puzzles. The idea is to use an inference-based solver that generates a *pre-compiled* explanation for each event.

We present here a new non-intrusive technique (based on AOP techniques – see the next Section) for computing contradiction and eliminating explanations thus filling a gap in the explanation computation algorithm menagerie. Such an approach is of great use in the following situations:

- when using black-box propagation engines that are not able to record local explanations like required by (A)TMS;
- for complex or large problems where methods such as (A)TMS lead to high overhead.

3 Aspect Oriented Programming

Separation of concerns is a well-known topic in software engineering. For instance, the Apache web server⁴ can be decomposed at the design level in a series of concerns: XML parsing, URL pattern matching, session logging, etc...

At the code level, it is often impossible, when a system is complex, to find an architecture that expresses the different concerns in a modular way. For instance, the Apache web server can be decomposed in a series of modules for XML parsing, URL pattern matching, etc... However, there is no module for session logging. Indeed, the corresponding code is distributed in many places in the other modules⁵. Session logging *crosscuts* the other concerns. Notice that explanation computing in constraints solvers also crosscuts the other concerns (e.g. propagation, enumeration, ...).

The current programming tools and languages do not support crosscutting concerns, and the programmer must tediously insert the session logging code by hand in other modules. Moreover, the lack of modularity of such a crosscutting concern makes its maintenance difficult. AOP [16] is a new programming paradigm supporting crosscutting concerns implementation. It allows the programmer to define the different concerns in a modular way; and it provides the

⁴apache.org

⁵O'Reilly Conference on Enterprise Java, March 29, 2001, pdf and ppt available at aspectj.org/doc/papersAndSlides.

programmer with a weaver responsible for mixing the different pieces of code together in order to generate the complete application.

Aspect-J⁶ [15] is an extension of Java⁷ that supports AOP. First, a base program is written in Java, and several aspects are defined in Aspect-J. Second, the aspect-weaver, basically a preprocessor, inserts calls to the aspectual code into the base program in order to generate the complete program.

In Aspect-J, an aspect is defined by a crosscut and an advice. A crosscut denotes a set of program points; it is defined with the help of method signatures. It designates where in the base program some code must be inserted. An advice is a piece of code to be inserted in the base program.

Aspect-J is not limited to code insertion at the beginning of methods. It can also make the variable of the base program accessible to advices, replace a method in the base program by another one, insert code when a field is accessed or when an exception is caught, introduce extra fields and methods in classes or modify the inheritance graph. For a more detailed description of Aspect-J, please refer to examples in section 5 and see [15].

Basically, Aspect-J is a preprocessor. However, its semantics is easier to understand from a monitoring perspective [6]. Let us consider the base program execution emits events such as *the method `fact` is called with parameter set to 4* or *the method `fact` returns with 24 as result*. In this context, an aspect can be seen as a monitor. For instance, a memoization aspect could monitor execution events corresponding to the method call signature as `fact(int x)`, pause the base program execution, and compare `x` with the value of the previous calls. If the value of the parameter has not been seen previously, the program execution is resumed and the corresponding return value will be stored. If the method has already been called with the same parameter value, the method body execution is skipped and the memoized result returned.

4 A minimal solver

In order to facilitate the presentation of our ideas, we use in the following a minimal solver written in Java: the **Cacao** solver⁸. We wanted to design a simple but not simplistic solver that nevertheless illustrates classical concerns in constraint programming.

Our solver only deals with binary constraints (relations) over CSP.

4.1 A basic behavior

Our minimal solver has a very basic but widely used behavior: a queue of relations (constraints) is used to propagate decision through the constraint network.

⁶aspectj.org

⁷java.sun.com

⁸All code described in this paper is available on request to the authors. We plan to make it available to the public audience on the web.

The main loop (see Figure 1) makes decisions *i.e.* extends the current partial assignment. That process ends when a solution is obtained (no remaining unassigned variable) or if the lack of solution has been proved. If a contradiction occurs, a Java exception (actually an instance of the class `Exception`) is thrown and caught within the loop where a function `error` is called. Its intended meaning is to perform a backtrack in order to move from the current dead-end. However, in our implementation, that function only states that a contradiction has occurred and halts the solver.

```

void run() {
    // finished is true if a solution has been found
    // or no solution can be found
    boolean finished = false;
    boolean feasible = true;
    try {
        // initial propagation
        relationQueue = originalRelations.copy();
        propagate();
        while (! finished) {
            try {
                // make some new decisions
                extend();
                propagate();
            }
            catch (Exception e) {
                // handling contradiction
                error();
            }
        }
        // a solution was found
        feasible = true;
        System.out.println("A solution\n" + this);
    }
    catch (Exception e) { // error itself threw an exception
        // there is no possible solution
        finished = true;
        feasible = false;
        System.out.println("No solution");
    }
}

```

Figure 1: Main loop of Cacao

Propagation in `Cacao` enforces arc-consistency by removing unsupported values in variable domains.

4.2 A basic architecture

- Variables are described in the class `Variable`. A variable has on `originalDomain` which contains values. Each value is unique (an instance of the class `Value`): *e.g.* the value 1 for variable x is different from value 1 for variable y : they are two different instances of the class `Value`.
- Constraints are described in the class `Relation` and are defined by repeatedly calling the method `add` to add an acceptable pair of values for the

variables of the constraint. Constraints have a proprietary method used to enforce arc-consistency on the related variables: `revise`.

- Value removals are performed by creating an instance of the class `Removal`.
- A general variable instance of the class `Problem` represents the current state of the problem. It contains the constraints and the general propagation set (`relationQueue`) and is able to propagate a decision through the constraint network by calling a method `propagate`.

Notice that we only use `Sets` in order to represent queues, stacks and lists. This prevents us from making too early choices in the design of the solver: strategies (variable choice, value choice, constraint selection, etc.) or representation (queues *vs.* stacks for constraint propagation, etc.)

5 Aspects for explanation-based constraint programming

We now define a few aspects in order to introduce explanations and dynamic backtracking in the minimal solver presented in the previous section. These enhancements are non-invasive since the minimal solver is never modified. We focus first on explanations.

5.1 Computing explanations

We decompose the explanation generation in two steps (*i.e.* aspects): *build list of supports for values* and *compute explanations*.

An aspect for storing information First, as noted in Section 2.3 the list of supports of a value is very useful in order to gather explanations of this value removal. Unfortunately our minimal solver does not maintain such a list. So, we define a new class `Support` (basically a pair relation-value) and an aspect `AspectBuildSupports`. In Figure 2, this aspect introduces a new field `supports` in the class `Value`. This extra field is initialized with an empty set of supports for each value. In the minimal solver, the method `add` of the class `Relation` extends a relation with two values; the second value supports the first one. So, the crosscut named `inRelation` captures these method calls and the corresponding advice (at the bottom of the figure) updates the set of supports of the first value every time the method `add` is called.⁹

At this point, we did not change a line of the minimal solver, however, with the help of `Aspect-J` and `AspectBuildSupports`, once the problem is build, each `Value` instance contains a list of its supports.

⁹Note that, in `Aspect-J`, an advice can denotes variables of the base program (for instance in Figure 2, `relation` denotes the receiver of the method call `add`). These variables can be used in the advice.

```

class Support {
    Relation relation;
    Value value;

    Support(Relation relation, Value value) {
        this.relation = relation;
        this.value = value;
    }
}

aspect AspectBuildSupports {
    Set Value.supports = new Set(); // introductions

    // build direct access to in relation values
    pointcut inRelation(Relation relation, Value value1, Value value2):
        target(relation) &&
        args(value1, value2) &&
        call(Relation Relation.add(Value, Value));

    before(Relation relation, Value value1, Value value2): inRelation(relation, value1, value2) {
        value1.supports.add(new Support(relation, value2));
    }
}

```

Figure 2: An aspect for supports building

An aspect for computing information The second step towards explanation generation requires to introduce an extra field `explanation` in every value as specified at the beginning of the aspect `AspectExplanation` in Figure 3. We also introduce an extra method `isDecision` in the class `Relation` in order to identify decision constraints (they are binary constraints that deal with the same variable twice). Finally, the aspect defines a crosscut named `removal` in order to detect the relation and the value involved every time a value is suppressed. In the minimal solver, when a value is removed (*i.e.* a `Removal` instance is created) the relation being revised is not known: all we know is there is a call to `Relation.revise` in progress in the control stack. So, the crosscut definition uses the Aspect-J construction `cflow` (for control flow) which allows us to remember the last revised relation (*i.e.* pending call to `revise`) when a value is removed. When a value is removed, the advice add the last revised relation (if it is a decision) to the explanation of this value removal. Then, it enumerates¹⁰ the support of this value, and it gathers the explanation of these supports removal.

At this point, we still did not change a line of the minimal solver, however every time a value is removed, its field `explanation` contains contains the set of relations that explain this suppression.

5.2 Using explanations

We now focus on backtracking in order to generate a solution with the help of aspects.

¹⁰In this paper, for the sake of conciseness, we use a pseudo javacode and note enumeration loops as `forall`.

```

aspect AspectExplanation {
  // introductions
  Set Value.explanation = new Set();
  boolean Relation.isDecision() {
    return var1.equals(var2);
  }
  pointcut removal(Relation r, Variable variable, Value value):
    cflow(target(r) && call(Removal revise()))
    && args(variable, value)
    && call(Removal.new(Variable, Value));

  before(Relation relation, Variable variable, Value value):
    removal(relation, variable, value) {
      // a set of decision constraints is now attached to its value
      if (relation.isDecision())
        value.explanation.add(relation);
      // and the explanations too
      Enumeration enum = value.supports.elements();
      forall support in value.supports
        if (support.relation == relation)
          value.explanation.union(support.value.explanation);
    }
  }
}

```

Figure 3: An aspect for explanation generation

An aspect for redefining parts of the solver In the minimal solver, when a domain becomes empty, an exception is thrown and the method `error` is called in order to print an error message and stop. In order to implement backtracking, our aspect must replace the original method `error` by another one that actually undoes decisions (with the help of explanations), repairs the state of the solver and resumes its execution. In Figure 4, the aspect `AspectBacktrack` defines a crosscut `callError` to denote the method call to `repair` and the associated advice replaces (*i.e.* keyword `around`) this method call by another one to `repair`.

```

aspect AspectBacktrack {
  pointcut callError():
    call(void Problem.error());

  void around() throws Exception: callError() {
    repair();
  }
  void repair() throws Exception {
    ...
  }
}

```

Figure 4: An aspect for backtracking

This long and complex method is detailed in Figure 5. It implements dynamic backtracking with the help of explanations as detailed in [13] (see also Section 2.2). This method could not be defined without aspects. Indeed, it has to access and modify the state of the solver but some pieces of information are not available in the minimal solver. We list here these different pieces of infor-

mation used in the `repair` method and the corresponding aspects that make them available.

```

void repair() throws Exception {
    Set contradictionExplanation = new Set();
    forall value in problem.lastModifiedVariable.originalDomain
        contradictionExplanation.union(value.explanation);
    if (contradictionExplanation.isEmpty()) { // no solution
        throw new Exception();
    } else {
        select most recent decisionToUndo from contradictionExplanation;
        // remove this constraint from the problem
        problem.originalRelations.remove(decisionToUndo);
        // remove past effects
        forall variable in problem.variables
            forall value in variable.originalDomain
                if (value.explanation.member(decisionToUndo)) {
                    // restore the value back in the domain
                    variable.domain.add(value);
                    // empty the explanation
                    value.explanation = new Set();
                    // prepare re-propagation
                    problem.relationQueue.union(problem.relations(variable));
                }
            }
        }
    try {
        problem.propagate();
    } catch (Exception e) {
        repair();
    }
    contradictionExplanation.remove(decisionToUndo);
    boolean decisionsStillValid = all decisions in contradictionExplanation valid;
    if (decisionsStillValid) {
        try {
            // remove a value (equivalent to a non-decision relation)
            Value value = (Value)((Pair)(decisionToUndo.pairs.get()).fst);
            decisionToUndo.var1.domain.remove(value);
            // set explanation
            value.explanation = contradictionExplanation;
            // prepare propagation
            problem.relationQueue.union(problem.relations(problem.lastModifiedVariable));
            problem.propagate();
        } catch (Exception e) {
            repair();
        }
    }
}

```

Figure 5: A method for repairing/backtracking

Aspects for accessing the state of the solver First, the method `repair` must be able to access the solver state. So, the aspect `AspectBacktrack` in Figure 4 is *extended* with a variable `problem` initialized with the reference of the instance of the problem to be solved¹¹. This way, the `repair` method can access for instance the `relationQueue` or call the method `propagate`. The complete aspect `AspectBacktrack` including the variable `problem` is defined in Figure 6.

¹¹Indeed, as stated in Section 4, the problem codes the state of the solver

Second, when a domain becomes empty, it is mandatory to know the last modified variable in order to study its contradiction explanation. In the minimal solver, the last modified variable is not known when `error` is called. So, the aspect `AspectLastModifiedVariable` is defined as in Figure 7 in order to keep track of this variable identity.

Third, the original domain of the last modified variable must be enumerated. This extra information is introduced in the minimal solver with the help of the aspect `AspectOriginalDomain` defined in Figure 8. This aspect stores all the values added in a variable (thus defining its domain) at creation time.

Finally, for the sake of completeness (see [7]), the method `repair` must select *the most recent* decision to undo. To this end, the aspect `AspectTimeStamp` defined in Figure 9 introduces a time stamp (actually a simple number rather than an actual time) in every relation at creation time.

5.3 Demonstration

The minimal solver and the different aspects described in the previous sections can be compiled with Aspect-J. We sketch here a scenario.

First, we introduce a few java lines in `CacaoSolver.java` at the beginning of the `main` method in order to build a problem to be solved (for a complete description of the problem the reader should request the code available to the authors¹²). Then, we compile the minimal solver with `javac` and run it with `java`. Unfortunately, the choice of values for variables leads to a dead end: the original method `error` prints `no solution`.

```
> javac CacaoSolver.java
OK
> java CacaoSolver
no solution
```

So, we compile the same minimal solver and the previously detailed aspects with the help of the aspect weaver `ajc`. Then, we run it with `java`¹³: this time the method `repair` is called instead of `error`. It undoes choices with the help of explanations and tries again different values. In this case, the solver finds a solution.

```
> ajc CacaoSolver.java AspectBuildSupports.java
AspectExplanation.java AspectBacktrack.java
AspectOriginalDomain.java AspectTimeStamp.java
AspectLastModifiedVariable.java
OK
> java CacaoSolver
one solution found
```

¹²A downloadable version will be made available online as soon as possible.

¹³remember `ajc`, the Aspect-J weaver is a preprocessor that relies on the standard `javac` compiler.

```

aspect AspectBacktrack {
    Problem problem;
    pointcut newProblem():
        call(Problem.new(..));
    after() returning(Problem problem): newProblem() {
        this.problem = problem;
    }
    ... // see Figure 4
}

```

Figure 6: An augmented aspect for backtracking

```

aspect AspectLastModifiedVariable {
    Variable Problem.lastModifiedVariable;

    pointcut removal(Relation r, Variable variable, Value value):
        cflow(target(r) && call(Removal revise()))
        && args(variable, value)
        && call(Removal.new(Variable, Value));

    before(Relation relation, Variable variable, Value value):
    removal(relation, variable, value) {
        // remember the last variable
        this.problem.lastModifiedVariable = variable;
    }
}

```

Figure 7: An aspect for keeping track of the last modified variable

```

aspect AspectOriginalDomain {
    Set Variable.originalDomain = new Set();

    pointcut newVariable(): call(Variable.new(String, Set));

    after() returning(Variable variable): newVariable() {
        variable.originalDomain = variable.domain.copy();
    }
}

```

Figure 8: An aspect for keeping track of each variable original domain

```

aspect AspectTimeStampRelation {
    static int timeStampGenerator = 0;
    int Relation.timeStamp;

    pointcut newRelation(): call(Relation.new(..));

    after() returning(Relation relation): newRelation() {
        relation.timeStamp = timeStampGenerator++;
    }
}

```

Figure 9: An aspect for dating the (decision) relations

5.4 Discussion

We have demonstrated how a minimal constraint solver could be non intrusively enhanced with the help of AOP. We have exemplified different uses of aspects. First, aspects can store information (such as the parameters of a method called at the beginning of the solver execution). Second, aspects can compute information on the fly as the execution progresses. Notice that these aspects do not modify the control flow of the minimal solver but enhance it by interleaving extra instructions. Third, we have redefined parts of the solver in order to modify its behavior (the minimal method `error` has been replaced by a backtracking one that modifies the state of the solver and resume its execution). Finally, aspects can make accessible the solver state at program points it was not accessible in order to implement `repair`.

The minimal solver architecture, described in Section 4, must be known by the aspect programmer. For instance, in order to build the lists of supports, he must know a relation is built by repeated calls to the method `add`. In the other hand, he does not need to know the detailed implementation of the minimal solver. For example, the propagation could use a stack instead of a queue (or the `Set` of values could be replaced by `List` of values in order to make wiser choice at enumeration time) and our aspects would still be valid.

6 Conclusion and Future work

In this paper, we have demonstrated how a minimal constraint solver could be non intrusively enhanced with the help of AOP in order to implement explanations and dynamic backtracking. Moreover, this automated modifications¹⁴ of code allow to ensure some correctness properties on the obtained code. Even more, as in partial evaluation, such an automated tool can make more optimization that a human can see.

Even though, the minimal solver architecture must be known by the aspect programmer, he does not need to know the implementation details of the minimal solver. This knowledge remains to be precisely characterized in order to formally define aspects validity for the sake of correctness and reuse.

Future works also include practical experiments with existing solvers. Indeed, we have developed our minimal solver with no assumption about its future enhancements. Then, we have defined aspects without modifying the minimal solver. We believe existing solvers could be enhanced the same way without modification. This should be practically studied. We should also study aspect based enhancements of more complex solvers that deal, for example, with global constraints.

Finally, we plan to study how aspects could be used to declaratively express strategies in order to guide the solver. For instance, *at enumeration time, select the variable but V1 with the smallest domain*, or *at propagation time, use*

¹⁴Notice that the obtained code is really close to what would have been obtained if the modifications were made by hand.

depth first propagation when the variable V2 is involved and use breadth first propagation when the relation R1 is involved could be expressed as aspects by accessing the state of the solver and redefining parts of it. We believe AOP is a very promising track in order to make constraint solvers programmable, hence more efficient.

References

- [1] R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In *Proceedings IJCAI'93*, pages 276–281, 1993.
- [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] Johan de Kleer. An assumption-based tms. *Artificial Intelligence*, 28:127–162, 1986.
- [5] N. de Siqueira and J. F. Puget. Explanation-based generalisation of failures. In *Proceedings ECAI'88*, pages 339–344, Munich, Germany, 1988.
- [6] Remi Douence, Olivier Motelet, and Mario Sudholt. A formal definition of crosscuts. In *Reflection*, pages 170–186, 2001.
- [7] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
- [8] 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.
- [9] 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.
- [10] Narendra Jussien. e-constraints: explanation-based constraint programming. In *CP01 Workshop on User-Interaction in Constraint Satisfaction*, Paphos, Cyprus, 1 December 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. 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.
- [15] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP*, pages 327–353, 2001.
- [16] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP ’97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 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] Thomas Schiex and Gérard Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.
- [19] Mohammed H. Sqalli and Eugene C. Freuder. Inference-based constraint satisfaction supports explanation. In *AAAI: National Conference on Artificial Intelligence*, pages 318–325, 1996.