A Framework for Constraint Satisfaction

Pierre Roy[1], Anne Liret[2] and François Pachet[3]

[1]LIP6, University Paris 6, FRANCE, Pierre.Roy@lip6.fr

[2]LIP6, University Paris 6, FRANCE, Anne.Liret@lip6.fr

[3]SONY-CSL Paris, FRANCE, pachet@csl.sony.fr

Abstract: This paper discusses the relevance of the framework approach for building efficient and powerful constraint satisfaction programming environments. Constraint satisfaction programming is a paradigm for solving complex combinatorial problems. Integrating this paradigm with objects addresses two different objectives. On the one hand, objects may be used to implement efficiently constraint satisfaction algorithms. On the other hand, objects can be used to state and solve complex constraint problems more easily, and more efficiently. Traditional systems offering an integration of the two paradigms use a language-based approach, i.e. are extensions of existing programming languages, which integrate both paradigms in various ways. In this chapter, we argue that the framework approach is more adapted to the requirements of embedded object oriented constraint satisfaction than the language-based approach. We propose such a framework for stating and solving constraint problems involving object, and illustrate it on various examples.

Constraint satisfaction programming is a powerful paradigm for solving combinatorial problems, which was initially seen as an *algorithmic* issue [19, 20]. The first proposals for integrating constraints in a language were developed within the community of logic programming. Constraint logic programming (CLP) was primarily designed to deal with specific computation domains like integer numbers. Its best known representatives are PROLOG III [10], CHIP [3] and CLP (FD) [9].

Using CSP from within a programming language is a definitive advantage, compared to the situation where the user must call an external system. Depending on what the language offers to the user, the integration of CSP may take the three forms reviewed below: library, language constructs, or framework.

Library of generic constraints

In this approach the objective is to identify generic constraints that can be used in a wide range of applications, (e.g. global constraints in CHIP [3]). This approach is adapted to classical problems, and in this case the only task is to formulate the problem in terms of the predefined constraints. This can be summed up by the phrase “constrain and solve”. For specific problems, since constraints are complex and domain independent, this formulation may be hard to find.

The language construct approach

This approach is illustrated by CLAIRE [8], a language for building constraint solvers. CLAIRE does not propose any predefined resolution mechanisms, but integrates general and efficient low-level constructs that can be used to build specific solvers (i.e. a save/restore and a forward chaining rule mechanisms). This approach can be seen as the opposite of the library approach: the user has a lot to do, but ends up with a efficient algorithms. This is well suited to hard problems not identified as instances of well-known classes of problems.

The framework approach

The framework approach is an intermediary position. It comes from works aiming at integrating *object-oriented languages* with constraints. Rather than providing specific computation domains as for CLP, the interest of integrating constraints and objects is to provide extensible and flexible implementations of CSP (e.g. COOL [1], ILOGSOLVER [26], LAURE [7]). Besides, objects provide facilities for *domain adaptation*. One particularly efficient way to achieve domain adaptation is to provide *frameworks* [13] in which 1) general control-loop and mechanisms are coded once for all, and 2) adaptation to specific problems can be achieved easily. More than a class library, a framework is a “semi-complete” application containing integrated components collaborating to provide a reusable architecture for a family of applications. We now outline the features of such a framework.

1. From Theory to Practice

In this section, we introduce briefly the basic concepts underlying constraint satisfaction as defined in the technical and theoretical literature. For the sake of simplicity, these theoretical works are based on a simplified model of constraint satisfaction, both from a technical and a conceptual viewpoint. We emphasize the fact that this simplified model is not adapted to the context of real-world applications, and argue in favor of a non-restrictive model of CSP. The implementation of this richer model of CSP is addressed, using the object-oriented framework approach, in the following section.

1.1 Finite Domain Constraint Satisfaction: Basic Notions

Stating a combinatorial problem as a CSP amounts at characterizing *a priori* what properties a solution should satisfy. A finite domain constraint satisfaction problem is defined by a set of *variables*, each variable taking its value in a finite set, its *domain*; and by a set of *constraints*, defining the properties of the solutions. A *solution* is an instantiation of all the variables satisfying every constraint.

A naive resolution mechanism may be described as follows (see for instance [25]):

Basic Enumeration Algorithm (BEA)

1) Choose a non-instantiated variable V of the problem.
2) Instantiate V with a value of its domain, and save the current state of the problem.
3) Check all instantiated constraints. If a constraint is violated, then backtrack to a previously saved state of the problem.
4) If all the variables are instantiated then a solution has been found
    i) Yield the solution.
    ii) Go backward to a previously saved state.
1) Go to 1).

Figure 1. The basic enumeration algorithm (BEA) for constraint satisfaction problems

This procedure yields all the solutions of the problem, and is the core of all complete algorithms for finite domain constraint satisfaction. Of course this basic algorithm is very inefficient. We will review now the main improvements of these basic algorithms as they are described in the literature.

1.1.1 Arc Consistency

In the BEA, a constraint is used once all its variables are instantiated. The following example shows that constraints can also be used, actively, to anticipate dead-ends. Consider for instance two variables X and Y whose domains are $\{1,2,\ldots,10\}$, and the constraint "$X > Y$". If X is instantiated with 3, the constraint can be used *right away* to *reduce* the domain of Y to $\{1,2\}$. This domain reduction prevents the algorithm from checking 3, 4, …, 10 for Y. Note that *this domain reduction does not discard any solution*.

Domain reduction is the main tool for pruning branches of the search tree developed by BEA. The maximum amount of "safe" domain reduction is determined by the property of *arc consistency* [5, 20]: a binary constraint C holding on variables X and Y, is *arc-consistent* if, and only if

$$\forall x \in Dom(X)\ ;\ \exists y \in Dom(Y) \text{ such that } C(x, y) = true$$
$$\text{and } \forall y \in Dom(Y)\ ;\ \exists x \in Dom(X) \text{ such that } C(x, y) = true.$$

Informally, a constraint is arc-consistent, if every value of the domain of a variable appears in at least one consistent tuple of the constraint. This definition generalizes easily to non-binary constraints. A constraint can be made arc-consistent by removing values in the domains of its variables. More precisely, arc consistency, for a constraint C that involves variables X and Y, can be enforced as follows:

While C is not arc-consistent do
$\quad \forall x \in \mathrm{Dom}(X); [\forall\ y \in \mathrm{Dom}\ (Y), \mathrm{C}\ (x, y) = \text{false} \Rightarrow \mathrm{Dom}\ (Y) \leftarrow \mathrm{Dom}(Y) \setminus \{y\}]$
$\quad \forall y \in \mathrm{Dom}(Y); [\forall\ x \in \mathrm{Dom}\ (X), \mathrm{C}\ (x, y) = \text{false} \Rightarrow \mathrm{Dom}\ (X) \leftarrow \mathrm{Dom}(X) \setminus \{x\}]$

A CSP is said to be *arc-consistent* if all its constraints are arc-consistent. We wrote above that a constraint can be made arc-consistent by reducing the domains of its variables, and that arc consistency is "safe", in the sense that it does not discard any solution. Therefore, arc consistency can be used during the execution of an enumeration algorithm to speed up the search by reducing the problem. This idea leads to much more efficient algorithms, as explained in the next section.

1.1.2 Arc Consistency-Based Algorithms

The most efficient algorithms for solving CSPs are based on the BEA, augmented with arc consistency, which is used to reduce the problem during the search. The main difference between these algorithms lies in the *amount* of arc consistency enforced. For instance, the algorithm called *real full look-ahead* reduces the problem using arc consistency as much as possible. Of course, enforcing arc consistency of the whole problem requires a considerable amount of computation time. One could say that this method *"slowly explores a small search space"*.

Real Full Look-ahead (RFL)
1) Choose a non-instantiated variable V of the problem.
2) Instantiate V with a value of its domain, and save the current state of the problem.
3) **Enforce arc consistency for the whole problem.**
If a variable domain is wiped out, then backtrack to a previously saved state of the problem.
4) If all the variables are instantiated then a solution has been found (stop).
5) Go to 1).

Figure 2 Real full lookahead algorithm

In the algorithm called *forward checking* (see Figure 3), the reduction consists in enforcing arc consistency only for constraints involving the last instantiated variable. Compared to real full look-ahead, this method prunes less branches of the search tree, but it spends less time during domain reduction phase: this algorithm *"quickly explores a large search space"*.

Forward-Checking (FC)
1) Choose a non-instantiated variable V of the problem.
2) Instantiate V with a value of its domain, and save the current state of the problem.
3) **Enforce arc consistency for the constraints involving V.**
If a domain is wiped out then backtrack to a previously saved state of the problem.
4) If all the variables are instantiated then a solution has been found (stop).
5) Go to 1).

Figure 3 Forward-checking algorithm

Many other algorithms have been devised that all fit in with this scheme. Differences concerns the amount of propagation performed, and the backtracking strategies applied. We will now review several limitations of this basic model in the context of real-world constraint satisfaction problems.

1.2 Restriction to Binary Problems

A binary CSP is a problem whose constraints hold on at most two variables. A theoretical result is that, for every CSP, there exists an equivalent binary CSP "Equivalent" here means that there exists a one-to-one mapping between the solutions of the two problems. Based on this equivalence, most of the theoretical and technical works limit themselves to the study of binary CSPs, assuming that the equivalent binary CSP retains all the properties of the original CSP.

This limitation is very restrictive, as illustrated by the following problem, and its statement as a binary CSP:

In the addition SEND+MORE=MONEY, replace each letter by a number between 0 and 9 so that:
1) S and M are positive.
2) S, E, N, D, M, O, R and Y are pairwise different.
3) The numeric addition obtained after replacing each letter by the associated number is correct.
There is only one solution to this problem: 9567 + 1085 = 10,652.

If one has to define this problem as a binary CSP, the resulting statement will look like the following:

Variables:	*s*, *e*, *n*, *d*, *m*, *o*, *r* and *y* whose domain is {0, 1, …, 9}.
Constraints:	$s \neq e$; $s \neq n$; $s \neq d$; …*etc*… $r \neq y$
Variables:	*se, nd, mo, re, on* and *ey* whose domain is {0, 1, …, 99}
Constraints:	$nd \equiv d$ (10) "*nd* equals *d* modulo 10" $[nd\ /\ 10] = n$ "where [.] denotes the integer part of a real number" $se \equiv e$ (10) and $[se\ /\ 10] = s$ $re \equiv e$ (10) and $[re\ /\ 10] = r$ $mo \equiv o$ (10) and $[mo\ /\ 10] = m$ $on \equiv n$ (10) and $[on\ /\ 10] = o$ $ey \equiv y$ (10) and $[ey\ /\ 10] = e$
Variables:	*send, more* and *oney* whose domain is {0, …, 9 999}
Constraints:	$send \equiv nd$ (100) and $[send\ /\ 100] = se$ $more \equiv re$ (100) and $[more\ /\ 100] = mo$ $oney \equiv ey$ (100) and $[oney\ /\ 100] = no$
Variable:	*money* with domain {0,…,99 999}
Constraints:	$money \equiv oney$ (1,000) and $[money\ /\ 1{,}000] = m$
Variable:	$sendmore \in$ {(0,0),(0,1),…,(0,9999),(1,1),…,(1,9999),…,(9999,0),…,(9999,9999)}
Constraints:	$sendmore1 = send$ and $sendmore2 = more$
Variable:	*sendplusmore* whose domain is {0,…,19 998}
Constraint:	$sendplusmore = (sendmore1 + sendmore2)$

This representation is very cumbersome: it requires 49 binary constraints, 20 variables and 100,000,000 domain values. We will see in Section 2.2.3 a more compact representation using non-binary constraints.

1.3 Intension vs. Extension

A constraint can be seen as a Boolean relation holding between variables. In the case of finite-domain CSP, this relation can be expressed either in *extension* or in *intension*. Defining a constraint in extension consists in providing the set of consistent tuples of values (see example below). Defining a constraint in intension consists in providing a formula of satisfaction.

For instance, the constraint C holding on two variables X and Y, which requires that the value of X should be greater than the value of Y can be either defined, in intension, by the formula "$X > Y$", or in extension by the set: $Ext(C) = \{\, (x, y) \in Dom(X) \times Dom(Y) \mid C(x, y) = true \}$. For instance, assuming that X and Y have domain {1, 2, 3}, the extension of constraint "$X > Y$" is {(2, 1), (3, 1), (3, 2)}.

The extensional representation of constraints is motivated by the desire to interpret CSP in the context of set theory. Indeed, it is relatively easy in an extensional context to describe algorithms, and to prove various properties. However, in practice, the extensional representation of constraints raises several issues:

1) It requires an important memory space, especially for non-binary constraints, because the size of the Cartesian product of the domains grows exponentially with the number of variables.
2) In many cases, constraints are more naturally expressed in intension than in extension. Additionally, evaluating a formula is often more efficient than checking that a tuple belongs to a set.
3) Representing constraints in extension is well suited to brute combinatorial reasoning, but is not adapted for higher-level reasoning, such as formal reasoning.

1.4 Arc Consistency vs. Constraint Filtering

As we saw in Section 1.1.2, arc consistency is used, during the resolution, to reduce the size of the domains after each instantiation. Unfortunately, enforcing arc consistency for a constraint is expensive since it requires, to compute the Cartesian product of the domains. To address this issue, arc consistency is, in practice, replaced by a weaker concept: *constraint filtering*. Filtering a constraint consists in performing only domain reductions that are reasonably computable.

Constraint filtering ranges within two extremes. On the one hand, the "upper limit" for constraint filtering consists in enforcing strict arc consistency, because enforcing more than arc consistency leads to discarding solutions. On the other hand, the "lower limit" for constraint filtering consists in checking satisfiability once all the variables are instantiated, with no domain reduction. Indeed, this is a limit because otherwise the solver would provide solutions that do not satisfy every constraints.

Just checking satisfiability ≤ Constraint filtering ≤ Arc consistency

The main idea behind filtering is that its depends on the constraint considered. For instance, for constraint "$X > Y$", arc consistency is enforced, because only lower and upper bounds have to be considered. The corresponding implementation using the BACKTALK solver is given in Section 2.3.1.

Conversely, for constraint "$X + Y = 0$", enforcing arc consistency is be very expensive, since all possible values for X and Y have to be considered. In this case, a good filtering method consists in considering only the bounds of X and Y. This is a good filtering because 1) it realizes almost arc consistency and 2) it is efficient, since only the bounds are considered, instead of the whole Cartesian product $Dom(X) \times Dom(Y)$.

Consequently, one of the most important issues of constraint solving is to define filtering methods that are efficient and as close as possible to full arc consistency. Of course, these two properties are conflicting, so the real issue is to find the right compromise.

1.5 Enumeration Algorithms

There is a profusion of solving algorithms [11, 25] and each of them is adapted to specific situations, and none of them is always better than the others. For instance, full look-ahead is especially efficient when the filtering of a constraint is cheap and when there are strong dependencies between variables. Forward checking is interesting for "weakly constrained" problems, especially when constraint are hard to filter. Unfortunately, these criteria are hard to specify formally, and thus to automate. In practice, what is needed is to adapt an existing algorithm to specific problems.

Another problem with the profusion of published algorithms is that they are usually described in the scope of binary CSP, and it is not easy to adapt them in our context. Trying to implement an exhaustive library of algorithms is therefore unrealistic, because there are virtually as many algorithms as there are problems! This situation led us to design a single resolution mechanism (see Section 2.2.4), with support for implementing specific resolution algorithms adapted to specific problems.

1.6 No Utilization of Knowledge on Variable Values

Several works explored the possibility of exploiting the properties of domains seen as specific types (e.g. order-sorted domains [6]). However, resolution algorithms implicitly assume no particular properties on the domain *values*. Values are considered as reducible to atomic entities.

Note that, since no hypothesis on the domains values are made, solving CSPs in which domains are collections of arbitrary objects does not raise any technical issue, as far as a language with pointers is used. However, in our case, we claim that the structure and properties of objects involved in real world problems can be exploited by the resolution process to compute solutions faster and to state problems more easily.

For instance, CSP techniques have been used to produce musical harmonization automatically. Such applications developed so far do not take advantage of properties of the musical structures handled. More precisely, complex musical structures, such as chords and melodies, are handled as mere collections of atomic values. We have shown that such an approach leads to building inefficient and bloated applications. We claim that objects can be used to represent the structures of the problem explicitly as values for the variables, and that doing so allows to improve the efficiency as well as the design of the resulting application. This point constitutes the main motivation in integrating objects with constraint satisfaction and it will be developed in Section 2, and an illustration is presented in Section 3.3.

To summarize, we outlined five restrictions of the standard theoretical constraint satisfaction model, in the context of practical, real world object-oriented problems, which are:

1. Restriction to binary constraints
2. Constraints defined in extension
3. Complexity of arc consistency
4. Profusion of algorithms
5. No utilization of knowledge on variable values

These limitations motivated the design of the BACKTALK framework, which we will now describe. As outlined in the introductory section, BACKTALK is considered according to two viewpoints: the technical aspects (points 1 to 4) are examined in Section 2. Point 5 is considered in Section 3, where we discuss the statement of problems involving objects.

2. BACKTALK: the Framework Approach for Implementing Constraints

BACKTALK [28] is a framework for constraint satisfaction: it consists of a library of classes representing the concepts of variables, domains, constraints, problems and solving algorithms, linked together by a propagation mechanism The requirements expressed in the preceding section prevailed throughout the design of BACKTALK, which provides predefined high-level non-binary constraints and implements sophisticated filtering mechanism as well as a resolution algorithm adaptable to specific cases.

2.1 Integration of BACKTALK in the Host Language

BACKTALK was designed as a traditional Smalltalk framework, with no modifications of the virtual machine. In other words, BACKTALK is implemented *without kernel support*, so that it can be run on any platform supported by Smalltalk. Moreover, BACKTALK introduces no syntactic extension to Smalltalk: stating and solving problems is done using standard mechanisms, class instantiation and message sending.

2.2 Overall Design

BACKTALK is implemented as a library of related classes, which we can classify into three main clusters: 1) variables, 2) constraints and 4) solving algorithms.

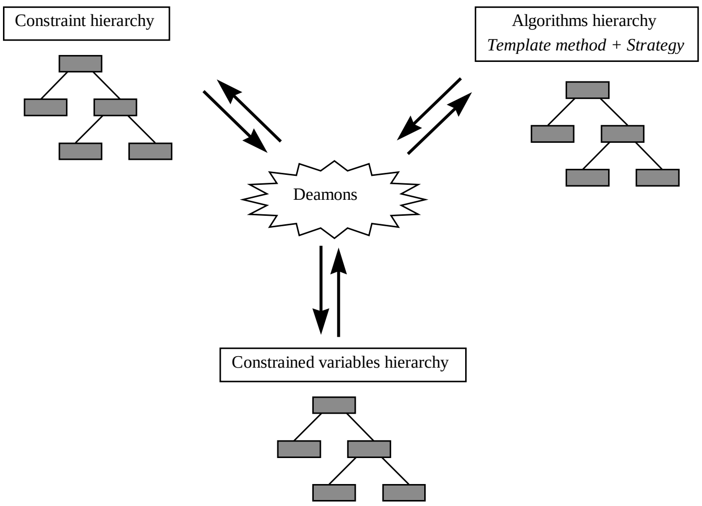


Figure 4 The overall design of the BACKTALK system. Three main hierarchies provide constrained variables, constraints and algorithms. Constraints, algorithms and variables are linked together by demons that are used to implement propagation mechanisms efficiently.

2.2.1 Constrained Variables

Constrained variables are the basic building blocks of the framework. They represent the notion of an unknown value, to be computed by the solver. Practically, it is the responsibility of constrained variables to inform the solver of domain modifications. A variable is defined by a *domain*, a *value* within the domain, and a *label* for display purposes.

The behavior of constrained variables depends heavily on the nature of their domain, therefore we reified variables into classes, and organized them into the hierarchy:

```
BTVariable ('domain' 'constraints' 'valueDemons' ...)
    BTBooleanVariable ()
    BTIntegerVariable ('min' 'max' 'minEvent' 'minDemons' ...)
        BTRealVariable ('precision')
    BTObjectVariable ('domainDictionary' 'actualDomain' ...)
        BTOrderedObjectVariable ('minEvent' 'minDemons' ...)
```

It is important to note that the user does not have to be aware of this class hierarchy: BACKTALK automatically chooses the most adapted class to instantiate, depending on the domain provided. We give below examples of BACKTALK expressions, with their evaluation ("V" is a shortcut for `BTVariable`), illustrating the behavior of constrained vairables.

```
"Creation of an integer variable, from the bounds of an interval"
x := V label: 'x' from: 1 to: 10 => ['x' int[1..10]]

"Modifying the bounds of the domain of an integer variable"
x min: 3; max: 4 => ['x' int[3..4]]

"Creation an Integer Variable from a collection of integers"
V label: 'Var' domain: #(1 6 7 8 9),#(2 3 18 0 -5) => ['Var' int[-5 0..3 6..9 18]]

"Creation an Object Variable from the collection of all Points"
V label: 'obj' domain: (Point allInstances) => ['obj' (0@0 0@1 142@274 0@4 etc...)]
```

Constrained variables are also responsible for a part of the resolution mechanism (the demon mechanism). This aspect will be emphasized in Section 2.3.

2.2.2 Expressions and Constraints

In the intensional model of constraint, constraints may be seen as "links" between several variables (instead of sets of consistent tuples). Moreover constraint filtering depends on the nature of the constraint considered, which leads naturally to the idea of organizing constraints into a hierarchy of classes. The root of this hierarchy is an abstract class, `BTConstraint`. This class implements a default filtering algorithm, which consists in enforcing arc consistency. This default filtering is redefined in subclasses to implement specific methods, achieving a better compromise between reduction and efficiency.

To define a new class of constraint in BACKTALK one needs to create a subclass of class `BTConstraint` or of any of its subclasses. Because all constraints are defined in intension in BACKTALK the new class will redefine the formula of satisfaction.

Of course, defining specific filtering methods is not necessary because of inheritance. Indeed, when the user defines a new constraint class, it benefits automatically from the filtering method defined in its superclass. In the less interesting cases, the default filtering mechanism will be used by the newly defined constraint class. Specific filtering methods are only used to implement more efficient filtering mechanisms.

2.2.2.1 Hierarchy of Constraints

Predefined BACKTALK constraints include arithmetic constraints, cardinality, difference and logical constraints. There are also particular predefined constraints, dedicated to using specific object-oriented mechanisms, that will be presented in Section 3.2. One important aspect of organizing constraints in such a hierarchy is to provide the user with many ready for use constraints. Moreover, the user does not have to know anything more than the class of the constraint to instantiate, and the appropriate creation method.

For instance, the constraint "$X \neq Y$" will be stated by the following class instantiation message, where X and Y denote constrained variables:

```
BTAllDiffCt on: X and: Y
```

Other constraints are created similarly, using, as far as possible, the same creation interface. For instance, the constraint "$X + 2.Y - 5.Z \geq 0$" will be stated by the following class instantiation message, where X, Y and Z denote constrained variables:

```
BTLinearGreaterOrEqualCt
    on: (Array with: X with: Y with: Z)
    coefficients: #(1 2 -5)
    value: 0
```

Figure 5 shows an excerpt of the hierarchy of constraints in BACKTALK:

```
BTConstraint ('isPersistent' 'isStatic' 'owner')
    BTBinaryCt ('x' 'y')
        BTBinaryExtensionCt ('relation')
        BTComparatorCt ()
            BTGreaterOrEqualCt ()
            BTGreaterThanCt ()
        BTEqualCt ()
        BTPerformCt ('expression')
        BTUnaryOperatorCt ()
            BTAbsCt ()
            BTOppositeCt ()
            BTSquareCt ()
    BTGeneralCt ('variables' 'arity')
        BTAllDiffCt ('remainingVars')
        BTBlockCt ('block')
        BTIfThenCt ('ifBlock' 'thenBlock')
        BTLinearCt ('expression' 'constant' ...)
            BTLinearEqualCt ('min' ...)
            BTLinearGreaterOrEqualCt ('max' 'maxList')
        BTNaryLogicalCt ('expression' 'leftVariablesNumber')
            BTAndCt ('trueVariablesNumber')
            BTOrCt ('falseVariablesNumber')
            BTXOrCt ('falseVariablesNumber')
```

Figure 5. Excerpts of the hierarchy of predefined constraint classes in BACKTALK

2.2.2.2 Expressions

BACKTALK provides another means of stating constraints, which is, in some cases, even simpler to use than explicit instantiation messages. The idea is to let the user state constraints using the syntax of standard arithmetical and logical expressions. BACKTALK transform these Smalltalk expressions into constraint creation messages automatically.

This notion of expression is purely syntactical. The idea is only to spare the user the explicit creation of constraints. In practice, this is realized by introducing a language of expressions, which is implemented as messages sent to constrained variables. For instance, messages making up arithmetic expressions (e.g. +, *) are implemented in the class of constrained variables, and yield particular expression objects. These expressions in turn understand these same messages to yield more complex expressions "on the fly". Finally, these expressions indeed behave like variables in the sense that they can be "constrained" using usual arithmetic operators (e.g. =, <, >). These operators in turn are implemented in the expression classes to generated corresponding BACKTALK constraints.

Consider for instance constraint: $x + 2.y - 5.z \geq 0$. Instead of using the following syntax:

```
BTLinearGreaterOrEqualCt
        on: (Array with: x with: x with: z)
        coefficients: #(1 2 -5)
        value: 0
```

it can be stated using the following Smalltalk expression: **`x + (2*y) -(5*z) >= 0.`**

2.2.3 Example

Since problems are stated to be eventually solved, there is no point in differentiating problems and solvers. Problems and solving algorithms are therefore represented in BACKTALK by a single class `BTSolver`. A problem basically consists of a collection of variables and a collection of constraints. The protocol for creating problems is illustrated below on the cryptogram: "send + more = money".

First, a problem is created by instantiating class `BTSolver`. Then variables and constraints are stated. Finally, the problem is initialized by sending message "`pbm variablesToInstantiate: letters`", which specifies that solving this problem amounts at instantiating all the eight letters. The `print:` messages sent to the problem defines how solutions will be printed.

```
sendMoreMoney
    | letters s e n d m o r y send more money pbm |
                        "The problem is created by instantiating class BTSolver"
    pbm := BTSolver new: 'send + more = money'.
                    "Constrained variables creation.  The domains of s and m are
                            restricted to 1..9 as required in problem statement"
    (letters := OrderedCollection new: 8)
        add: (s := V from: 1 to: 9); add: (e := V from: 0 to: 9);
        add: (n := V from: 0 to: 9); add: (d := V from: 0 to: 9);
        add: (m := V from: 1 to: 9); add: (o := V from: 0 to: 9);
        add: (r := V from: 0 to: 9); add: (y := V from: 0 to: 9).
                      "Arithmetic expressions that are to be used to declare the
                                                            actual constraints"
        send := (1000*s + (100*e) + (10*n) + d).
        more := (1000*m + (100*o) + (10*r) + e).
        money := (10000*m + (1000*o) + (100*n) + (10*e) + y).
                                                        "Constraints statement"
        (send + more - money) @= 0.
        BTAllDiffCt on: letters.
                                 "Pattern used to display the eventual solution"
        pbm print: s; print: e; print: n; print: d; print: '+';
            print: m; print: o; print: r; print: e; print: '=';
            print: m; print: o; print: n; print: e; print: y.
                                                        "The resulting problem"
    ^pbm variablesToInstantiate: letters
```

Once created, such a problem can be sent solving messages (e.g. `printFirstSolution`, `printAllSolutions`, `printNextSolution`, `allSolutionsDo: aBlock`) as follows:

```
pbm printFirstSolution
 (send + more = money) 0.002sec ; 1 bt ; 2 choices
 9567+1085=10652

pbm printAllSolutions
 SOL 1: (send + more = money) 0.002sec ; 1 bt ; 2 choices
 9567+1085=10652

 No more solutions.
 (send + more = money) 0.004sec ; 4 bt ; 3 choices
```

Message `printFirstSolution` (resp. `printAllSolutions`) triggers the computation of the first solution (resp. of all solutions). For each solution, information related to the resolution is printed, as well as the solution itself. Information printed includes the name, the resolution time, the number of backtracking (`bt`) and the number of branches developed (`choices`).

2.2.4 Solving Algorithm

As we wrote, a wealth of algorithms has been developed for solving constraint satisfaction problems. Their respective efficiency is highly dependent of the nature of the problem to solve, so none of them can be considered better than the other ones. More generally, as claimed in [6], *"no constraints solver to our knowledge holds all the techniques that we have found necessary to solve [particular problems]"*. This speaks for the design of a general and extensible solving algorithm, which can be augmented, if necessary, with specific mechanisms.

A second remark is that enumeration algorithms are based on the same basic idea: combining a propagation mechanism with a more or less sophisticated backtracking strategy. As argued in Section 1.5, we propose to unify all these algorithms into a single control loop, and use inheritance to adapt the control loop to specific cases, using the "Strategy" [14]. This control loop implements the following general scheme:

General Solving Algorithm (GSA)

1) Choose a non-instantiated variable V and a value x of Dom(V).
2) Instantiate V with x, and save the current state of the problem.
3) Filter constraints to reduce the domains of the remaining problem variables.
4) If there is an empty domain then backtrack to a previous state.
5) If instantiation is complete then
   If a solution is found then stop;
   Else backtrack to a previous state.
6) Goto 1).

In this algorithm, Point 1) is generally undertaken by heuristics. Point 3) is the core of the algorithm. Depending on the actual algorithm considered, Point 3) ranges from "doing nothing" (algorithm BEA described in Section 1.5) to “enforce arc consistency on all constraints”. Point 4) triggers a backtracking to a previous state of the problem. The strategy used for choosing this “previous state” is an important characteristic of the algorithm.

This general algorithm is implemented in class `BTSolver`, and represents the default solving procedure. The main job of this default control loop is to implement an efficient and robust save/restore mechanism, which takes into account the modifications of domains performed by filtering methods. It also handles arbitrary modifications of the problem during the resolution, such as dynamic constraint creation or suppression.

The algorithm is described in Figure 6. We use here the “Template Method” pattern [14]: the algorithm is decomposed into several methods, each of them representing a part of the algorithm. It uses methods `forward` and `backward`, which are redefined in subclasses.

```
firstSolution
“goes forward until a solution is found”
[self solutionFound] whileFalse: [self moveForward].
^self solution

moveForward
“goes forward until a failure occurs, which triggers a backtracking”
[self domainWipedOut] whileFalse: [self forward].
self backward

forward
“saves the state of the problem and chooses an instantiation to enforce”
self saveContext.
self makeAChoice

backward
“chooses a previously saved state of the problem and restores it
| context |
context := self choosePreviousAContext.
self restoreContext: context

makeAChoice
“chooses a variable to instantiate and a value for this variable. Then
performs the instantiation and triggers constraint propagation”
currentVar := self chooseAVar.
currentVal := currentVar nextValue.
self propagateInstantiationOf: currentVar
```

Figure 6. The basic control loop for enumeration algorithms. Parts of this control loop are redefined in concrete subclasses, following the “Template Method” pattern

This solving mechanism, which is close to real full lookahead method, is implemented in class `BTSolver`. Using the “Template Method” and “Strategy” design patterns, one can subclass `BTSolver` to implement other algorithms by redefining some of the five methods above.

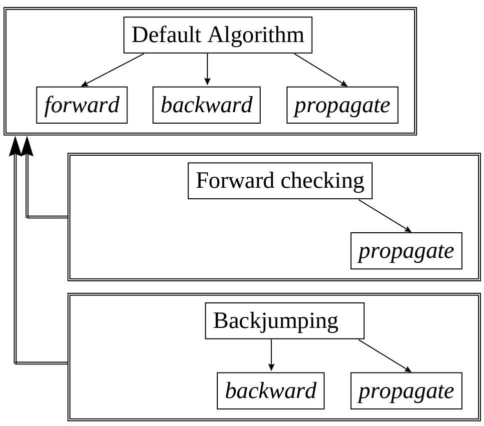


Figure 7 A graphical representation of the library of algorithms, which are represented as classes, following the Strategy pattern. Three classes are represented here: forward checking and backjumping are subclasses of the default algorithm class that implements three key methods, following the Template Method pattern. Subclasses redefine some of these methods, and inherits the others.

2.3 Constraint Propagation

This section aims at presenting the mechanism implemented for constraint filtering in BACKTALK, as defined in Section 1.4. We examine here how we implement constraint filtering in the BACKTALK framework. The main idea is to implement filtering as a set of special methods, called demons, that will be triggered automatically by the solving algorithms, at the right time. This demon mechanism allows to specify filtering methods in a modular and efficient way. We show in the next section that this mechanism can also support the definition of higher-level global constraints.

2.3.1 The Demon Mechanism: Filtering According to Events

The main idea is to decompose the filtering procedure for a given constraint into a set of independent methods that take in charge only a part of the filtering procedure. This allows to define a more efficient propagation mechanism and to define the filtering methods more easily by decomposing it into several elementary methods. The key idea is that the domain of a variable can undergo 1) the suppression of one of its elements, 2) the suppression of several e0lements, 3) and 4) the modification of its lower or upper bound and 5) its reduction to a singleton (i.e. instantiation). We propose to implement the filtering procedure of a constraint with at most five methods, one for each event. Note that a constraint does necessarily respond to all the five events.

Technically, when a constraint C is created, the solver declares a set of demons associated to each variable involved in C. Each demon corresponds to a specific domain reduction event (value, min, max, remove or domain changes). When a variable undergoes a dmoain reduction, the corresponding demon will trigger the execution of the corresponding filtering method implemented in the class of C.

For instance, constraint "$X > Y$" has to be filtered when the upper (resp. lower) bound of X (resp. Y) is decreased (resp. increased). Therefore, when such a constraint is created, the following methods declares the two corresponding demons:

```
postDemons
    "Defines the demons constraint 'self' has to respond to.  In this case,
    self is filtered when the maximum bound of x is changes (i.e. decreased)
    and when the minimum bound of y increases"

    x addMaxDemonOn: self.
    y addMinDemonOn: self
```

the corresponding filtering methods follow:

```
max: v
    "Here v is necessarily x since no maxDemon is defined for y."
    y max: v max; remove: v max

min: v
    "Here v is necessarily y because no minDemon is defined for x"
    x min: v min; remove: v min
```

As said in Section 1.4, to ensure the correctness of the solver, constraint filtering has at least to test that a constraint is satisfied once all its variables are instantiated. This is done in BACKTALK by a method called `minimalFiltering`, which is automatically executed when a variable is instantiated to check that no constraint is violated, otherwise, a failure is raised that provokes a backtracking.

2.3.2 High-Level Constraint Definition

The demon mechanism described above can support the definition of high-level constraints. This is a typical use of BACKTALK as a black-box framework, following the terminology of [18], that is by composition of existing components, instead of inheritance.

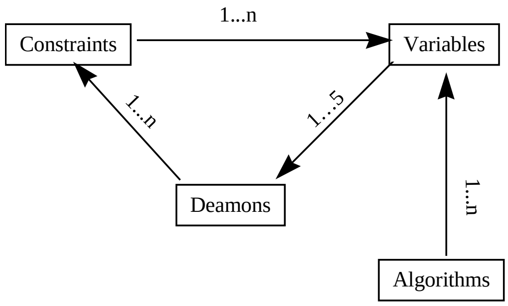


Figure 8 A diagram of the demon mechanism. The algorithm performs reduction messages to its variables. The variables propagate these reductions as events to its demons (a variables can have between 1 and 5 demons, corresponding to its instantiation, the modification of its bounds, the suppression of a value and an arbitrary modification of its domain). Then, the demons forward the events to constraint associated with it. Depending on the event it receives, the constraint will trigger the execution of one of its filtering methods, which causes new modifications to the domains of its variables.

2.3.2.1 Using Dynamic Constraint Management to Control the Resolution

BACKTALK was designed in such a way that constraints can be dynamically created or removed during the resolution of a problem. When a constraint is removed during the resolution, it is restored when the system backtracks. Thanks to this save/restore mechanism, dynamically created constraints can be used to define high-level constraints without implementing new classes.

Consider a constraint expressing that depending on some condition, a constraint has to be satisfied. For instance, if variable `x=1` then variables `y` and `z` should be equals, otherwise they should be different. To implement such a constraint class one need to write filtering methods as shown in the previous section, which can be discouraging. Instead, BACKTALK allows to "compose" the existing filtering methods for each of the constraints appearing in the conditional statement automatically.

Conditional constraints are instances of class `BTIfThenElseCt`, which composes existing constraints. An instance of `BTIfThenElseCt` holds three blocks representing the components of a conditional statement: `ifBlock`, `thenBlock` and `elseBlock`. For example:

```
x := V from: 1 to: 2. y := V from: 1 to: 2. z := V from: 1 to: 2.
BTIfThenElseCt on: x
    if: [x value = 1]
    then: [y @= z] else: [BTAllDiffCt on: y and: z].
```
```
                           SOL 1: x = 1; y = 1; z = 1
                           SOL 2: x = 1; y = 2; z = 2
                           SOL 3: x = 2; y = 1; z = 2
                           SOL 4: x = 2; y = 2; z = 1
```

We give below the filtering method `value`: for conditional constraint. The evaluation of the `ifBlock`'s yields either true, false or nil, and triggers the evaluation of `thenBlock`, the `elseBlock` or nothing.

```
value: aVar
    ifBlock value == nil ifTrue: [^self]."nothing to do"
    self remove.    "restored after backtrackings"
    Condition   ifTrue: [thenBlock value]
                ifFalse: [elseBlock value]
```

2.3.2.2 Disjunctive Constraints

A similar example of using BACKTALK as a black-box framework is given by the definition of *disjunctive* constraints. Disjunctive constraints are often needed, for instance in scheduling problems, to specify that a resource must be used by at most one process at a time. BACKTALK provides a general mechanism for creating disjunctive constraints from existing ones, which uses a class called `BTDisjunctiveCt`:

```
 "The constraint states that tasks t1 and t2 are not simultaneous"
BTDisjunctiveCt  either: [t1 precedes: t2] or: [t2 precedes t1]
```

2.4 The Resulting Framework

To summarize, the BACKTALK framework is essentially made up of four distinct class hierarchies, which entertain complex relationships. As far as the user is concerned, defining a constraint problem amounts to the definition of a small number of subclasses, and, for each of them, a limited number of methods. The difficult parts of constraint satisfaction processes are reused, mainly by inheritance. In this respect, BACKTALK is a white-box framework [18]. The only composition mechanism offered by BACKTALK is the ability of defining higher-level constraints by composition, as seen in the previous section.

Efficiency is an important issue, and the design of BACKTALK was strongly influenced by efficiency objectives. Figure 9 shows the performance of BACKTALK on several well-known combinatorial problems, and compares it with the ILOGSOLVER system.

The difference between the performance of BACKTALK and ILOGSOLVER, renowned for its efficiency, is constant (BACKTALK is 15 times slower). This means that the complexities of are the same. This difference of efficiency is due to the host language, Smalltalk, which is known to be slower than C++. On problems involving objects (the 3 last lines of the table), BACKTALK is more efficient. This is due the way is can be used to state and solve such problems [24].

Problem Instance	BACKTALK	ILOGSOLVER
send + more = money	0.016	0.01
donald + gerald = robert	0.252	0.4
8 queens	0.021	0.01
40 queens	0.181	0.1
100 queens	1.813	0.5
magic square 4x4	0.129	0.01
magic square 5x5	7.490	0.5
magic square 6x6	821.045	50.0
Automatic Harmonization*	BACKTALK	ILOGSOLVER
harmony: 12-note melody	1 sec.	180 sec.
Harmony: 16-note melody	1.5 sec.	240 sec.

*Figure 9. Resolution times of BACKTALK on well-known numeric problems, compared to ILOGSOLVER, one of the most efficient commercial solvers, written in C++. * The automatic harmonization problem is described and the results presented here are explained in Section 3.3.*

We will now examine the other side of BACKTALK: what are the benefits gained by combining objects and constraints for defining and solving constraint satisfaction problems?

3. BACKTALK: Objects for Stating Constraint Problems

In the previous section, we considered object-orientation as a means of designing and implementing a constraint satisfaction solver. In this section, we address the opposite issue, that is: how can our framework be used for stating and solving object-oriented constraint satisfaction problems? We show that there are two radically different ways to constrain objects. We also examine how to mix, as much as possible, the natural mechanisms of object-oriented programming with constraints.

3.1 Constraining Objects: Two Approaches

As already written, constraint satisfaction is based on the notion of constrained variable, which represents unknown values to be computed by the solver. In the previous sections, these values were implicitly considered as atomic (i.e. numerical) values. When solving problems, whose unknown values are complex structures, it becomes necessary to represent the concept of *unknown objects*. The purpose of this section is therefore to answer the following question: "*What happens when unknown values become objects?*"

Two different approaches can be used to address this issue: whether constraints are put *inside* objects or *outside* objects. The choice of one of these approaches is of utmost importance as for reusability, easiness and efficiency as well. The following sections introduce these two approaches.

3.1.1 Constraints within Objects: Constraining Partially Instantiated Objects

A natural way to constrain objects is to consider attributes as constrained variables, thus leading to the notion of *partially instantiated objects*. A partially instantiated object is an object whose attributes are constrained variables instead of being *fully-fledged objects*. This approach, called *attribute-based*, corresponds to the following idea: *constraining an object amounts at expressing a property holding on its attributes*.

Figure 10 illustrates the attribute-based approach. Circled question marks represent constrained variables, which are indeed the instance variables of the objects, and square boxes represent partially instantiated objects. Simple arrows are internal constraints, also called structural constraints, which express properties of the objects. The double arrow represents an external constraint:

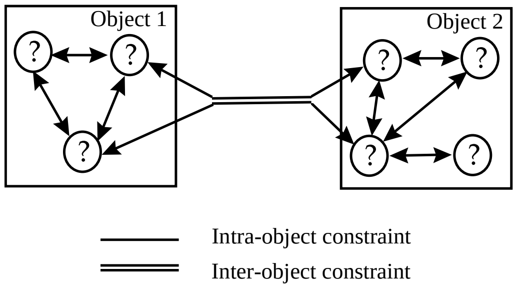


Figure 10. Graphical illustration of the attribute-based approach for constraining objects. Instance variables are considered as constrained variables. Constraints are therefore "inside" objects.

3.1.2 Constraints outside Objects: Constraining Fully-Fledged Objects

The orthogonal approach consists in putting the constraints outside objects that need to be constrained. This approach, henceforward called *class-based*, aims at handling *fully-fledged* instead of *partially instantiated* objects. The idea is to use *classes* as natural *domains* for constrained variables by putting objects in the domains. Figure 11 illustrates the class-based approach. Question marks represent the variables, and boxes represent fully-fledged objects. In this case, objects are in the domains of the constrained variables.

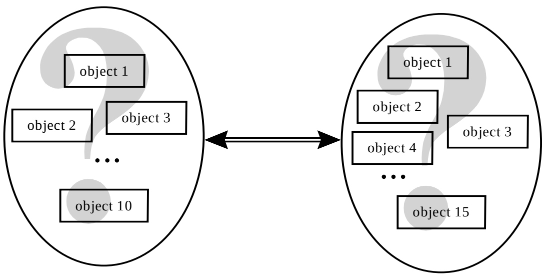


Figure 11. Graphical illustration of the class-based approach for constraining objects. Objects are in the domains of constrained variables. Constraints are therefore "outside" objects.

3.1.3 Illustrating the Two Approaches

Given a set of rectangles with their sides parallel to the axis, consider the problem of finding two squares that do not overlap each other. We assume that a rectangle is determined by the coordinates of its upper-left and lower-right vertices (i.e. `left`, `right`, `top` and `bottom`). Let us express this problem using both attribute-based and class-based approaches.

Using attribute-based approach, one would define partially instantiated rectangles, say `R1` and `R2`, whose attributes are the constrained variables of the problem. In this case, the constraint "*being square*" and the non-overlapping constraint would be defined as arithmetical relations between these constrained attributes. The problem statement would look as follows:

```
"R1 is a square"
(R1 top) - (R1 bottom) = (R1 right) - (R1 left)

"R2 is a square"
(R2 top) - (R2 bottom) = (R2 right) - (R2 left)

"R1 and R2 do not overlap"
(R1 right < R2 left)
OR (R1 top > R2 bottom)
OR (R1 left > R2 right)
OR (R1 top < R2 bottom)
```

Defining this problem following the class-based approach would lead to state variables `R1` and `R2`, whose domains contain fully-fledged rectangles, and to define constraints directly between these variables. The following problem statement illustrates this point:

```
"R1 is a square"
R1 isASquare

"R2 is a square"
R2 isASquare

"R1 and R2 do not overlap"
R1 doNotOverlap: R2
```

Where it is to be understood that `isASquare` (resp. `doNotOverlap:`) messages state constraints holding on the `R1` (resp. `R1` and `R2`) constrained variable(s).

3.1.4 Comparison of the Two Approaches

Intuitively, the attribute-based approach answers the question: "What happens when unknown values become *partially instantiated* objects?" while the class-based approach answers the question: "What happens when unknown values become *fully-fledged* objects?". The two problem statements above illustrate the fundamental differences between them, which are concerned with predefined class reuse, constraint definition, problem structure and efficiency as well.

As for class reuse, when using the attribute-based definition, one has to deal with partially instantiated rectangles that cannot be instances of predefined Smalltalk classes. In other words, the attribute-based forces to define *ad hoc* classes. Conversely, the class-based approach allows predefined classes to be reused as is. The idea is that the class-based approach does not depend on class implementation (encapsulation is not jeopardized by class-based statements while it is by attribute-based).

As for constraint definition, in the attribute-based case, arithmetical constraints are used to state the problem. Conversely, the class-based definition uses constraints directly holding on the rectangle constrained variables. The question that arises here is how to define these "class-based" constraints? We will address this issue in Section 3.2.

The structures of the two problems are radically different both in terms of variables and constraints. The attribute-based problem is a numerical CSP, whose objects are mere collections of variables while fully-fledged object structures are used in the class-based problem as values for variables. This leads to dramatic efficiency differences (see Section 3.3 and [28]).

3.2 Defining Class-Based Constraints

Recall that the central question here is "*What happens when unknown values become fully-fledged objects?*" In the previous section we wrote that using the class-based approach for stating problems involving objects leads to the statement of constraints holding directly on object variables (i.e. constrained variables whose domain contains fully-fledged objects).

Numerical problems involve constraints that are generally combinations of basic operators (e.g. +, -, *, /, =, ≠, >, <, ≤, ≥, X^Y). Of course, other constraints have to be considered for specific problems, such as graph theory constraints (e.g. cycle constraint in CHIP) or set theory constraints (e.g. cardinality and distribution constraints [27]). However, these constraints have generally a well-known semantics and are in limited number.

On the contrary, when stating problems involving objects using the class-based approach, as illustrated in the previous section, we need to define constraints expressing arbitrary properties of objects. The general idea is that any Boolean expression involving objects is likely to be used as a constraint. For instance, in Section 3.1.3, one can assume that class Rectangle implements a method, say `isAsquare`, testing if a rectangle is a square. In this case, our purpose is to use this method in order to create a constraint. BACKTALK provides two different means of stating such constraints.

3.2.1 A General Mechanism: Block Constraints

The simpler, and the most general way to use directly methods to state constraints is to use *block constraints*, which allow to define constraints by way of any Smalltalk `BlockClosure` whose evaluation yields a Boolean.

For instance, consider two constrained variables, `R1` and `R2`, whose domain contains instances of class `Rectangle`. A constraint requiring the two rectangles do not overlap can be defined as follows, as far as method `doNotOverlap:` is defined in class `Rectangle`:

```
BTBlockCt
    on: R1 and: R2
    block: [:a :b | a doNotOverlap: b]
```

The block has one argument for each variable involved in the constraint. These arguments are not the constrained variables, but rather possible values for these variables. Indeed, during the resolution of the problem, these arguments will be assigned values of the domain of the corresponding variable. In other words, the block can use the language of the values in the domains (i.e. the methods defined in their class). For instance, in the block constraint defined above, argument `a` (resp. `b`) is assigned a value picked up in the domain of `R1` (resp. `R2`) when the block is evaluated.

To give another example, consider two variables, `C1` and `C2`, whose domain contains Smalltalk classes. The following block constraint forces the value of variable `C2` to be a subclass of the value of `C1`:

```
BTBlockCt
    on: C1 and: C2
    block: [:a :b | b inheritsFrom: a]
```

However, block constraints are useful for many problems because they allow complex relations to be stated as constraints. For instance, we used such constraints to implement a simple application that finds design patterns in a standard Smalltalk image, which is not reported here for space limitations.

Block constraints are a general means of defining arbitrary constraints on arbitrary objects. Of course, because of this generality, the filtering method of implemented in class `BTBlockCt` is not efficient; it consists in enforcing arc consistency by computing, in the worst case, the Cartesian product of the domains of the variable, according to the definition of given in [20].

Block constraints are particularly well adapted to prototyping applications rapidly, although they are often replaced by user-defined constraints in the final application for efficiency reasons.

3.2.2 An Efficient Mechanism: Perform Constraints

Block constraints, introduced above, are the most general type of "class-based" constraints provided in BACKTALK. These constraints cannot be efficiently implemented in their full generality, because nothing is known about the semantics of the corresponding block. However, there exist families of "class-based" constraints for which an efficient filtering method can be provided. One of them is the family of constraints defined by a single Smalltalk message. This is the purpose of *perform constraints*.

Perform constraints are used to specify that there is between two variable X and Y a relation defined by a method selector m, i.e. the value Y is the image of the value X by the method named m. Notice that a perform constraint C holding on variables X and Y and whose message is m can be stated as the following block constraint:

```
BTBlockCt on: X and: Y block: [:a :b | (a perform: m) = b]
```

A perform constraint is made up with two constrained variables, say X and Y, and an arbitrary Smalltalk selector (possibly with arguments), and it means that variable Y is deduced from variable X by applying it the associated selector. Consider for instance a variable X whose domain contains Smalltalk rectangles and a variable Y whose domain contains integer numbers. Stating a perform constraints between *X* and *Y* associated to selector `area` will ensure that the value of *Y* (a number) is the area of the value of *X* (a rectangle).

The following BACKTALK session illustrates the creation of perform constraints, and how consistency is maintained between variables linked by a perform constraint. When variable `y` is assigned value `false`, the perform constraint is used right-away to remove every vowel from the domain of `x`. Conversely, removing all the vowels of the domain of variable `y` would have caused variable `x` to be assigned value false.

```
x := V label: 'x' domain: #(a b c d e f).          =>['X' (#a #b #c #d #e #f)]
y := (x btPerform: #first) btPerform: #isVowel.    =>['X first isVowel' (t f)]
y value: false.                                    =>['X first isVowel' (f)]
x                                                  =>['X' (#b #c #d #f)]
```

3.2.3 Using Methods for Constraints: Choosing the Right Approach

Block and perform constraints are implemented using second-order abilities of Smalltalk. Their purpose is to put the language of objects at user's disposal. This is a *sine qua non* to allow the statement of constraint satisfaction problems involving fully-fledged objects. Moreover, these particular constraints favor the reusability of predefined class, since their methods can be used straightforwardly to define constraints.

An important thing to remark is that stating constraint problems involving complex objects using the class-based approach, complex constraints can be defined, by way of perform of block constraints, from methods implemented in the corresponding classes. On the contrary, when using the attribute-based approach, constraints expressing relations between complex objects have to be stated in terms of rock-bottom objects (i.e. attributes of complex objects). This implies an overhead in problem's statement, and is very often less efficient, as argued in Section 3.3.

3.3 Stating Problems Involving Objects: A Case Study in Automatic Harmonization

This section reports the design of a system that solves harmony exercises. It illustrates the points discussed in the previous sections because harmony exercises, hereafter referred to as AHP (*Automatic Harmonization Problems*), are particularly representative of "object + constraint" problems. Solving a harmony exercise consists in finding harmonization of a melody, e.g. the melody shown by Figure 12, or, more generally, any *incomplete* musical material, which satisfies the rules of harmony. The standard exercise is to harmonize four voices (see Figure 13 for a solution of the melody below).

Figure 12. An initial melody to harmonize (the beginning of the French national anthem, 18 notes).

The constraints needed to solve the AHP can be found in any decent treatise on harmony [29]. The problem is an interesting benchmark because it involves many complex objects (e.g. chords). Moreover, there are various types of constraints which interact intimately: 1) horizontal constraints on successive *notes* (e.g. "two successive notes should make a consonant interval"), 2) vertical constraints on the *notes* making up a *chord* (e.g. "no interval of augmented fourth, except on the 5th degree" or "voices never cross") and 3) constraints on *sequences of chords* (e.g. "two successive chords have different degrees").

3.3.1 Previous Attempts at Solving Harmony Exercises with Constraints

Harmonization of a given melody naturally involves the use of constraints, because of the way the rules are stated in the textbooks. Indeed, several systems proposed various approaches to solve the AHP using constraints. The pioneer was Ebcioglu [12], who designed a constraint logic programming language (BSL) to solve this problem. His system not only harmonizes melodies (in the style of J.-S. Bach), but is also able to generate chorales from scratch. Although interesting, the architecture is difficult to transpose in our context because constraints are used passively, to reject solutions produced by production rules.

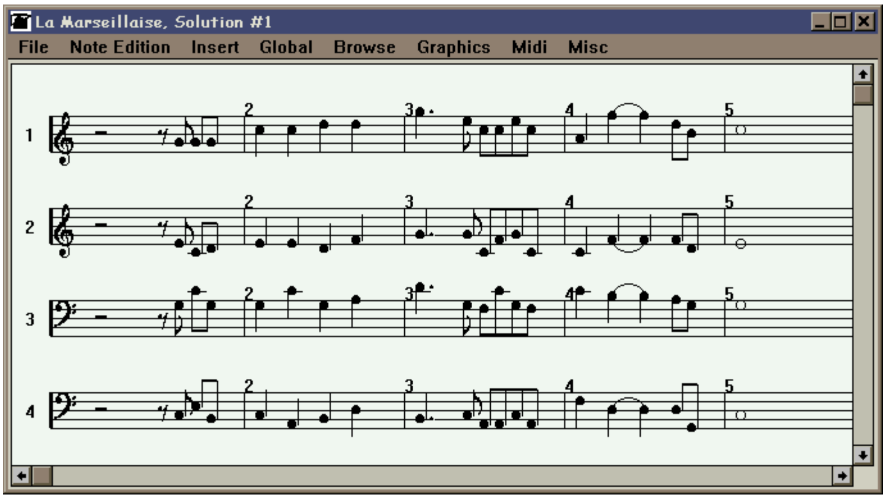


Figure 13. A solution proposed by BACKTALK from the initial melody of Figure 12.

More recently, [30] proposed to solve the AHP using CLP, a constraint extension to Prolog [16]. Their results are poor: more than 1 min. and 70 Mb of memory to harmonize an 11-note melody (see Figure 15). Ovans [22] was the first to introduce the idea of using arc consistency techniques to solve the AHP, but his system is poorly structured and imposes an unnatural bias on the representation of musical entities. The system proposed by Ballesta [2] is much more promising. Ballesta uses IlogSolver to solve the AHP, and uses both objects and constraints.

All these systems are based on a representation of musical structures as atomic variables and structural constraints. In other words, and using our terminology, they follow the attribute-based approach. For instance, in Ballesta's system, 12 attributes are defined to represent one interval. Nine constraints are then introduced to state the relations that hold between the various attributes.

As a result, constraints have to be defined using a low-level language, thus requiring a translation of harmonic and melodic properties, given in harmony treatises, in terms of numbers. The constraint representing the rule expressing that relations of parallel fifth are forbidden between two successive chords is given below as it is expressed in Ovan's system.

$$\texttt{parallel-fifth}(c_i,m_i,c_{i+1},m_{i+1}) \Leftrightarrow \neg \texttt{perfect}(c_{i+1},m_{i+1}) \vee (c_i-c_{i+1}).(m_i-m_{i+1}) \leq 0$$

where

$$\texttt{perfect}(c_i,m_i) \Leftrightarrow |c_i-m_i| \in \{7,19\}$$

Figure 14. The constraint corresponding to the parallel fifth rule, expressed in Ovan's system, based on CLP.

Moreover, the attribute-based approach leads to stating a huge amount of constraints and variables. For instance, in Ballesta's system, to state the AHP on a *N*-note melody, $126 \times N - 28$ variables are defined.

3.3.2 Our Approach for Solving Harmony Exercises

The poor performance of existing systems, see Figure 15, led us to experiment a radically different approach. The drawbacks of these systems can be summed up as follows: first, there are too many constraints. The approaches proposed so far do not structure the representation of the domain objects (e.g. intervals, and chords). When such a structure is proposed, as in Ballesta's system, objects are treated as passive clusters of variables. Second, the constraints are treated uniformly, at the same level. This does not reflects the reality: a musician reasons at various levels of abstraction, working first at the note level, and then on the chords. The most important harmonic decisions are actually made at the chord level. This separation could be taken into account to reduce the complexity.

These remarks led us to reconsider the AHP in the light of the class-based approach for object + constraint problems, that is, with a reverse viewpoint from our predecessors. Rather than “starting from the constraints”, and devising object structures that fit well with the constraints, we “start from the objects”, and fit the constraints to them. Indeed, a lot of properties of the domain objects may be more naturally described as methods instead of constraints.

To do so, we reuse an object-oriented library, the MusES system [23], which contains a set of around 90 classes that represents the basic elements of harmony, e.g. notes, intervals, scales and chords. In our application, the domains contain musical objects provided by MusES. The constraints hold directly on these high-level objects, using the methods defined in the corresponding classes. These constraints are instances of block and perform constraints, introduced in Section 3.2.

The main idea here, is to consider high-level objects of the problem, namely chords, as possible values for constrained variables. In other words, domains contain fully-fledged chords, which are instances of MusES class `Chord`. As a consequence of reifying chords as values for variables, the resulting system is very much understandable, because constraints can be stated using directly the language of chords. For instance, the rule forbidding parallel fifth is simply defined by the following BACKTALK expression:

```
BTConstraint
   on: chord and: nextChord
   block: [:c1 :c2 | c1 (hasParallelFifthWith: c2) not]
```

Moreover, the problem to solve is much smaller, since for a *N*-note melody, only *5×N* constrained variables are created (to be compared to the *126×N - 28* variables in Ballesta's system). This results in improving efficiency in a dramatic way, as shown in Figure 15, which gives the performance of our application compared with the systems of Ballesta and Tsang. Ovan's work is not reported here since it addresses a simpler problem, namely two-voice harmony exercises instead of four voice.

	11 notes	12 notes	16 notes
Tsang (CLP)	60 sec.	?	?
Ballesta (ILOGSOLVER)	?	180 sec.	240 sec.
BACKTALK + MusES	1 sec.	1 sec.	1.5 sec.

Figure 15. Comparing the performance of our solution with previous approaches using CSP to solve the AHP.

4. A Complete Application: Crossword Puzzle Generation

As shown in crossword puzzle generation is a highly combinatorial problem that can be solved by procedural approaches [15, 17, 21]. Addressing this problem with a declarative approach leads generally to inefficient systems [4, 31]. The reason is that a standard puzzle contains about 30 words slots, leading to a huge search space of about 10^{100} combinations if the dictionary contains approximately 150,000 words.

The problem consists in finding a crossword, given an initial empty puzzle with open and closed cells and a list of words. Standard CSP techniques are not able to cope with this problem. Here, we show how domain-specific knowledge can be expressed using the full range of features of BACKTALK to solve this problem. This knowledge is three-fold: topologic knowledge, lexical knowledge, and knowledge on letter distribution.

4.1 Choosing the Right Algorithm

The crossword problem is a typical example of a “weakly constrained” problem. Intuitively, the idea is that instantiating a variable with a given word will have a limited impact on the variables not directly crossing it. This is explained by the fact that the distribution of letters in words is quite uniform, except for special letters (such as “q”, see 4.3). Therefore we chose the forward-checking algorithm. Exceptional cases due to non-uniform distribution of letters are examined in Section 4.3.

4.2 A Filtering Method for Intersection Constraints

The crossword problem, in its basic form, contains only intersection constraints. Knowledge on intersection can be used to devise an efficient filtering method for these constraints. The method follows:

```
filter intersection between X and Y:
• (i, j) := intersection of X and Y.
• Compute possibleLetters (X,i) = the set of possible letters at position i for X.
• Remove from domain(Y) words that don't contain one of possibleLetters (X,i) at j.
• Compute possibleLetters (Y,j) = the set of possible letters at position j for Y.
• If possibleLetters(Y,j) ⊄ possibleLetters (X,i) then remove from domain(X) words that
don't contain one of possibleLetters (Y,j) at position i.
```

It is easy to show that this procedure achieves full arc consistency for the intersection constraint. The complexity is linear, to be compared to the quadratic complexity of the default filtering method!

4.3 Exploiting Knowledge on Letter Distribution

A particular knowledge is that all letters are not distributed uniformly in words. A typical example is that letter "q" is almost always followed by letter "u" (at least, in English and French). There are numerous examples of this kind of rule, such as: "no word starts by the same consonant twice" or "j is never repeated twice", "letters are rarely repeated three times", and so forth. These regularities are not always true, but only give strong indications on letters not yet found.

It is possible to express this piece of knowledge in terms of constraints and variables, by considering a *virtual constraint* between two parallel words `v` and `v'`, only when certain conditions are satisfied (here, letter "q" appears). Of course, it would be awkward to actually add dynamically this virtual constraint to the problem, because this virtual constraint is already represented by constraint `u` between `v'` and `w`.

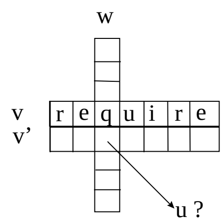


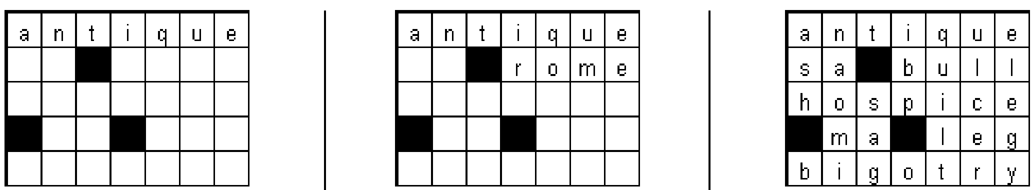
Figure 16. The letter "q" implicitly creates a relation between `v` *and* `v'`*, which corresponds to the intersection between* `v'` *and* `w`*.*

A way to implement this "virtual constraint" is to define a conditional constraint, called `LetterDependencyCt`. The statement of this constraint consists in 1) a condition on values of variables, and 2) a set of intersection constraints to filter when the condition is satisfied:

```
BTIfThenCt
  on: wordVariableList
  if: [there exists a variable v, whose value contains a "q"]
  then: [filter intersection between v', parallel to v, and w (perpendicular to v at
position of letter "q")]
```

4.4 Results

We conducted a series of experiments on crosswords, with and without these three kinds of knowledge. Figure 17 illustrates the effect of exploiting knowledge on letter distribution.



A. Word 'antique' is fixed before the resolution.

B. Without rule 'q' → 'u', word 'rome' is tried, which leads to the development of a useless search tree.

C. With rule 'q' → 'u', the variable parallel to 'antique' is only instantiated with words having a 'u' in 2nd position.

Figure 17. The resolution of a crossword.

These experiments show clearly that our approach allows to reduce the domains of word variables, thereby reducing the number of backtracks. The following table gives execution times and number of failures when combining the various knowledge representations described in this section. Expectedly, the best strategy is achieved when combining all three types of knowledge with forward checking.

Specialized filtering	First fail heuristic	Knowing 'q' → 'u'	CPU (in sec)	fails
NO	NO	NO	> 3,600	> 5,000
YES	NO	NO	545	10,546
YES	NO	YES	166	3,396
YES	YES	NO	23	268
YES	YES	YES	6	36

5. Summary

The framework paradigm offers a smooth and efficient integration of CSP with objects. One way to assess the relevance of this approach, as opposed to the language-based approach, is to compare it with two extreme cases: CHIP and CLAIRE. The main difference with CHIP is that since BACKTALK provides the relevant concepts of CSP as classes, it allows to redefine them by inheritance, thus gaining flexibility. The difference with CLAIRE is that BACKTALK imposes the main control-loop, whereas CLAIRE leaves it to the responsibility of the user: since CLAIRE has all the abilities of a complete hybrid language, it is suitable for highly specific applications. Table 1 illustrates the position of the framework approach.

Approach	Main characteristics	Examples
Library	Parameterized high-level constraints	CHIP
Framework	Control-loop, simple constraints	BACKTALK, ILOGSOLVER
Language	Low-level language constructs	CLAIRE

Table 1 The three approaches in proposing CSP mechanisms to a user

The framework approach is claimed more comfortable for standard applications because it provides relevant predefined abstractions. By hiding from the user the difficult mechanisms of CSP techniques, while allowing him to redefine parts of it, BACKTALK achieves a desirable feature of frameworks, that is a good compromise between efficiency and complexity. This echoes Steve Jobs' opinion concerning interface builder frameworks: "Simple things should be simple, complex things should be possible."

References

[1] P.Avesani, A.Perini, and F. Ricci, COOL: An Object System with Constraints, TOOLS'2, pp. 221-228, 1990

[2] P.Ballesta, Contraintes et objets: clefs de voûte d'un outil d'aide à la composition ?, Recherches et applications en informatique musicale, Ed. Hermès, Paris, 1998

[3] N.Beldiceanu and E. Contejean, Introducing global constraints in CHIP, Journal of Mathematical and Computer Modelling, 20 (12), pp. 97-123, 1994

[4] H.Berghel and C.Yi, Crossword Compiler-Compilation, The computer Journal, 32(3), pp. 276-280, Jun. 1989

[5] C. Bessière, Arc consistency and Arc consistency Again, Artificial Intelligence, vol. 65(1), pp. 179-190, 1994

[6] Y. Caseau, Abstract Interpretation of Constraints over an Order-Sorted Domain, International Logic Programming Symposium, Sand Diego (Ca), pp. 435-454, 1991

[7] Y. Caseau, Constraint Satisfaction with an Object-Oriented Knowledge Representation Language, Journal of Applied Intelligence, vol. 4, pp. 157-184, 1994

[8] Y.Caseau and F.Laburthe, Introduction to the CLAIRE Programming Language, LIENS, Paris, TR, 1996

[9] P.Codognet and D.Diaz, Compiling Constraints in clp(FD), Journal of Logic Programming, 27(3), pp. 185-226, 1996

[10] A. Colmerauer, An Introduction to Prolog-III Communication of the ACM, vol. 33(7), pp. 69, 1990

[11] R. Dechter and I. Meiri, Experimental Evaluation of Preprocessing Algorithms for Constraint Satisfaction Problems Artificial Intelligence, vol. 68, pp. 211-241, 1994

[12] K. Ebcioglu, An Expert System for Harmonizing Chorales in the Style of J.-S. Bach in Understanding Music with AI: Perspectives on Music Cognition, K. E. O. L. In M. Balaban, Ed., AAAI Press ed. California, 1992

[13] M. Fayad and D. Schmidt, Object-Oriented Application Frameworks, Communications of the ACM, Special Issue on Object-Oriented Application Frameworks, 40 (10), October 1997

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns, Addison-Wesley, 1994

[15] M.L. Ginsberg, M. Frank, M.P. Halpin and M.C. Torrance, Search Lessons Learned from Crossword Puzzles, 8th National Conference on AI, 210-215, Boston, Massachusetts, 1990

[16] J. Jaffar and J. L. Lassez, Constraint Logic Programming, IEEE 4th International Conference on Logic Programming, Melbourne, 25-29 may 1987

[17] S.C. Jensen, Design and Implementation of Crossword Compilation Programs Using Serial Approaches. Master's Thesis. Dept. of Mathematics and Computer Science, Odense Univ, 1997

[18] R. Johnson and B. Foote, Designing Reusable Classes JOOP, vol. 1(2), pp. 22, 1988

[19] J.-L. Laurière, A Language and a Program for Stating and Solving Combinatorial Problems Artificial intelligence, vol. 10(1), pp. 29-127, 1978

[20] A.K. Mackworth, Consistency in Networks of Relations Artificial intelligence, 8(1), pp. 99-118, 1977

[21] L.J. Mazlack, Computer Construction of Crossword Puzzles using Precedence Relationships, Artificial Intelligence 7(1):1-19, 1976

[22] R. Ovans, An Interactive Constraint-Based Expert Assistant for Music Composition, the Ninth Canadian Conference on Artificial Intelligence, University of British Columbia, Vancouver (Can), 1992

[23] F. Pachet, The MusES System: an Environment for Experimenting with Knowledge Representation Techniques in Tonal Harmony, First Brazilian Symposium on Computer Music - SBC&M, Caxambu, Minas Gerais (Brazil), pp. 195-201, August 1994

[24] F. Pachet and P. Roy, Integrating Constraint Satisfaction Techniques with Complex Object Structures, 15th Annual Conference of the BCS Specialist Group on Expert Systems, Cambridge, pp. 11-22, Dec. 1995

[25] P. Prosser, Hybrid Algorithms for the Constraint Satisfaction Problem Computational Intelligence, vol. 9(1993), pp. 268-299, 1993

[26] J.-F. Puget and Michel Leconte, Beyond the Glass Box: Constraints as Objects, International Logic Programming Symposium, ILPS, pp. 513-527, Portland, Oregon, Dec. 1995

[27] J.-C. Régin, Generalized Arc Consistency for Global Cardinality Constraint, Thirteenth National Conference on Artificial Intelligence, Portland, Oregon, 1996

[28] P. Roy and F. Pachet, Reifying Constraint Satisfaction in Smalltalk, Journal of Object-Oriented Programming, 10 (4), pp. 51-63, 1997

[29] A. Schoenberg. (1978) Theory of Harmony, University of California Press, Berkeley

[30] C. P. Tsang and M. Aitken, Harmonizing Music as a Discipline of Constraint Logic Programming, ICMC, Montréal (Can), pp. 61-64, 1991

[31] J.M. Wilson, Crossword Compilation Using Integer Programming, The computer Journal, 32(3), pp. 273-275, Jun. 1989