Scientific Annals in Computer Science Vol.17, 2007
"Alexandru Ioan Cuza" University of Iaşi, Romania

Basic Techniques for Creating an Efficient CSP Solver [1]

Cristian FRĂSINARU[2]

Abstract

Many computationally difficult problems from areas like planning and scheduling are easily modelled as *constraint satisfaction problems* (CSP). In order to have an uniform practical approach of these, a new programming paradigm emerged in the form of *constraint programming*, providing the opportunity of having declarative descriptions of CSP instances and also obtaining their solutions in reasonable computational time. This paper presents from both theoretical and practical points of view the design of a general purpose CSP solver. The solver we have created is called **OmniCS** (Omni Constraint Solver) and is freely available at `http://omnics.sourceforge.net`

1 Introduction

Many problems that are very important from a scientifical or economical point of view prove to be very hard to approach in an uniform manner with usual mathematical techniques, like linear programming for instance. Based on these premises, it was developed the theory of *constraint satisfaction* which defines a model for representing the problems as networks of constraints and also defines algorithms and programming techniques for effective solving of these problems. Papers dedicated to constraint satisfaction appeared since the '70s ([18], [17], [6]), but the CSP area became very

[1] This paper is an extended abstract of the thesis submitted to the "Al. I. Cuza" University of Iaşi for the degree of Doctor of Philosophy in Computer Science.

[2] Faculty of Computer Science, "A.I.Cuza" University, General Berthelot 16, 700483 Iasi, Romania, email: `acf@info.uaic.ro`

active only in the last two decades, today being identified by ACM (Association for Computing Machinery) as a "strategical direction in computer research". In 1997, Eugene C. Freuder said:

> *"Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."*

Because constraint programming received very much attention also from the industry, a lot of *CSP solvers* emerged, i.e. applications that offer solutions to model a problem using constraints and also an engine able to solve it. These solvers are both commercial and open-source and have been developed on different programming platforms like Java, C++, Prolog. To give only a few examples, we can mention Ilog Solver [12], Koalog Constraint Solver[13], Choco [14] or our solver OmniCS [8].

Recently, there have been published a series of monographs dedicated to CSP like [5], [1], [25], [2], [21] which offer a global picture over the whole theory.

This paper presents from both theoretical and practical points of view the process of creating an original CSP solver called OmniCS, that is suitable for any type of problem that can be represented as a CSP instance. The main idea was to analyze the basic techniques and algorithms that were proposed so far in this particular research area, to offer efficient implementations for them and to create a unified framework for modelling and solving constraint satisfaction problems, that is very easy to understand, use and extend. We also approached topics that are hardly found in any CSP solver available on the market as generating explanations, solving multi-criteria optimizations problems or offering an interactive environment between the user and the solver.

The paper is organized as follows. In Section 2 we present the basic definitions and concepts related to constraint satisfaction. In Section 3 there are described all the aspects that must be analyzed when designing a software system able to solve CSP problems. The focus is put on the internal implementation of OmniCS, especially on the control layer that contains the components responsible with the actual solving of a problem.

2 Preliminaries

2.1 Classical networks of constraints

The definitions and notations from this section are taken from [5].
A *k-tuple* (or simply *tuple*) is a sequence of k elements, not necessarily distinct, denoted by $(a_1, ..., a_k)$. The *cartesian product* of the sets $D_1, ..., D_k$ is denoted by $D_1 \times \cdots \times D_k$ and contains all the tuples $(a_1, ..., a_k)$ for which $\forall i : \ a_i \in D_i$.

Given a set of variables $X = \{x_1, ..., x_k\}$, each associated with a domain $D_1, ..., D_k$ respectively, a *k-arity relation* R on the set of variables is any subset of the cartesian product of their domains.

A *constraint network* is a triplet $\mathcal{R} = (X, \mathcal{D}, \mathcal{C})$ where:

- $X = \{x_1, ..., x_n\}$ is a finite set of variables;
- $\mathcal{D} = \{D_1, ..., D_n\}$ represents finite domains that are associated to variables of X;
- $\mathcal{C} = \{C_1, ..., C_t\}$ is a finite set of constraints.

A *constraint* C_i is a relation R_i on a subset of variables S_i, $S_i \subseteq X$, which denotes their simultaneous legal value assignments. We may also make the notations $C_i = (R_i, S_i)$ and $schema(\mathcal{R}) = \{S_1, ..., S_t\}$. If S is a list of variables, we denote $T(S) = \prod_{x_i \in S} D_i$.

In the above definition there is no restriction on the types of variables, their domains may be integers, strings or anything else. There are also no specifications on how constraints are defined. The purpose of a constraint is to restrict the possible values a variable x_i can be assigned from D_i.

Let us consider a simple example of modelling a NP-hard problem, the *graph coloring problem*, as a network of constraints. Let $G = (V, E)$ be a graph and we want to obtain a p-coloring of the vertices V, where p is a positive integer. To represent this problem we create the following network:

- The variables are: $X = \{v_1, ...v_n\}$
- The domains are: $D_i = \{1, ..., p\}$, $\forall i \in \{1, ..., n\}$
- The constraints are: $C_{v_i v_j} = \{c(v_i) \neq c(v_j)\}$, $\forall v_i v_j \in E$

The *instantiation* of a variable is the process of assigning it to a value from its domain. We shall denote $(\langle x_{i_1}, a_{i_1} \rangle, ..., \langle x_{i_k}, a_{i_k} \rangle)$ or

$(x_{i_1} = a_{i_1}, ..., x_{i_k} = a_{i_k})$ or even $\overline{a} = (a_{i_1}, ..., a_{i_k})$ the tuple that represents the instantiation of a set of variables $Y = \{x_{i_1}, ..., x_{i_k}\}$.

Let $\overline{a}$ be a tuple, and let $Y = \{x_{i_1}, ..., x_{i_k}\}$ be a set of variables. We denote $\pi_Y(\overline{a})$ or $\overline{a}[Y]$ the *projection* of $\overline{a}$ onto Y, that is a new tuple created from $\overline{a}$ out of which we eliminate all the components that do not belong to Y.

Let R be a relation, and let $Y = \{x_{i_1}, ..., x_{i_k}\}$ be a set of variables. We denote $\pi_Y(R)$ or $R[Y]$ the *projection* of R onto Y, that is a new relation created from the projections of all the tuples of R onto Y.

We say that an instantiation $\overline{a}$ of the variables $Y = \{x_{i_1}, ..., x_{i_k}\}$ *satisfies a constraint* $C = (R, S)$ if and only if the projection of $\overline{a}$ over S belongs to R: $\overline{a}[S] \in R$

A *solution* of a network of constraints $\mathcal{R} = (X, \mathcal{D}, \mathcal{C})$ is an instantiation of all variables such that no constraint is violated:

$$sol(\mathcal{R}) = \{\overline{a} = (a_1, ..., a_n) \mid (\forall i = 1..n : \ a_i \in D_i) \wedge (\forall j = 1..t : \ \overline{a}[S_j] \in R_j)\}$$

If $\mathcal{R}$ has at least one solution, we say that it is *satisfiable* or *consistent*.

A partial instantiation $\overline{a}$ of a set of variables S is *consistent* if and only if it satisfies all the constraints defined only over variables already instantiated:

$$\forall S_i \in \ schema(\mathcal{R}), \ S_i \subseteq S \ \Rightarrow \ \overline{a}[S_i] \in R[S_i].$$

2.2 Soft networks of constraints

The classical model of constraint satisfaction is defined over the premises that identifying a solution means satisfying all the constraints of the problem. But there are many real life situations that cannot be solved this way, either because they are *over-constrained* ([7], [3]) and thus not consistent or because their restrictions cannot be imposed in a *yes-or-no* manner. A good example of such a problem is the *timetable problem* that has *hard* constraints that must not be violated under any circumstances: students cannot be in two different places at the same time or a shared resource cannot be assigned simultaneously to different activities, but it also has *soft* constraints that represent for instance the *preferences* of the teachers. In a "perfect" solution all these preferences would be satisfied but in most cases this is not feasible and we are concerned in creating a timetable that is "as good as possible", in other words minimizing somehow the number or the magnitude of the constraints that are not satisfied.

A *soft networks of constraints* [22] is a triplet $(X, \mathcal{D}, \mathcal{C})$, where:

- $X = \{x_1, ..., x_n\}$ is a finite set of variables;
- $\mathcal{D} = \{D_1, ..., D_n\}$ are the domains of the variables;
- $\mathcal{C} = \{C_1, ..., C_t\}$ is a finite set of soft constraints.

A *soft constraint* C is a pair (S, f) where $S \subseteq X$ and f is a function defined over the tuples T_S and having values in a set E denoting *levels of preference*: $f : \prod_{x_i \in S} D_i \to E$. We may also note $C = (S, f)$ as f_S.

In order to tell if an instantiation t of the variables S is better than another one t' with respect to satisfying a constraint f_S we define a total ordering $\preceq$ over the levels of preferences E. We also note $\bot$ the bottom of E meaning "allowed tuple" and $\top$ the top of E meaning "forbidden tuple":

$$\forall a \in E \;\; \bot \preceq a \preceq \top.$$

We also define an operator $\oplus$ that "combines" levels of preferences over the set E, specifying how good is an instantiation with respect to satisfying all the constraints of the network. We define the *degree of satisfaction* offered by an instantiation a as a function $F : T(X) \to E$:

$$F(a) = \bigoplus \{f_S(a[S]) \mid \forall f_S \in \mathcal{C}\}.$$

We say that a soft network of constraints has a solution if and only if $\exists t \in T(X)$ such that $F(t) \prec \top$ and such a solution is called *admissible*. A solution t is *optimal* if and only if $\forall t' \in T(X)\ F(t) \preceq F(t')$.
If $\forall t \in T(X)\ F(t) = \top$, the network is not consistent.

A *valuation structure* [24] is a quintuple $\mathcal{S} = (E, \oplus, \preceq, \bot, \top)$ having the meanings defined above.

A *valuated network of constraints* is a quadruple (X, D, C, S), where (X, D, C) is a soft network of constraints and $\mathcal{S}$ is a valuation structure.

By specifying particular types of valuation structures, we obtain different models of constraint satisfaction problems:

CSP	E	$\preceq$	$\bot$	$\top$	$\oplus$
Classical	$\{0, 1\}$	$0 \preceq 1$	0	1	$\wedge$
Additive	$\mathbb{N} \cup \infty$	$\leq$	0	∞	$+$
MAX-CSP	$\{0, 1\}$	$\leq$	0	$\lvert\mathcal{C}\rvert$	$+$

2.3 Solving constraint satisfaction problems

There are many approaches to solving satisfaction problems depending on the nature and the hardness of the problem. In this paper, we are interested in determining if a CSP instance (classical or soft) is consistent and we want to develop a simple but efficient algorithm that will search for exact solutions.

Let $\mathcal{R} = (X, D, C)$ be a constraint network. A *systematic search algorithms* is defined as a finite determinist automaton $\mathcal{A}$ consisting of:

- A finite set S of states, each state corresponding to a partial instantiation of variables from X. S is also called the *search space* of the algorithm and $|S|$ is the *size* of the search space.
- A set O of operators responsible with the transition of $\mathcal{A}$ from one state to another.
- An initial state s_0;
- A set $S_f \subseteq S$ of final states, representing the solutions of the network.

Usually, the search space will be represented as a tree rooted at s_0 and having $|S|$ leaves. A transition from one state to another will signify descending from one level of the tree to the next one. This representation is important because the width of the tree will have a significant impact over the efficiency of the search algorithms.

We say that two constraint networks are *equivalent* if and only if they are defined over the same set of variables and have the same set of solutions. We say that a network $\mathcal{R}' = (X, \mathcal{D}', \mathcal{C}')$ is *easier to solve* than an equivalent network $\mathcal{R} = (X, \mathcal{D}, \mathcal{C})$ and we write $\mathcal{R}' \preceq \mathcal{R}$ if and only if the search space of $\mathcal{R}'$ is "smaller" than the search space of $\mathcal{R}$, that is $\forall x_i \in X \ |D'_i| \leq |D_i|$.

A *filter* is an algorithm that applied to a constraint network $\mathcal{R} = (X, \mathcal{D}, \mathcal{C})$ will reduce its search space, transforming it into an equivalent easier to solve network. Because a filtering algorithm transforms the network it was invoked on, reducing its domains, it is natural to apply it again and again until we obtain an empty domain, a solution or we reach a point where its execution produces no more reductions. We call such a technique *filter-and-propagate*.

A constraint is *redundant* if and only if removing it from the network does not affect the set of solutions, i.e. the new network is equivalent to

the original one. The *inference* is the process of adding to the network new redundant constraints deduced from the existing ones.

Obtaining a solution for a constraint network $\mathcal{R}$ using a systematic search algorithm can be viewed as a finite succession of transformations $\mathcal{R} = \mathcal{R}_0 \rightarrow \mathcal{R}_1 \rightarrow \cdots \rightarrow \mathcal{R}_f$, where $\mathcal{R}_i$ and $\mathcal{R}_j$ are equivalent, $\mathcal{R}_{i+1} \prec \mathcal{R}_i$ and all domains of $\mathcal{R}_f$ are singletons.

The transformation $\mathcal{R}_i \rightarrow \mathcal{R}_{i+1}$ is accomplished by either:

1. An instantiation $x = a$ which reduces the domain of x to $D_x = \{a\}$, or

2. The propagation of a decision $x = a$ using filtering algorithms that will reduce the domains of the network's variables.

$$\mathcal{R}_i \xrightarrow[propagate/filter]{decision} \mathcal{R}_{i+1}$$

The basic structure of the algorithm we have used to develop our solver OmniCS is *backtracking* that uses a flexible filter-and-propagate mechanism in order to obtain efficiency. The challenge we are facing is creating an implementation as general as possible, highly configurable, that could be applicable to any CSP instance, but without inflicting a serious performance penalty.

Let us consider now the case of soft network constraints. It must be noticed that solving these types of problems using systematic search algorithms is much more difficult than in the case of classical model because in the situation where the operator $\oplus$ is not idempotent, as in additive valuation structures, the quality of an assignment is known only when we sum all the penalties induced by the constraints that are not satisfied. In order to create a solver that is able to solve both classical and soft CSP instances we have merged the *backtracking* and the *branch-and-bound* algorithms in a manner that is transparent for the user of the solver.

3 Creating a CSP solver

The objectives we aimed to reach in creating the CSP solver OmniCS were:

- **Uniformity** - to solve both classical and soft constraint satisfaction problems in an uniform manner.

- **Full control** - to provide a mechanism for controlling the whole process of systematically searching the solution.
- **Extensibility** - to be able to:
  - add new filter-and-propagate algorithms special created for the problem we solve.
  - add new simple or global constraints without having to know much about the internal behavior of the solver.
- **Observability** - the solver should generate events as it searches for solutions and inform special objects called *observers* .
- **Explanatory** - when a problem is inconsistent, we want to know why.
- **Dynamic** - to be able to rewrite the problem even when the solver is running, without having to restart the whole process again.
- **Interactive** - to allow human intervention in the search process, overwriting the default behavior of the solver.
- **Simplicity** - creating a CSP model for a problem should be very easy and intuitive.
- **Performance** - even if all the previous requirements may slow down our solver, we want to achieve performances comparable to other similar products.

We conceived our solver based on the *MVC (Model-View-Controller)* design pattern [10], [26]:

- **Model** layer - offers the instruments for modelling a problem as a CSP instance;
- **Control** layer - contains all the components involved in solving the problem:
  - the solver
  - the filtering algorithms
- **View** layer - defines the interface with the user.

For developing the whole framework, we have used the Java programming platform [9].

3.1 The model layer

In designing the model layer we must create an API (Application Programming Interface) that will allow a user to define a problem, specify its variables, their domains and add the constraints.

In order to prove that OmniCS' API is very intuitive and easy to use let us consider the following network of constraints $\mathcal{R} = (X, D, C)$, $X = \{x, y\}$, $D_x = D_y = \{1, 2\}$, $C = \{x \neq y, x < y\}$. The source code that models the network $\mathcal{R}$ is presented below:

```
// Step 1: Create the domains of the variables
Domain domain = new Domain(1, 2);

// Step 2: Create and set the variables
Var x = new Var("x", domain);
Var y = new Var("y", domain);
problem.setVariables(x, y);

// Step 3: Create and set the constraints
problem.addConstraint(new NotEqual(x, y));
problem.addConstraint(new Less(x, y));
```

The domains accepted by OmniCS are heterogeneous, meaning they can contain objects of any type. Variables are identified by unique names and each has an associated domain. All constraints have a common interface and are represented by classes with specific purposes.

In our implementation we impose a total ordering on the values (objects) of each variable's domain and we call this the *natural order* of the domain. This is obtained in the following manner: if two values from one domain are *comparable* (their corresponding classes implement the interface `Comparable`) then the rules defined by their implementations are used to determine their order inside the domain; if they are not comparable, the internal hash codes of the objects will be used to sort them.

Each constraint is defined by a class extended from `Constraint`. There are a lot of commonly used predefined constraints, but creating new ones is very simple - all we have to do is specify how our constraint evaluates a tuple.

Let us consider the implementation of the binary constraint `NotEqual`, which enforces that two given variables cannot have the same value in any solution of the problem:

```
public class NotEqual extends Constraint {
    private Var x, y;

    /**
     * The constructor of the class receives the
     * variables of the constraint.
     */
    public NotEqual(Var x, Var y) {
        super(x, y);
        this.x = x;
        this.y = y;
    }

    /**
     * This method evaluates a given tuple
     * against the constraint.
     */
    public int eval(Tuple tuple) {
        Object a = tuple.get(x);
        Object b = tuple.get(y);
        if (a == null || b == null) return ALLOWED;
        if (a.equals(b)) return FORBIDDEN;
        return ALLOWED;
    }
}
```

The default mechanism for defining the constraints is *declarative* - it only specifies the behavior of the constraint without connecting it to the solving process, and this is a major advantage comparing to other solvers that force the user to "learn" details about the solving process in order to create new constraints.

3.2 The control layer

As we said earlier, the basic structure of the algorithm we have used to develop our solver OmniCS is *backtracking*, which performs a systematic search over the solution space. The general structure of a systematic solver is given in the Algorithm 1.

Algorithm 1 The structure of a systematic solver

Input: $\mathcal{R} = (X, D, C)$ a constraint network
Output: A solution of $\mathcal{R}$

Step0 Apply filters to reduce the initial problem
{This ensures the arc-consistency of the network}
if *the network is solved* **then**
  Process solution
  return
end if
if *the network is inconsistent* **then**
  return
end if

Step1 Save the current state of the problem
Step2 Make a decision {A decision is usually the instantiation of a variable}
Step3 Apply filters to propagate the decision
if *the network is solved* **then**
  Process solution
  return
end if
if *the network is inconsistent* **then**
  Restores the state of the problem before the decision
  goto Step2
end if
goto Step1

Based on this structure we have designed a plug-in mechanism that allows the user to specify filtering algorithms and also the heuristics that control the exploration of the search tree (that is how the solver takes a decision in Step 2). In order to provide a simple and efficient "out of the box" functionality the solver comes with a predefined filter-and-propagate algorithm in the form of the *Global Arc Consistency (GAC)* algorithm.

Let $\mathcal{R} = (X, \mathcal{D}, \mathcal{C})$ be a network of constraints. We say that a variable $x_i \in X$ is *arc-consistent* [5] relative to another variable $x_j \in X$ if and only if $\forall\, a_i \in D_i\ \exists\, a_j \in D_j$ such that the partial instantiation $\{\langle x_i, a_i\rangle, \langle x_j, a_j\rangle\}$ is consistent. In other words, from an algorithmic point of view, assigning

the value a_i to the variable x_i will not immediately create a situation that requires backtracking.

A constraint network is *arc-consistent* if and only if $\forall x, y \in X$, x is arc-consistent relative to y and y is arc-consistent relative to x. Obviously, if a network is not arc-consistent then it has no solution.

Arc-consistency is an important property that must be satisfied by a constraint network in order to achieve efficient systematic search algorithms. The default implementation of our solver uses the AC-3 algorithm in order to maintain the arc-consistency of the network.

Algorithm 2 Arc-consistency(**AC-3**)

Input: $\mathcal{R} = (X, \mathcal{D}, \mathcal{C})$ a constraint network
Output: An arc-consistent network $\mathcal{R}'$ equivalent to $\mathcal{R}$

$\mathcal{R}' = \mathcal{R}$
$queue = \emptyset$
for all $x_i, x_j \in X$, $x_i \neq x_j$ participate in a common constraint **do**
  $queue = queue \cup \{(x_i, x_j), (x_j, x_i)\}$
end for
while $queue \neq \emptyset$ **do**
  Select a pair (x_i, x_j) from $queue$
  $queue = queue - \{(x_i, x_j)\}$
  $revised = Revise(\mathcal{R}', x_i, x_j)$
  if $revised$ **then**
    $\{D'_i$ was reduced$\}$
    $queue = queue \cup \{(x_i, x_k) | k \neq i\}$
  end if
end while
return $\mathcal{R}'$

It is easy to note that the execution of Algorithm 2 is finite, since at every step of the loop either we remove one element from the *queue* or we remove one from some domain. Eventually, the queue or the domains will become empty. According to the definitions, the algorithm generates an equivalent arc-consistent network. However, in the actual implementation, whenever an empty domain is encountered, the algorithm will stop since this proves the inconsistency of the network. An analysis of time complexity can be found in [5].

The *Revise* procedure is responsible with making a variable x_i arc-

consistent relative to x_j, eliminating values from D_i. After its execution we will have: $D_i = D_i \cap \pi_{x_i}(R_{ij})$. The actual implementation of this procedure (described in Algorithm 3) is critical in achieving performance since it will be invoked many times in the process of filtering the domains.

Algorithm 3 The **Revise** procedure

Input: $\mathcal{R} = (X, \mathcal{D}, \mathcal{C})$ a network of constraints, $x_i, x_j \in X$, $x_i \neq x_j$
Output: The domain D_i is filtered such that x_i is arc-consistent relative to x_j; if the domain has been reduced it returns `true`, otherwise `false`.

$R_{ij} = \bigcup\{\pi_{x_i x_j}(R) | R \in \mathcal{C}\}$
$\{R_{ij}$ is the relation of allowed pairs for $x_i, x_j\}$

$revised = false$
for all $a_i \in D_i$ **do**
  if $\nexists a_j \in D_j$ such that $(a_i, a_j) \in R_{ij}$ **then**
    $D_i = D_i - \{a_i\}$
    $revised = true$
  end if
end for
return *revised*

It is easy to prove that the execution of Algorithm 3 is finite, since each value in D_i is compared, in the worst case, with each value in D_j. Details can be found in [5].

3.3 The exploration strategies

The systematic search algorithm must make a series of decisions in order to explore the search space. We define the following configurable strategies:

A **forward strategy** is responsible with selection of the next variable that will be instantiated, thus defining a relation of ordering over the whole set of variables. However, this order is not static and can be specified during execution depending on specific conditions that can be evaluated only at runtime. The interface that defines this strategy is called `ForwardStrategy` and it has some simple implementations like:

- `SimpleForward` - selects variables in the order specified by the definition of the problem.

- `MinDomainForward` - selects first variables with the smallest domains, thus reducing the width of the search space.
- `MostConstrainedForward` - selects variables that appear the most in the network's constraints; instantiating such a variable is likely to determine a better behavior of filter-and-propagate algorithms.

An *assignment strategy* is responsible with defining a relation of ordering over the values of a variable's domain. As in the case of the forward strategy, this relation can be defined dynamically during execution. The interface that defines this strategy is `AssignmentStrategy`. The default implementations are:

- `SimpleAssignment` - selects values according to the natural order of their domain;
- `MaxReductionAssignment` - performs a *forward-checking* operation, determining for each possible value of the current variable the degree of filtering that it produces (i.e. how many values will be removed from the other variables domains), then chooses the value with the highest degree.

A *backward strategy* defines how the solver will select the variable from which it will resume the search process, after a failure was detected. The interface that defines this strategy is `BackwardStrategy` and its default implementation is `SimpleBackward` which returns to the last chronologically variable instantiated before the one that provoked the failure. This strategy is very easy to implement but it has several drawbacks, since it can perform redundant work or it can fail for the same reason multiple times (thrashing). However, we can use other heuristics that can improve this behavior such as `BackJumping` that attempts to identify the real variable whose current instantiation is responsible for the current failure and return to it (avoiding thrashing) or `BackMarking` that maintains a data structure with incompatible assignments that will help the algorithm to avoid redundancy.

The strategies OmniCS offers by default are standard strategies commonly provided by any CSP solver. Our advantage is that we provide a programming interface that allows the user to create new strategies in a very simple fashion and to specify them in a "plug-in" manner even during the solving process. This offers a high degree of flexibility for the general systematic search algorithm and it is very common to define specific strategies for specific problems, in order to effectively solve them.

3.4 Creating global constraints

Let $\mathcal{C} = \{C_1, ..., C_k\}$ be a set of constraints. We define a *global constraint* C_G as the conjunction $C_G = C_1 \wedge C_2 \wedge \cdots \wedge C_k$.

Let $\mathcal{R} = (X, \mathcal{D}, \mathcal{C})$ be a constraint network and $C_G = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ a global constraint. It is easy to observe that $\mathcal{R}$ and $\mathcal{R}' = (X, \mathcal{D}, \mathcal{C} \cup \{C_G\})$ are equivalent, so C_G is a redundant constraint. We say that a filtering algorithm $\mathcal{F}$ is *pertinent* if and only if applied to the network $\mathcal{R}'$ it reduces the search space better than when it is applied to the network $\mathcal{R}$, that is: $\mathcal{F}(\mathcal{R}') \prec \mathcal{F}(\mathcal{R})$.

Apart from the fact that global constraints are more expressive from a syntactical point of view, we are interested in developing special filtering algorithms dedicated to global constraints that will perform better than the general filters applied to the individual constraints. Let us consider a simple example that illustrates this fact. Let $\mathcal{R} = (X, \mathcal{D}, \mathcal{C})$ be a constraint network, where $X = \{x, y, z\}$, $D_x = D_y = D_z = \{0, 1\}$ and $\mathcal{C} = \{x \neq y, y \neq z, z \neq x\}$. The general arc-consistency algorithm will simply do nothing in this case because, no matter how we pick two variables, let's say x and y, for any value $a \in D_x$ there is a value $b = 1 - a \in D_y$ such that the partial instantiation $\{x = a, y = b\}$ is consistent; that means that the network is arc-consistent. Since we cannot reduce the domains, the search algorithm will have to make several wrong decisions in order to detect the inconsistency of the network. However, if we add the global constraint $C_G = x \neq y \wedge y \neq z \wedge z \neq x$, the filtering algorithm will immediately reduce the domain of the starting variable to $\emptyset$ and no backtracking will be required.

The *value graph* [16] associated to a constraint C is a bipartite graph $G = (X, Y, E)$ where:

- $X = X(C)$ - are the variables of C
- $Y = D(X(C))$ - are the values the variables of $X(C)$ may be assigned
- $(x, a) \in E \Leftrightarrow a \in D(x)$

Let us consider the *alldiff* global constraint that imposes that all variables from a specified set must be different. The following proposition gives us a first indication of how should we develop a filtering algorithm for *alldiff*: An *alldiff* constraint C is consistent if and only if the value graph of C contains a matching that covers all the vertices $X(C)$ [20].

It is easy to see that in our example the maximum matching of the value graph is of size 2 and $|X(C)| = 3$, thus the *alldiff* constraint will provide the information that the network is not consistent.

An efficient implementation of a global constraint should offer a specific filtering algorithm that will detect special cases of inconsistency that cannot be identified by the general algorithm. For instance, the actual implementation of the `alldiff` constraint is the class `AllDiff` which contains an instance of `AllDiffFilter`:

```
public class AllDiff extends Constraint {
    public AllDiff(Var ... variables) {
        super(variables);
        setFilter(new AllDiffFilter());
    }
    public int eval(Tuple tuple) {
        // Normal implementation of AllDiff
        ...
    }
}
```

A simple implementation of `AllDiffFilter` would verify the proposition stated above:

```
private class AllDiffFilter extends AbstractFilter {
    ...
    public boolean filter() {
        Graph graph = new Graph();
        // Create the value graph
        ...
        BipartiteMaximumMatching alg =
                new BipartiteMaximumMatching(graph);
        Map<Node, Node> matching = alg.maximumMatching();
        int n = variables.length;
        boolean consistent = (matching.size() == n);
        return consistent;
    }
}
```

4 Solving optimization problems

There are many techniques for solving optimization problems, such as linear programming, that are very efficient for a large class of problems. However, as we have seen, the constraint satisfaction theory offers us an elegant solution for modelling multi-criteria optimization problems as *valuated constraint networks*. This approach is very useful for situations where modelling a problem using standard mathematical techniques is very difficult or even impossible, a good example for such a problem being the timetable problem. In our implementation, each problem must have a *valuation structure* described by the interface `ValuationStructure`:

```
public interface ValuationStructure {
    // The top and bottom elements
    int MIN=0;
    int MAX=Integer.MAX_VALUE;
    int ALLOWED=MIN;
    int FORBIDDEN=MAX;

     // The operator that combines levels of preferences
    int plus(int a, int b);

    // The operator that defines the total ordering
    int compareTo(int a, int b);
}
```

For the classical model we have implemented this interface and we have created the class `ClassicalValuation` that contains only two elements, the bottom ($ALLOWED = 0$) and the top ($FORBIDDEN = \infty$):

```
public class ClassicalValuation implements ValuationStructure{
    // logical AND
    public int plus(int a, int b) {
        return (a == FORBIDDEN ||
                b == FORBIDDEN ? FORBIDDEN : ALLOWED);
    }
    // ALLOWED < FORBIDDEN
    public int compareTo(int a, int b) {
        return a - b;
    }
}
```

In the additive model the domain that represents levels of preferences is formed by the natural numbers and the operators `plus`, respectively `compareTo` are the usual operators for natural numbers. The value $ALLOWED = 0$ means "complete satisfaction" of a constraint, $FORBIDDEN = \infty$ means "not feasible" and the rest of the numbers indicate a "penalty" induced by not satisfying a constraint.

```
public class AdditiveValuation implements ValuationStructure {
    public int plus(int a, int b) {
        return a + b;
    }
    public int compareTo(int a, int b) {
        return a - b;
    }
}
```

Using this approach the evaluation of a tuple is totally independent by the type of problem we solve:

```
    public int eval(Tuple tuple) {
        int cost = 0;
        VarSet vars = tuple.variables();
        ConstraintSet constraints = problem.getConstraints(vars);
        ValuationStructure valuation =
            problem.getValuationStructure();
        for(Constraint constraint : constraints) {
            int eval = constraint.eval(tuple);
            if (eval == ValuationStructure.FORBIDDEN) {
                return eval;
            }
            cost = valuation.plus(cost, eval);
        }
        return cost;
    }
```

Let us consider a classical example of optimization, "the knapsack problem": we have a bag of capacity c, n kind of items, each item i has a *weight* w_i and a *value (profit)* p_i. We want to add as many items in the bag in order to maximize the profit. In order to model this problem as a constraint satisfaction problem, we will consider n variables x_i, $i = 1..n$ that can be

assigned values 0 or 1. The problem has only one hard constraint:

$$\sum_{i=1..n} w_i x_i \leq c$$

The function we have to maximize is:

$$f(x_1, ..., x_n) = \sum_{i=1..n} p_i x_i$$

The technique for modelling this type of problem is determining a maximum value $M = \sum_{i=1..n} p_i$ of the objective function and to create a soft constraint $f(x_1, ..., x_n) = M$ that imposes that we try to approach the maximum as much as possible. The preference level of a tuple $\overline{a} = (x_1 = a_1, ..., x_k = a_k)$ will be $|M - f(\overline{a})|$.

```
public class KnapsackProblem extends Problem {
    public KnapsackProblem() {
        setValuationStructure(new AdditiveValuation());
        int n = 6;
        int w[] = {100, 50, 45, 20, 10, 5};
        int p[] = { 40, 35, 18,  4, 10, 2};
        int c = 100;
        int m = 0;
        for(int i=0; i<n; i++) {
            m += p[i];
        }
        Domain domain = Domain.createIntEnum(0,1);
        Var[] x = createVariables(n, domain);
        addConstraint(new ScalarProductLeq (x, w, c));
        addConstraint(new SoftScalarProduct(x, p, m));
    }
}
```

The solution we obtain for this problem is: $x[0] = 0, x[1] = 1, x[2] = 1, x[3] = 0, x[4] = 0, x[5] = 1$, the optimum being $35 + 18 + 2 = 55$.

Unlike the previous example in which we have modelled a classical combinatorial problem that defines a single objective function, there are many situations that require us to perform *multi-criteria* optimization.

Let us consider the situation of balancing the qualitiy/price ratio of an acquisition. We have n products that we want to purchase, each product

can be bought from several vendors, each vendor having a specific price and quality for that product. We want do buy these products in order to:

- minimize the price
- maximize the quality

Obviously, in that case we have to decide on what degree we focus on quality, and on what degree we focus on price. For each product $i = 1..n$ and each vendor $j = 1..m$ we will note q_{ij} the quality offered and p_{ij} the price (both may be represented in percents, 100 meaning the best price and the best quality). Let x_{ij} be the variable assigned to the pair (i, j) of product and vendor, having values 0 or 1. The soft constraints of our problem will be:

$$\sum_{i=1..n;j=1..m} p_{ij}x_{ij} = 0$$

and

$$\sum_{i=1..n;j=1..m} c_{ij}x_{ij} = 100$$

The hard constraint will impose that we don't buy the same product from different vendors:

$$\forall i = 1..n \sum_{j=1..m} x_{ij} = 1$$

The problem will be modelled with only three constraints:

```
addConstraint(new SoftScalarProduct(x, q, 100));
addConstraint(new SoftScalarProduct(x, p, 0));
addConstraint(new ConstantSum(x, 1));
```

The advantage of our approach to modelling multi-criteria optimization problems is it retains as much as possible of the informal definition of the problems, therefore making it very easy to represent and to solve efficiently.

5 Explaining the inconsistency

Many of the current CSP solvers are not able to offer us a motivation of the fact that a problem has no solution and this is certainly a drawback since the user has to figure out himself the reason of failure: "is the problem over-constrained, is there an error in the model or maybe there is a bug in the

solver's algorithm ?". We would like to obtain answers in a human-readable form that explain why the problem is inconsistent or even why some variable cannot be assigned a certain value.

Recently, some solvers like Choco [14] with the PaLM (Propagation and Learning with Move) extension [4], offer some instruments for generating explanations regarding mainly the current state of the system.

In our solver OmniCS we are interested in developing an algorithm that creates a data structure that maintains a minimal set of information that we could use to prove that a problem is inconsistent, if this is the case.

Let $R = (X, D, C)$ be a constraint network. A failure situation of a systematic search algorithm is identified by a tuple $\overline{a} = (x_1 = a_1, ..., x_k = a_k)$ representing the current partial instantiation from where the search process cannot continue, that is because $\exists y \in X - \{x_1, ..., x_k\}$ such that $\forall b \in D_y$ $(x_1 = a_1, ..., x_k = a_k, y = b)$ is not consistent. Such a tuple is called a *nogood* [19] and we shall note: $C \vdash \neg(x_1 = a_1, ..., x_k = a_k)$ or $\neg(x_1 = a_1 \wedge ... \wedge x_k = a_k)$. For any variable $x_j, j \in [1..k]$ wa may also write: $\bigwedge_{i \in [1..k] \setminus j} (x_i = a_i) \Rightarrow x_j \neq a_j$

We call an *explanation* of a nogood $\overline{a}$ and we note $expl(\neg\overline{a})$ a deductive reasoning having as premises the network $\mathcal{R}$ and the tuple $\overline{a} = (x_1 = a_1, ..., x_k = a_k)$ and as conclusion the fact that the network $\mathcal{R}'$ obtained after reducing the domains $D_1 = \{a_1\}, ..., D_k = \{a_k\}$ is inconsistent. If the tuple has only one element $(x = a)$ we note $expl(\neg(a)) = expl(x \neq a)$.

An explanation algorithm must offer answers to questions like:

- $expl(x \neq a)$: "why a variable cannot be assigned a specific value ?";
- $\bigcup_{a \in D_x} expl(x \neq a)$: "why the problem has no solution ?".

The atomic unit of a reasoning will be the violation of a constraint. For example, let us consider the network $\mathcal{R} = (X, \mathcal{D}, \mathcal{C})$, $X = \{x, y\}$, $D_x = D_y = \{0, 1\}$ having only one constraint $x \neq y$.
In this case, $expl(\neg(x = 0, y = 0)) = \{x \neq y\}$ and no further explanations are required. There are situations when the implementation of a constraint is not trivial or it has additional filtering algorithms. Let $\mathcal{R} = (X, D, C)$, $X = \{x, y, z\}$, $D_x = D_y = D_z = \{1, 2, 3\}$ and C contains the constraint $x + y + z = 9$. The explanation $expl(\neg(x = 1)) = \{x + y + z = 9\}$ may not be very clear because it is based on the internal behavior of the constraint. The result we would like to see is: $x = 1 \wedge max(D_y) = 3 \wedge max(D_z) = 3 \Rightarrow x + y + z \leq 7$. So, if we want to develop an algorithm that generates

explanations we must offer a mechanism that explains inconsistency also at the constraint level.

Let $\mathcal{R} = (X, D, C)$ be a constraint network and $\overline{a} = (x_1 = a_1, ..., x_k = a_k)$ a partial instantiation of some variables. If we want to prove that this tuple is a nogood we have to identify one of the following situations:

(E1) There is a constraint c that is not satisfied by $\overline{a}$;

(E2) There is a variable x such that: $\forall a \in D_x \ \overline{a'} = (\overline{a},\ x = a)$ is a nogood;

(E3) After applying the filter-and-propagate algorithms triggered by the decision $x_i = a_i$, the domain of a variable becomes empty.

Let us explain why the queen-problem on a 3×3 table is inconsistent. In the absence of any filtering algorithm the proof would be formed only using (E1) and (E2) rules:

```
Explain problem is inconsistent
  Explain {x[0]=0} is inconsistent
    {x[0]=0} => x[1] != 0
    {x[0]=0} => x[1] != 1
    Explain {x[0]=0, x[1]=2} is inconsistent
      {x[0]=0, x[1]=2} => x[2] != 0
      {x[0]=0, x[1]=2} => x[2] != 1
      {x[0]=0, x[1]=2} => x[2] != 2
  Explain {x[0]=1} is inconsistent
    {x[0]=1} => x[1] != 0
    {x[0]=1} => x[1] != 1
    {x[0]=1} => x[1] != 2
  Explain {x[0]=2} is inconsistent
    {x[0]=2} => x[1] != 1
    {x[0]=2} => x[1] != 2
    Explain {x[0]=2, x[1]=0} is inconsistent
      {x[0]=2, x[1]=0} => x[2] != 0
      {x[0]=2, x[1]=0} => x[2] != 1
      {x[0]=2, x[1]=0} => x[2] != 2
```

Obviously, for greater values of the size of the table this proof is very hard to understand and it becomes meaningless.

The *support set* offered for an instantiation $x = a$ by some variable y is defined as: $support(x = a, y) = \{b \in D_y | (x = a, y = b)\ consistent\}$. If

there is a variable y such that $support(x = a, y) = \emptyset$ then we can eliminate the value a from D_x.

support	$x[0]$	$x[1]$	$x[2]$
$x[0] = 0$	-	$\{2\}$	$\{1\}$
$x[0] = 1$	-	$\emptyset$	$\{0, 2\}$
$x[0] = 2$	-	$\{0\}$	$\{1\}$
$x[1] = 0$	$\{2\}$	-	$\{2\}$
$x[1] = 1$	$\emptyset$	-	$\emptyset$
$x[1] = 2$	$\{0\}$	-	$\{0\}$
$x[2] = 0$	$\{1\}$	$\{2\}$	-
$x[2] = 1$	$\{0, 2\}$	$\emptyset$	-
$x[2] = 2$	$\{1\}$	$\{0\}$	-

Using the notion of *support set*, the arc-consistency algorithm can be described as in Algorithm 4. Details about Algorithm 4 can be found in [5].

From this perspective, the reason why a value a is eliminated from the domain of a variable x is the complete loss of the support from some variable y: $\exists y \in X\ support(x = a, y) = \emptyset$. If initially $support(x = a, y) = \{b_1, ..., b_t\}$ then: $expl(x \neq a) = expl(y \neq b_1) \cup expl(y \neq b_2) \cup \cdots \cup expl(y \neq b_t)$ This mechanism allows us to prove inconsistency in a manner that reveals dependencies between variables. For the previous example we would obtain the following explanation:

```
Explain problem is inconsistent
   Explain x[2] != 0
     - support(x[2]=0, x[0])=[1]
       Explain x[0] != 1
         - support(x[0]=1, x[1])=[]
   Explain x[2] != 1
     - support(x[2]=1, x[1])=[]
   Explain x[2] != 2
     - support(x[2]=2, x[0])=[1]
       Explain x[0] != 1
         - support(x[0]=1, x[1])=[]
```

Based on this idea, we have integrated in our solver an explanation algorithm that runs as an observer of the solving process gathering data required to prove the inconsistency of a problem.

Algorithm 4 Arc-consistency (**(AC-4)**)

Input: $\mathcal{R}_0 = (X, D, C)$ a constraint network
$\mathcal{R} = \mathcal{R}_0$
Create $\mathcal{S}$ the set of all the support sets
$queue = \emptyset$
{$queue$ is the list of empty support sets}
for all $E = support(x = a, y) \in \mathcal{S}$ **do**
  if $E = \emptyset$ **then**
    $queue = queue \cup \{(x, y, a)\}$
  end if
end for
while $queue \neq \emptyset$ **do**
  Select a triplet (x, y, a) from $queue$
  $queue = queue - \{(x, y, a)\}$
  $D_x = D_x - \{a\}$
  if $D_x = \emptyset$ **then**
    {The network is inconsistent}
    return *null*
  end if
  for all $A = support(z = c, x) \in \mathcal{S}$ **do**
    $A = A - \{a\}$
    if $A = \emptyset$ **then**
      $queue = queue \cup \{(z, x, c)\}$
    end if
  end for
end while
return $\mathcal{R}$

5.1 Interactive and dynamic problems

Traditionally, the algorithms for solving constraint satisfaction problems have been designed considering that the network created after the modelling phase is in its final form, that is assuming that it is static during the solving process. In a series of papers dedicated to the field of artificial intelligence, E. Lamma [15], M. Gavanelli [11] and others proposed an interactive framework for modelling constraint satisfaction problems (ICSP) in which the domains of the variables are populated dynamically while the solving algorithm is executed.

In the timetable problem we have the following situation: there are hard-constraints that must not be violated and there are also soft constraints representing the preferences of the participants. Suppose that after gathering all these preferences we start the solver and wait to find the solution; unfortunately, depending of the size of the problem, this may require some not so short period of time. In a real life situation, it is not uncommon that in the middle of the solving process we receive additional information about the preferences of some participant that must be added to the existing ones. So, what do we do ? It would be very frustrating if we had to start the whole process again, wasting all the computational effort performed so far.

In the paper "A query-the-user facility of logic programming" [23] M. Sergot makes the following statement: "It is unreasonable and unrealistic to force the user to anticipate and supply all the information of the problem in advance". An efficient system should offer the possibility to change the problem dynamically.

Thus, a CSP solver should offer the following possibilities:

- to stop the execution and resume it at a later moment;
- to save the current state of the solver in a persistent form with the possibility to restore it later;
- to extend the current partial solution with external decisions, manually made by the user;
- to cancel decisions made by the solver;
- to eliminate (inactivate) some variables;
- to dynamically add new variables;

- to dynamically add, modify or remove constraints.

In solving the timetable problem, these feature proved to be very valuable.

We call the *state* or *simple state* of a CSP solver a structure $S = (\alpha, \delta, \pi, \omega)$ where:

- α represents the candidate variables (not instantiated yet);
- $\delta(x)$ is the domain of x, $\forall x \in X$;
- π is the stack of forward decisions, that is all the variables instantiated already and, at the top, the current variable;
- ω is the tuple representing the current partial solution.

The state of the initial network that has to be solved is:

$$(S_0) : \alpha_0 = X, \delta_0(x_i) = D_i \, \forall i = 1..n, \pi_0 = \emptyset, \omega_0 = \emptyset.$$

A state $(\alpha_f, \delta_f, \pi_f, \omega_f)$ is *final* if and only if:

$$(S_f) : \alpha_f = \emptyset, |\delta_f(x_i)| = 1 \, \forall i = 1..n, \pi_f = X, var(\omega_f) = X.$$

A solver S assigned to a network $\mathcal{R}$ is *correct* if and only if for any sequence of states $S_0 \rightarrow \cdots \rightarrow S_f$ such that S_f is final then ω_f is consistent.

A solver S assigned to a network $\mathcal{R}$ is *complete* if and only if for any solution ω of $\mathcal{R}$ there is a sequence of states $S_0 \rightarrow \cdots \rightarrow S_f$ such that $\omega_f = \omega$.

The default implementation of a CSP solver should be correct and complete. However, if we allow the user to make external decisions we may loose these properties - so additional mechanism are required in order to prevent that.

Let S be a solver assigned to a network $\mathcal{R}$ and $S_0 \rightarrow \cdots \rightarrow S_i$ the simple states corresponding to the partial problems generated by the decisions taken until the current moment. The set $\mathcal{S}_i^* = \{S_0, ..., S_i\}$ is called the *extended state of the solver* S.

In order to implement a *stop-resume* mechanism we have to start the solving algorithm in its own thread, as a *daemon (worker)* that will depend by another thread called *controller* [9].

```
Thread worker = new Thread(new Runnable() {
    public void run() {
        // start the solver
        solver.solve();
    }
});
worker.setDaemon(true);
worker.start();
```

We must also implement a method that notifies the algorithm's thread that it must stop or resume execution. OmniCS offer the method `setPause` that is an accessor of a shared variable that controls this aspect. Also, the main method of the solver must check the control variable in order to stop its execution:

```
while (running) {
    synchronized(this) {
        while (paused) {
          wait();
        }
        if (!running) break;
    }
...
}
```

Once we have implemented a mechanism that allows the user to stop and resume the solving process, it is not very difficult to offer methods that can alter the current simple or extended state of the solver and manually control different aspects of the solving process. All we have to do is carefully update the data structures $S = (\alpha, \delta, \pi, \omega)$ maintained by the algorithm in order to take into consideration the changes made by the user.

Another aspect we have to analyze is the situation when solving a problem takes a very long time (hours, even days). There is a risk that at some moment, because of a hardware failure our computational effort would be lost. OmniCS uses the standard serialization mechanism in order to save the extended state of the solving process on some external device. Using the *save-restore* facility is straightforward:

```
// Create an initial solver for the problem
solver = problem.createSolver();
```

```
...
// Save the extended state into a file
solver.save(filename);
...
// Create a new solver that resumes the execution of the previous
// using the extended state saved in the file
solver = problem.createSolver(filename);
```

6 Conclusions and future work

This paper presents an original approach in developing a framework that allows a user to model and solve constraint satisfaction problems, both classical and soft instances. The solver we have created is called OmniCS and was developed on the Java platform using state of the art programming techniques and software design patterns. The algorithmic layer of the solver is very effective "out of the box" using a simple but effective solution for maintaing arc-consistency of the constraint network being solved, but it is also highly configurable and allows the users to "plug-in" filtering algorithms designed for a particular problem or to control the exploration strategies of the search space using custom heuristics.

Compared to other similar open-source solvers (like Choco or Minion), the main advantages of OmniCS are its simple but efficient architecture that brings together in an unified framework standard constraint programming techniques and algorithms, a very easy to use application programming interface that allows the users to model CSP problems in a natural way and also to extend or override the default behavior of the solver, if that is necessary.

As future research direction we are interested in creating a mechanism for solving constraint satisfaction problems in a distributed manner, that will allow us to deploy our solver in a grid architecture. From this perspective, we are also working on developing an XML-based protocol for specifying CSP instances in a declarative, standard way.

Each aspect of improving a CSP solver rises difficult challenges, both from theoretical and practical point of views, but this will only motivate us in our attempt to continue and refine the results obtained so far.

References

[1] K. Apt. *Principles of constraint programming*. Cambridge University Press, 2003.

[2] R. Bartak. On-line guide to constraint programming. http://kti.ms.mff.cuni.cz/ bartak/constraints/index.html.

[3] S. Bistarelli. *Semirings for Soft Constraint Solving and Programming*. Springer, 2004.

[4] Y. Caseau and F. Laburthe. Palm. http://www.e-constraints.net/palm.

[5] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[6] E. C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.

[7] E. C. Freuder. Partial constraint satisfaction. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, IJCAI-89, Detroit, Michigan, USA*, pages 278–283, 1989.

[8] C. Frăsinaru. Omnics. http://omnics.sourceforge.net.

[9] C. Frăsinaru. *Curs practic de Java*. Matrix Rom Bucuresti, 2005.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[11] M. Gavanelli, E. Lamma, P. Mello, and M. Milano. Performance measurement of interactive CSP search algorithms. In *Giornata di Lavoro RCRA su "Analisi sperimentale di algoritmi per l'Intelligenza Artificiale"*, Rome, Italy, Dec 16 1999.

[12] Ilog. Ilog solver. http://www.ilog.com.

[13] Koalog. Koalog constraint solver tutorial. http://www.koalog.com.

[14] F. Laburthe and N. Jussien. Choco. http://choco.sourceforge.net.

[15] E. Lamma, P. Mello, M. Milano, R. Cucchiara, M. Gavanelli, and M. Piccardi. Constraint propagation and value acquisition: Why we should do it interactively. In *IJCAI*, pages 468–477, 1999.

[16] J.-L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence. An International Journal*, 10(1):29–127, 1978.

[17] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[18] U. Monatanari. Networks of constraints: Fundamental properties and application to picture processing. *Information Science*, 7(2):95–132, 1974.

[19] P. Boizumault N. Jussien, R.Debruyne. Mantaining arc-consistency within dynamic backtracking. In *Sixth international conference on principles and practice of constraint programming (CP'2000)*, 2000.

[20] J.-C. Regin. Global constraints. First International Summer School on Constraint Programming Acquafredda di Maratea – Italy, September 11-15 2005.

[21] F. Rossi, P. van Beek, and T. Walsh (editors). *Handbook of Constraint Programming*. Elsevier, 2006.

[22] T. Schiex. Soft constraint processing. First International Summer School on Constraint Programming, July 2005.

[23] M. Sergot. A query-the-user facility of logic programming. In *In Degano, P., Sandwell, E., eds.: Integrated Interactive Computer Systems, North Holland*, pages 27–41, 1983.

[24] G. Verfaille, T. Schiex, H. Fargier. Valued constrained satisfaction problems: hard and easy problems. *Proc. of the 14th IJCAI*, pages 631–637, 1995.

[25] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[26] J. W.Cooper. *The Design Patterns Java Companion*. Addison-Wesley, 1998.