

# Non-intrusive constraint solver enhancements\*

Rémi Douence and Narendra Jussien

École des Mines de Nantes – 4, rue Alfred Kastler – BP 20722  
F-44307 Nantes Cedex 3 – France  
{douence, jussien}@emn.fr  
WWW: <http://www.emn.fr/{douence,jussien}>

**Abstract.** Using conflict sets (or nogoods) and explanations within constraint programming has been proved very effective. However, most constraint solvers do not provide this feature. This statement could have been made for many other improvements. Indeed, one of the main reasons of that fact is that many improvements in constraint programming are **intrusive**: their integration requires a general modification of the solvers' implementation and/or architecture. The core part of constraint solvers is often quite simple, however, it represents only a small part of the implementation. The main part of the code is devoted to specific constraint handling, global constraints, search techniques, API, etc. Modifying this code requires a real development effort that may become overly costly. Constraint solvers need non intrusive approaches. Actually, solvers should not be modified at all and only a general information about implementation should be needed to integrate improvements. In this paper, we present a technique used in software engineering to reach that aim: aspect oriented programming. As an example, the non intrusive integration of conflict set generation and use is presented and some insights of what could be done are provided.

## 1 Introduction

Constraint programming improvements take a long time to make it through commercial constraint solvers. For example, using conflict sets (or nogoods) and explanations within constraint programming has been proved very effective [1–5] but no modern constraint solver (Ilog Solver [6], Chip [7], gnuProlog [8], `choco` [9]) do not provide this feature.

One of the main reasons of that fact is that many improvements in constraint programming are **intrusive**: they require rewriting large parts of the solvers in order to be handled [10]. Indeed, modifications need to be done to variable and domain representations (to store that new information), to constraint propagation (to compute new information to be recorded), to search mechanisms (replacing backtracking by other techniques), etc.

Constraint solvers are heavy and complex pieces of software. The core part of constraint solvers is often quite simple and relatively small. However, specific constraint handling, global constraints, search techniques, API add many lines of codes that may need modification when integrating improvements. Modifying a constraint solver requires a real development effort that may become overly costly.

Actually, solvers should not be modified at all on only a general information about the actual implementation should be sufficient to integrate improvements. In this paper, we present a technique used in software engineering to reach that aim: aspect oriented programming [11].

The aim of our paper is two-fold: introducing *aspect oriented programming* through an example: adding conflict set generation and use within an event-based constraint solver; paving the way to more intricate uses of aspect oriented programming within constraint programming: implementing complex strategies within constraint solvers, enhancing solvers while keeping simple and easily understandable implementations, etc.

This paper is organized as follows. First, aspect oriented programming (AOP from here on) is presented (section 2). Second, conflicts and explanations concepts are recalled in section 3. Third a conflict-based search algorithm is non-intrusively implemented using AOP (section 5) within

---

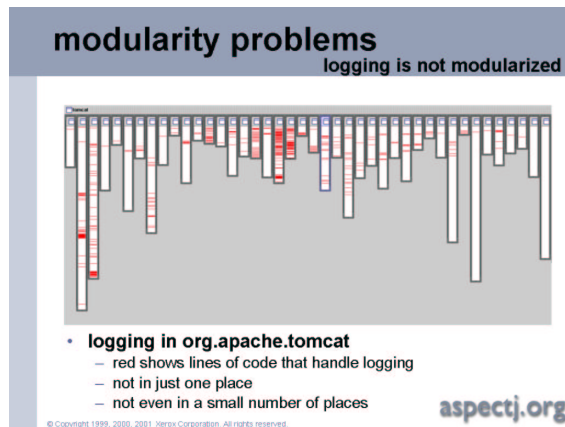
\* Work partially funded by the EU project *EasyComp* ([www.easycomp.org](http://www.easycomp.org)), No. IST-1999-014191.

a minimal solver (described in section 4) which therefore remains unchanged. Finally, we open a discussion about the use of AOP within constraint programming sketching how this approach could be used to make programmable solvers (*eg.* guiding them through declarative strategies) in section 6.

## 2 Aspect Oriented Programming

Separation of concerns is a well-known topic in software engineering. For instance, the Apache web server<sup>1</sup> 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 example, in the Apache web server there is no module for session logging. Indeed, the corresponding code is distributed in many places in the other modules as shown on the slide<sup>2</sup> in Figure 1. Each vertical white bar represents a module, and each horizontal grey line in a white bar represents a line of code implementing session logging.



**Fig. 1.** In Apache, session logging *crosscuts* the other concerns.

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 [11] is a new programming paradigm supporting crosscutting concerns. It allows the programmer to define the different concerns in a modular way; and it provides the programmer with a weaver responsible for mixing the different pieces of code together to generate the complete application.

Aspect-J<sup>3</sup> [12] is an extension of Java 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 pre-processor<sup>4</sup>, inserts calls to the aspectual code into the base program to generate the complete program.

In Aspect-J, an aspect is defined by gathering crosscuts and advices. 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. For instance, `pointcut profile(): call(Removal`

<sup>1</sup> [apache.org](http://apache.org)

<sup>2</sup> O'Reilly Conference on Enterprise Java, March 29, 2001, pdf and ppt available at [aspectj.org/doc/papersAndSlides](http://aspectj.org/doc/papersAndSlides).

<sup>3</sup> [aspectj.org](http://aspectj.org)

<sup>4</sup> However, we believe its semantics is easier to understand from a monitoring perspective [13].

`Relation.revise()`) denotes program points where the method `revise` is called. An advice is a piece of code to be inserted in the base program. For instance, `before(): profile() { counter++; }` increments the local variable `counter` of the aspect every time `revise` is called. So, a complete profiling aspect can be defined as in Figure 2.

```

aspect AspectProfiling {
  int counter = 0;
  boolean isOn = true;

  pointcut profile(): call(Replacement Relation.revise());

  before(): profile() { if (isOn) counter++; }
}

```

**Fig. 2.** A tiny profiling aspect

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.

Obviously, in order to define more interesting profiling aspects, we could capture variables of the profiled program (*e.g.* the relation being revised), dynamically change the value of `isOn` by defining two extra crosscuts and their corresponding advices (*e.g.* `isOn` is assigned with `true` – resp. `false` – every time the method `propagate` – resp. `extend` – is called), or use wildcards in signatures (*e.g.* in order to monitor every method of the class `Relation`). Section 5 exemplifies more of these constructions (see [12] for a complete description of Aspect-J).

### 3 Conflict sets and constraint programming

A *Constraint Satisfaction Problem* (CSP) is defined by a set of variables  $V = \{v_1, v_2, \dots, v_n\}$ , their respective value domains  $D_1, D_2, \dots, D_n$  and a set of constraints  $C = \{c_1, c_2, \dots, c_m\}$ . A solution of the CSP is an assignment of a single value to each of the variables such that all constraints in  $C$  are satisfied. We denote by  $\mathbf{sol}(V, C)$  the set of solutions of the CSP  $(V, C)$ . Variable domains are considered as unary constraints. Moreover, the classical enumeration mechanism that is used to explore the search space is modelled as a series of constraints additions (value assignments) and retractions (backtracks). Those particular constraints are called *decision constraints*.

Let us consider a constraint system whose current state (*i.e.* the original constraint and the set of decisions made so far) is contradictory. A **conflict set** (*a.k.a.* **nogood** [2]) is a subset of the current constraint system that, on its own, leads to a contradiction (no feasible solution contains a nogood). A conflict can be divided in two parts: a subset of the original set of constraints ( $C' \subset C$  in the following equation) and a subset of decision constraints introduced so far in the search (here  $dc_1, \dots, dc_k$ ). We have:  $\mathbf{sol}(V, (C' \wedge dc_1 \wedge \dots \wedge dc_k)) = \emptyset$ .

An operational viewpoint of conflict sets can be made explicit by rewriting the previous equation the following way:  $C' \wedge \left( \bigwedge_{i \in [1..k] \setminus j} dc_i \right) \rightarrow \neg dc_j$ .

Let us assume  $dc_j : v = a$  in the previous formula. This leads to the following result ( $s(v)$  is the value of variable  $v$  in the solution  $s$ ):  $\forall s \in \mathbf{sol}\left(V, C' \wedge \left( \bigwedge_{i \in [1..k] \setminus j} dc_i \right)\right), s(v) \neq a$ .

The left hand side of the implication in the previous equation 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 is noted:  $\mathbf{expl}(v \neq a)$ .

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

For example, a conflict can be computed from the empty domain of a variable  $v$  (using explanations for each of the values):  $\bigwedge_{a \in d(v)} \text{expl}(v \neq a)$ .

### 3.1 Using conflicts and explanations

Explanations can be used in several ways [14, 10, 4, 5]. Debugging purposes come to mind: to explain **clearly** failures, to explain differences between intended and observed behavior for a given problem (*eg.* 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 [15]. Moreover, being able to explain failure and to dynamically remove a constraint facilitates the building of dynamic over-constrained problem solvers.

Explanations and conflicts can be used to efficiently guide search. Indeed, classical backtracking-based searches only proceed when encountering failures (by backtracking to the last choice point). Conflicts can be used to improve standard backtracking and to exploit information gathered to improve the search: to provide intelligent backtracking [16], to replace standard backtracking with a jump-based approach *à la Dynamic Backtracking* [1, 4], or even to develop new local searches on partial instantiations [17].

### 3.2 Computing explanations

As we saw, explanations can be used to derive conflicts. Therefore, we will focus in the following in computing explanations. The most interesting explanations (and conflicts) 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 [18].

Several explanations generally exist for the removal of a given value. Recording all of them is not affordable (it leads to an exponential space complexity). A good compromise between precision and ease of computation is to use the solver-embedded knowledge to provide interesting explanations. Indeed, constraint solvers always know, although it is scarcely explicit, *why* they remove values from the domain of the variables<sup>5</sup>. By making that knowledge explicit and therefore sort of *tracing* the behavior of the solver, quite precise and interesting explanations can be computed.

To achieve this behavior, the obvious action is to alter the code of the solver itself [10]. However, this intrusive behavior can become unbearable:

- modifications need to be made at numerous points in the solver code (namely, around each domain modification *i.e.* for each type of provided constraint);
- complex propagation algorithms may be quite difficult to edit (no operational documentation, etc.)

### 3.3 Related work

There is a lot of work on computing conflicts. On one hand, intrusive techniques include (A)TMS-like systems [19] or adaptations [1, 10]. On the other hand, non-intrusive techniques have recently arisen: for example, QUICKPLAIN [18] following work from [20] and [21] iteratively tests the consistency of subsets of the constraints to compute a minimal conflict for a given inconsistent set of constraints.

However, computing explanations has always, as far as we known, been done through intrusive algorithms. Consider, for example, [22] 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.

<sup>5</sup> For 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.*  $\text{expl}(v_i \neq a) = \bigcup_{b \in \text{supp}(c, v_i, a)} \text{expl}(v_j \neq b)$ .

We present in the following a new non-intrusive technique based on AOP techniques for computing conflicts and 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 cannot record local explanations like required by (A)TMS;
- for complex or large problems where methods such as (A)TMS lead to high overhead.

## 4 A minimal solver

To present our ideas, we use in the following a minimal solver written in Java: the `Cacao` solver<sup>6</sup>. We designed a simple but not simplistic solver that illustrates classical concerns in constraint programming.

### 4.1 A basic architecture

Our minimal solver implements the following classical concepts using an object-based representation.

- Variables are described in the class `Variable`. A variable has a slot `originalDomain` which contains the original set of values that represent the domain of the variable. Each value is unique<sup>7</sup> (an instance of the class `Value`): *eg.* 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 repetitively 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`. Our minimal solver only deals with binary constraints (relations) over CSP<sup>8</sup>.
- Variables and constraints are part of a CSP (represented by the `Problem` class). It stores the representation of the state of the problem during search (for example, there exists a general propagation set (`relationsToPropagate`)). Moreover, a `Problem` provides API methods such as: `propagate` which propagates the current constraint system through the constraint network and a general `solve` method.

As we use arc-consistency techniques, constraint propagation is performed through domain reduction. To provide a clear and clean model for our solver, value removals are explicitly stated through a dedicated class<sup>9</sup>: `Removal`.

Notice that our minimal solver only uses `Sets` to represent queues, stacks and lists. This prevents us for making too early choices in the design of the solver: representations and strategies (variable choice, value choice, constraint selection, queues *vs.* stacks for constraint propagation, etc.). We discuss this topic at the end of this paper.

<sup>6</sup> All code describe in this paper is available on request to the authors. We plan to make it available to the public audience on the web.

<sup>7</sup> People may argue that this representation may be too costly (object creation, complicated comparisons, etc.). However, we do not focus here on efficiency but rather on concept illustration.

<sup>8</sup> Notice that classical solvers deal with much more kind of constraints. What we will present in the following needs to be generalized to those constraints.

<sup>9</sup> Notice that classical solvers do not provide such a dedicated class: removals are performed immediately. However, we chose to implement such a class to ease the presentation of our ideas. As we will see in the remainder of this paper, AOP could have been easily used to implement such a behavior from outside the core solver.

## 4.2 A basic behavior

Our minimal solver is not complete. Actually, it cannot even perform a mere backtrack. The idea behind this choice is to focus on the real part of the work performed by a constraint solver: constraint propagation. Backtracking can be considered as a simple enhancement of a basic solver.

Therefore our solver is only capable to make decisions (adding decision constraints) and propagate them through the constraint network. It has a very basic but widely used behavior: a set of relations (constraints) is used for propagation.

The main loop (see figure 3) 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 (an instance of the class `Exception`) is thrown and caught within the loop where a function `error` is called. In classical solver, `error` would perform chronological backtracking to escape from the current dead-end. However, in our implementation, that method 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
        relationsToPropagate = 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");
}
}

```

Fig. 3. Main loop of Cacao

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

## 5 Aspects for conflict-based constraint programming

We now define a few aspects to introduce explanations, conflicts and dynamic backtracking in the minimal solver presented in the previous section. These enhancements are non-intrusive because the minimal solver is never modified. We first focus 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 3.2 the list of supports of a value is very useful in 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 4, 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.<sup>10</sup>

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.

```

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 rel, Value val1, Value val2):
        target(rel) &&
        args(val1, val2) &&
        call(Relation Relation.add(Value, Value));

    before(Relation rel, Value val1, Value val2)
    : inRelation(rel, val1, val2) {
        val1.supports.add(new Support(rel, val2));
    }
}

```

Fig. 4. 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 5. We also introduce an extra method `isDecision` in the class `Relation` to identify decision constraints (they are binary constraints that deal with the same variable twice). Finally, the aspect defines a crosscut named `removal` 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 that 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 adds the last revised relation (if it is a decision) to the explanation of this value removal. Then, it enumerates<sup>11</sup> 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 the set of relations that explain this suppression.

## 5.2 Using explanations

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

*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 to print an error message and stop. 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 6, the aspect `AspectDynamicBacktrack` defines a crosscut `callError` to denote the method call to `error` and the associated advice replaces (*i.e.* keyword `around`) this method call by another one to `repair`.

<sup>10</sup> Note that, in Aspect-J, an advice can denote variables of the base program (for instance in Figure 4, `relation` denotes the `target` (*i.e.* receiver) of the method call `add`). These variables can be used in the advice.

<sup>11</sup> In this paper, for the sake of conciseness, we use pseudo Java Code and write 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);
    }
}

```

Fig. 5. An aspect for explanation generation

<pre> aspect AspectDynamicBacktrack {   pointcut callError():     call(void Problem.error());    void around() throws Exception: callError() {     repair();   }   void repair() throws Exception {     ...   } } </pre>		<pre> aspect AspectDynamicBacktrack { // augmented   Problem problem;   pointcut newProblem():     call(Problem.new(..));   after() returning(Problem problem): newProblem() {     this.problem = problem;   }   ... // see original version } </pre>
--	--	---

Fig. 6. An aspect for dynamic backtracking (*left*) and its augmented version (*right*)

This long and complex method is detailed in Figure 7. It implements dynamic backtracking with the help of explanations as detailed in [4] (see also Section 3.1). 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 information 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.relationsToPropagate.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.relationsToPropagate.union(problem.relations(problem.lastModifiedVariable));
                problem.propagate();
            } catch (Exception e) {
                repair();
            }
        }
    }
}

```

**Fig. 7.** 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 `AspectDynamicBacktrack` in Figure 6 is *extended* with a variable `problem` initialized with the reference of the instance of the problem to be solved<sup>12</sup>. This way, the `repair` method can access the `relationsToPropagate` and call the method `propagate`. The complete aspect `AspectDynamicBacktrack` including the variable `problem` is defined in Figure 6.

Second, when a domain becomes empty, it is mandatory to know the last modified variable 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 8 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`

<sup>12</sup> Indeed, as stated in Section 4, the problem codes the state of the solver

```

aspect AspectLastModifiedVariable {
    Variable Problem.lastModifiedVariable;

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

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

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

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

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

```

**Fig. 8.** An aspect for keeping track of the last modified variable (*left*) and another one for keeping track of each variable original domain (*right*)

```

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

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

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

```

**Fig. 9.** An aspect for time-stamp generation

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 [1]), the method `repair` must select *the most recent* decision to undo. The aspect `AspectTimeStamp` defined in Figure 9 introduces a time stamp (actually a simple number rather than the actual time) in every relation at creation time.

## 6 Discussion

We have demonstrated how a minimal constraint solver could be non intrusively enhanced with the help of AOP. In this section, we introduce a taxonomy of aspects for constraint programming and then discuss how our work can be extended for commercial solvers.

### 6.1 AOP features

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. These aspects do not modify the control flow of the minimal solver but interleave extra instructions. Third, aspects have redefined parts of the solver to modify its behavior (the minimal method `error` has been replaced by a dynamic-backtracking one that modifies the state of the solver and resume its execution). Finally, aspects can make the solver state accessible at program points where it was not originally accessible (here to implement `repair`).

### 6.2 AOP and existing solvers

We chose to present our vision of AOP for constraint programming through a minimal solver. This is arguable. Indeed, our solver does not cover all the features of modern solvers. For example, we only deal here with binary constraints represented in extension. Classical solvers handle different kinds of constraints. When computing explanations, different kinds of constraints lead to different

crosscuts [10]. However, there exist generic operators in Aspect-J that can be used to design generic crosscuts in a very compact way (*e.g.* `call(* *.add(..)`) denotes *every* call sites of the method `add`).

Another arguable point is that our minimal solver and the enhancements we present do not provide backtracking or trailing mechanisms. Indeed, we chose to focus on dynamic backtracking because its implementation in constraint solvers is difficult [1, 10]. Backtracking/trailing is also a crosscutting concern that could be expressed as an aspect.

A current limitation of our approach relies in the fact that constraint solvers need to be deployed in AOP-aware platforms. However, notice that both `choco` and Ilog Solver are being ported to Java which allow to use Aspect-J. Moreover, work is being done leading to an AOP platform C-compatible [23] which will lead to being able to use AOP for most of constraint solvers.

In our opinion, the challenge is to design an Aspect-Constraint platform which would be dedicated to constraint programming using its concepts (propagation, variable selection, etc.) as first-order constructs in the aspect definition language. With such a platform, the programmer will not have to know the solver architecture and will be able to focus only on constraint programming concepts.

## 7 Conclusion

In this paper, we have demonstrated how a minimal constraint solver could be non intrusively enhanced with the help of AOP to implement explanations and dynamic backtracking. With this technique, the tradeoff between maintainability and efficiency vanishes and the programmer feels free to express any optimization.

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

Future work 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 must be practically experimented. We must also study aspect-based enhancements of more complex solvers that deal, for example, with global constraints.

In this paper, we focussed on dynamic backtracking, obviously alternative versions of backtracking (*e.g.* standard backtracking or intelligent backtracking) could be introduced in the same way. However, non intrusive solver enhancements are not limited to backtracking algorithms. In particular, programmers should be allowed to guide solvers by defining strategies, and we believe AOP is a good candidate to express them. Even more, we feel that AOP is a very promising track in order to make constraint solvers programmable, hence more efficient.

## Acknowledgments

We would like to thank the anonymous referees who helped improve our paper and Philippe David for fruitful discussions.

## References

1. Ginsberg, M.: Dynamic backtracking. *Journal of Artificial Intelligence Research* **1** (1993) 25–46
2. Schiex, T., Verfaillie, G.: Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools* **3** (1994) 187–207
3. Jussien, N.: e-constraints: explanation-based constraint programming. In: CP01 Workshop on User-Interaction in Constraint Satisfaction, Paphos, Cyprus (2001)

4. Jussien, N., Debruyne, R., Boizumault, P.: Maintaining arc-consistency within dynamic backtracking. In: *Principles and Practice of Constraint Programming (CP 2000)*. Number 1894 in *Lecture Notes in Computer Science*, Singapore, Springer-Verlag (2000) 249–261
5. Jussien, N., Lhomme, O.: Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence (2002)* to appear.
6. Ilog: Solver reference manual (2001)
7. Aggoun, A., Dincbas, M., Herold, A., Simonis, H., Van Hentenryck, P.: The CHIP System. Technical Report TR-LP-24, ECRC, Munich, Germany (1987)
8. Diaz, D., Codognet, P.: The GNU prolog system and its implementation. In: *ACM Symposium on Applied Computing*, Villa Olmo, Como, Italy (2000)
9. Laburthe, F.: Choco: implementing a cp kernel. In: *CP00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems (TRICS)*, Singapore (2000)
10. Jussien, N., Barichard, V.: The palm system: explanation-based constraint programming. In: *Proceedings of TRICS: Techniques for Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, Singapore (2000) 118–133
11. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: *ECOOP '97 — Object-Oriented Programming 11th European Conference*, Jyväskylä, Finland. Volume 1241. Springer-Verlag, New York, NY (1997) 220–242
12. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In: *ECOOP*. (2001) 327–353
13. Douence, R., Motelet, O., Sudholt, M.: A formal definition of crosscuts. In: *Reflection*. (2001) 170–186
14. Albin-Amiot, H., Cointe, P., Guéhéneuc, Y.G., Jussien, N.: Instantiating and detecting design patterns: Putting bits and pieces together. In: *16th IEEE conference on Automated Software Engineering (ASE'01)*, San Diego, USA (2001)
15. Jussien, N., Boizumault, P.: Best-first search for property maintenance in reactive constraints systems. In: *International Logic Programming Symposium*, Port Jefferson, N.Y., USA, MIT Press (1997) 339–353
16. Guéret, C., Jussien, N., Prins, C.: Using intelligent backtracking to improve branch and bound methods: an application to open-shop problems. *European Journal of Operational Research* **127** (2000) 344–354
17. Jussien, N., Lhomme, O.: Local search with constraint propagation and conflict-based heuristics. In: *Seventh National Conference on Artificial Intelligence – AAAI'2000*, Austin, TX, USA (2000) 169–174
18. Junker, U.: QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. In: *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, Seattle, WA, USA (2001)
19. de Kleer, J.: An assumption-based tms. *Artificial Intelligence* **28** (1986) 127–162
20. Bakker, R., Dikker, F., Tempelman, F., Wognum, P.: Diagnosing and solving over-determined constraint satisfaction problems. In: *Proceedings IJCAI'93*. (1993) 276–281
21. de Siqueira, N., Puget, J.: Explanation-based generalisation of failures. In: *Proceedings ECAI'88*, Munich, Germany (1988) 339–344
22. Sqalli, M.H., Freuder, E.C.: Inference-based constraint satisfaction supports explanation. In: *AAAI: National Conference on Artificial Intelligence*. (1996) 318–325
23. Coady, Y., Kiczales, G., Feeley, M., Smolyn, G.: Using AspectC to improve the modularity of path-specific customization in operating system code. In: *Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*. (2001)