Reifying Constraint Satisfaction in Smalltalk

Pierre Roy, François Pachet

Laforia-IBP, Université Paris 6, Boîte 169
4, place Jussieu,
75252 Paris Cedex, France.
Tel: (33) 1 44277004
Fax: (33) 1 44277000
E-mail: {roy|pachet}@laforia.ibp.fr

Abstract

We discuss the problem of the representation of constraints in an object-oriented programming language. We focus on a particular topic - constraint satisfaction for finite domains - and propose a class library that integrates constraints within an object-oriented language. The library is based on the systematic reification of variables, constraints, problems and algorithms. The library is implemented in Smalltalk, and is used to state and solve efficiently complex constraint problems involving Smalltalk structures.

1. Introduction

Constraint satisfaction programming (hereafter referred to as CSP) is a powerful paradigm for solving complex combinatorial problems, which has gained attention in recent years, particularly with work on constraint logic programming. The notion of constraint was initially seen as an algorithmic problem, e.g. by [Mackworth 77] who sees constraint graphs as networks of relations for finite domains, or in the Alice system [Laurière 78] for integrating finite domain constraints with optimization. Other techniques for non finite domains have been extensively studied, such as simplex-based algorithms for linear problems, or simulated annealing for optimization in non linear problems.

Constraint languages were introduced by logic programming

The first attempts at integrating these algorithms in a programming language have been made by the community of logic programming. Constraint logic programming was primarily designed as an extension of logic programming to deal with arithmetic computation and with specific computation domains like integer, real or rational numbers and booleans. Its best known representatives are Prolog III [Colmerauer 1990], CHIP [Van Hentenryck 89], CLP (R) [Jaffar & Lassez 87]. Logic programming heavily relies on the notion of "logical variable", representing some unknown, *untyped* value, to be computed by the system. CLP extended it

towards the notion of *constrained variable*, which may be seen as an intentional representation of an unknown *typed* value, satisfying some set of conditions. More precisely, the type of a constrained variable is defined by the set of its possible values, also called its *domain*. Depending on the nature of the domain (finite, infinite, boolean, sets, intervals, etc.), different techniques are used to compute the values of variables, with a semantics directly inherited from logic programming, i.e. based on unification, resolution and backtracking.

Constraints and objects

On the other hand, object-oriented programming and representation languages have become commonplace in software engineering and knowledge representation. The integration of constraint mechanisms in object-oriented languages introduces a radically different way of seeing constraints, by putting the emphasis on *domains*, rather than on algorithms. Indeed, classes in the sense of OOP may be seen as intentional representations of domains. Class instances then become possible *values* for constrained variables. Furthermore, the interface of the classes provides a *language* in which the constraints can be specified.

Many constraint satisfaction systems have been integrated in object-oriented languages. There are two main notions of constraints prevalent in the community of object-oriented programming: constraints in a "propagation" sense, historically the first to be proposed, and more recently, constraints in a "solving" sense.

Propagation

The first integrations of constraint mechanisms in a object-oriented language was pioneered by works of Alan Borning, with the ThingLab system [Borning 81]. These works embody the so-called "propagation" view (also called the *perturbation model*) of constraints. In this approach, constraints are declarative statements of relations that must be always true. In practice, they are a means of specifying a "stable" state of a system, specified as a set of constrained variables. The main job of the constraint interpreter is to restore the state of the system after a perturbation, typically when one object changes state, by changing the state of other objects so that the constraints still hold. This notion of constraint is particularly well suited to the specification of graphical interfaces. For instance, the relation between several objects within a window may be stated in terms of constraints. When an object is moved, the system enforces the constraints, i.e. computes the position of other objects so that constraints are still satisfied. The typical example is the Farenheit-to-Celsius converter, in which the user may change graphically the value of a slider representing a temperature in Celsius degrees, and the system will change accordingly the value of another slider expressing the temperature in Farenheit. The works on ThingLab paved the way for exploring the integration of constraint propagation with object-

oriented programming, and was a source of inspiration for other systems, such as *Kaleidoscope* [Freeman-Benson 90], [Freeman-Benson & Borning 92].

Satisfaction

The other view on constraints is the problem solving view, also called *constraint satisfaction*, or *refinement model.* In this context, constraints are seen as a declarative means of specifying a *problem* to be solved. Here, the idea is not to restore the state of a system after a perturbation, but rather to find one solution (or all possible solutions) of a given problem. If the notion of variable still holds, variables are not initialized as in the perturbation model; they are initially unbound. Typical constraint satisfaction problems include: allocation problems (e.g. human resource management), scheduling problems (e.g. scheduling vehicles on an a assembly line, scheduling trains, boats or planes), planning (e.g. planning human resources), optimization of resources under constraints (train container optimization, minimization of garbage when cutting wood planes or leather), etc. A particularly interesting branch of CSP is *finite domain* CSP; widely used to design and solve combinatorial problems, when domains of variables are finite sets of values, such as intervals of integers. A wealth of algorithms have been developed to compute efficiently the list of all possible solutions. Developing fast algorithms for finite domain CSP is still today a very active area of research.

Proposals to integrate constraint satisfaction techniques in object-oriented languages have been made recently, specifically with finite domain CSP (FD-CSP). *IlogSolver* is a library of routines that provides the C++ language with facilities for constraint logic programming [Puget 1994]. FD-CSP mechanisms have been integrated in other languages: the KEE system in the COOL language [Avesani & al 90]; the LAURE system introduces efficient CSP in a proprietary object-oriented language [Caseau 94].

In this paper, we propose a design and an implementation of an integrated finite domain constraint satisfaction solver in Smalltalk. Our position is that the choice of the underlying language is of great importance, since we are interested in reusing existing class libraries and frameworks. The choice of Smalltalk as the autochtonous language is therefore justified by its widespread use and the quality of its class libraries. The only available extension of Smalltalk that provides constraint facilities is the OTI constraint module [Borning & Freeman-Benson 95]. This library implements several constraint propagation algorithms (the *ultra-violet* family), and is used mainly for building graphical interfaces. No proposal has been made, to our knowledge, to extend Smalltalk with finite domain constraint satisfaction mechanisms.

Our integration is based on the systematic exploitation of the vision of classes as potential *domains* for constrained variables. The design involves a reification of the important notions of

finite domain CSP: variables, constraints, problems and solving algorithms. The resulting system, called BackTalk, allows the definition of complex CSP problems involving fully-fledged Smalltalk objects, and arbitrary Boolean expressions, and provides efficient algorithms to solve these problems. Moreover, object-orientation allows us to propose a constraint solver with all the good properties of object-oriented software. The solver can be used as a mere class library, without a priori knowledge of finite domain CSP, or used as a *framework* for specialists who want to design and implement their own constraint satisfaction algorithms.

The paper is structured as follows. In section 2 we define more precisely finite domain CSP; in section 3 we introduce our system, called BackTalk, a canonical integration of FD-CSP with Smalltalk. In section 4, we outline the main design of a complete application using BackTalk. In section 5, we discuss the main characteristics of the BackTalk system.

2. Finite domain CSP

Intuitively, formulating a problem in terms of constraints amounts to characterizing *a priori* a solution of this problem: the *constraints* of the problem are the properties that a solution should verify. More precisely, a constraint satisfaction problem (CSP) is defined by 1) a finite set of *variables*, each one taking values in a finite set (its *domain*), and 2) a finite set of *constraints* on these variables, usually expressed as conditions on the variables. A *solution* of a CSP is an assignment of values to variables satisfying simultaneously all the constraints.

Finding a solution to a CSP, in general, is a NP-complete problem. In practice, several algorithms have been developed to solve efficiently most CSPs. Algorithms are usually classified into two main families: the *enumeration* algorithms and the *filtering* algorithms. We will briefly review these two families in the next section.

2.1. An illustrative example

To illustrate the notion of constraint in our context, let us take as an example the following crossword problem. Find a crossword without holes, such that words read the same horizontally and vertically. The initial input of the problem is a list of English words containing the most frequently used words (see e.g. the file `/usr/dict/words` on Unix stations, which contains about 25000 words of lengths 2 to 22).

In BackTalk, this problem is stated and solved as follows:

- There are *n* variables $h_1\ h_2\ h_3\ ..\ h_n$, each with domain the list of words of size `n` (around 5000 for values of *n* between 4 and 7).

- There are *(n.(n-1)/2)* constraints, stating that the *ith* letter of the *jth* variable is equal to the *jth* letter of the *ith* variable.

Figure 1 shows one solution for this problem when n = 6.

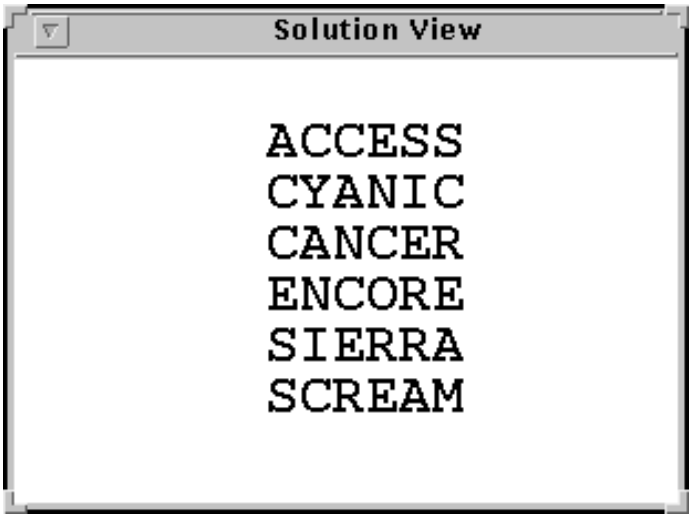

Figure 1. A view on a solution for the cross-word example when *n=6*.

Note that this problem is a hard one, if one is to consider all the possible combinations of the values of variables. With a list of 5000 words of size 6 for instance, a naive computation of all possible solutions yields 5000^6 combinations (about 10^{22}).

2.2. Algorithms

There are basically two families of solving algorithms: the enumeration algorithms, and the prefiltering algorithms. Figure 2 shows a hierarchy of the most popular of these algorithms. This hierarchy reflects the way we programmed these algorithms, and is discussed in section 3.3.

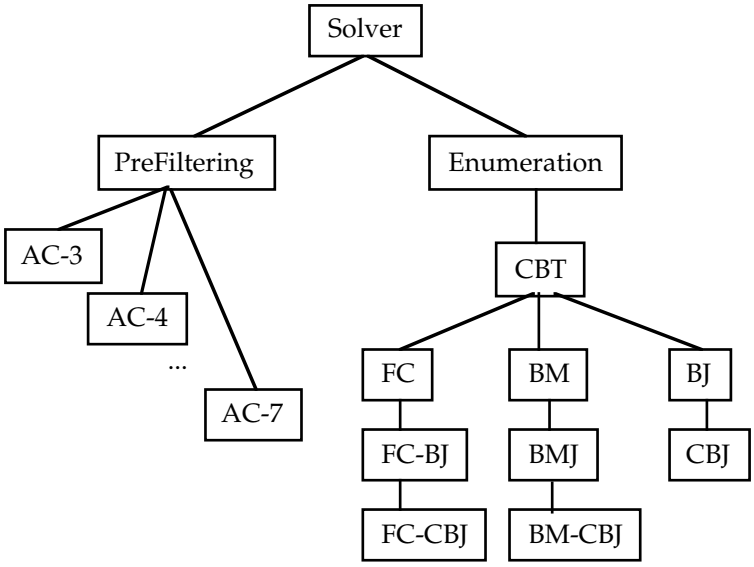

Figure 2. The hierarchy of solvers in BackTalk. `AC` stands for Arc-consistency, `FC` for Forward-checking, `BJ` for Backjumping, `BM` for Backmarking, `CBJ` for Conflict-directed Backjumping.

2.2.1. Filtering algorithms

There is today a wealth of filtering algorithms, the family extends virtually every month. However, most of these algorithms revolve around the same notion of *arc-consistency*, introduced originally by [Mackworth 77].

Solving a CSP amounts to exploring the large space of all possible combinations of values for each variable. A way to prevent search algorithms from combinatorial explosion is to reduce the domain of variables before beginning the actual enumeration of the solutions. The essential notion here is the notion of arc-consistency, which can be seen as a "local" form of satisfaction: a constraint involving two variables `a` and `b` is said to be arc-consistent when, for each value `x` of the domain of `a`, *there exists* a value `y` for variable `b`, such that the couple (`x`, `y`) satisfies the constraint. This property is called arc-consistency because it is local to a given constraint (seen as an *arc* in the network of variables). In particular, the fact that two constraints are arc-consistent does not necessarily imply that there exists a solution that satisfies the two constraints simultaneously! Other notions of consistency extend this basic definition in several ways: path-consistency [Mackworth 77] or k-consistency [Freuder 78].

The first arc-consistency algorithm was Waltz's filtering algorithm [Waltz 72]. Mackworth improved it with AC-3 [Mackworth 77]. Mohr & Henderson designed AC-4 [Mohr & Henderson 86], an optimal worst-case complexity algorithm. Unfortunately, AC-4 is slower

than AC-3 on a lot of CSPs, because is requires too complex data structures [Wallace 93]. The algorithm AC-5 [Deville & Van Hentenryck 91] has been shown to be better than AC-4 on specific CSPs, but not on average cases. AC-6 [Bessière 94] yet improves the theoretical average complexity of arc-consistency, and AC-7 [Freuder et al. 95] is the latest of the family, a specific version of AC-6 exploiting metaknowledge on constraints to reduce the number of checks on specific types of problems. All these algorithms perform differently depending on the structure of the problem. Having all them at hand is therefore useful in the general case.

2.2.2. Enumeration algorithms

An enumeration algorithm is a procedure that yields one or all the solutions of a CSP. As for filtering algorithms, these algorithms have been studied extensively, and are all based on a tree search of the solution space.

The simplest enumeration algorithm is chronological backtracking. This method starts with a predefined order of the variables. It instantiates variables one by one, and checks each partial instantiation against all the constraints. In case of failure, it goes back to the last instantiated variable, and tries the next value in its domain. When a domain has been totally explored, the algorithms "backtracks" one step back in the variables history. This search algorithm is very inefficient because it does not take into account the structure of the problem and therefore can fail several times on the same inconsistencies.

Two main families of enumeration algorithms have been developed to improve the inherent deficiencies of chronological backtracking: look-back and look-ahead.

In case of failure the chronological backtracking goes back to the previous variable. A better strategy would be to backtrack to the variable(s) that actually caused the failure. Such algorithms are called *look-back* procedures. The most famous look-back algorithm is *back-jumping* [Prosser 93a] which backtracks to the last variable linked by a constraint to the current one. This strategy avoids exploring several times the same part of the search space when an inconsistent instantiation occurred earlier.

Another way to improve the efficiency of the search is to prevent future instantiations from failure. Algorithms implementing such mechanisms are called *look-ahead* procedures. In these methods, before instantiating a variable, its domain is reduced. The most efficient one, according to [Nadel 88], is *forward-checking* [Haralick & Elliot 1980]. In forward-checking, only values which are inconsistent with *adjacent* constraints are removed. Another look-ahead

algorithm is *real-full-lookahead* [Nadel 88] which performs a complete arc-consistency procedure after each instantiation.

Finally, attempts at combining both strategies have led to very efficient hybrid algorithms, such as FC-CBJ [Prosser 93b].

To improve on all these algorithms, it can be worthwhile to choose a special *instantiation order* for the variables. The instantiation order can be fixed before beginning the resolution (static heuristic), or it can be modified after each operation on the problem (dynamic heuristic) (see [Dechter & Pearl 88]). To choose the next variable, a frequently used dynamic heuristic is to pick up the one having the smallest domain (first-fail strategy). A frequent static strategy is to choose the most constrained variable.

3. BackTalk

BackTalk can be seen as a class library for stating and solving constraint satisfaction problems in Smalltalk. More precisely, we designed the BackTalk system to achieve the three following goals:

- To provide a *library* of the best available CSP algorithms, that can be used off the shelf, for testing and comparing them. This library should also easily support extensions, in the spirit of the Actalk system [Briot 89], a framework for designing actor languages in Smalltalk.

- To implement these algorithms *without kernel support*: the library should run on any platform supported by Smalltalk, without modification of the virtual machine, in the spirit of the backtrack mechanism of [Lalonde & Van Gulik 88].

- To minimize the well-known *language mismatch*, between the underlying language (here Smalltalk) and the language for describing constraint problems. This language mismatch problem was apparent already in our work on integrating production rules with an object-oriented language [Pachet 95]. Similarly, we want a constraint solver that introduces the less possible new concepts and syntax to Smalltalk.

We will now describe the main features of BackTalk, and give an outline of its design. We will first introduce the main concepts used for *stating* a problem, and then those used for *solving* it.

3.1. Reifying CSPs: stating a problem in BackTalk

Stating a problem in BackTalk is based on the explicit manipulation of the concepts pertaining to CSPs. *Domains* are naturally represented by arbitrary Smalltalk collections. To integrate

BackTalk smoothly in Smalltalk, we do not impose any restriction on the nature of the domains: only that they should be an instance of one of the Collection classes. Three additional notions are introduced to state CSPs: variables, constraints and problems. We will now review each of these classes more in detail (they are prefixed by `BT`).

3.1.1. Variables

Variables are represented by class `BTVariable`, characterized by its name (a symbol), its `domain` (a set of Smalltalk objects) and its `value` (a Smalltalk object). `BTVariable` is a concrete class. A subclass, `BTOrderedVariable`, implements more efficient methods for testing inclusion. Variables are created using message `name:domain:`, taking the name and domain as parameter. The domain may be any kind of Smalltalk collection (an `OrderedCollection`, a `Set`, an `Interval`, an `Array`), having as its elements arbitrary Smalltalk objects. The value of variables are set by the solver during the solving phase.

3.1.2. Constraints

Constraints are represented by class `BTConstraint`, characterized by the list of the variables it involves, and a predicate holding on these variables. `BTConstraint` is an abstract class. A set of predefined constraints are available in BackTalk, represented as concrete subclasses of `BTConstraint`.

To facilitate the declaration of a constraint, BackTalk's `LinearCombination` class is interfaced with a small parser, allowing to state constraint in an infix syntax (e.g. `(2 * x) + (4 * y) = 5`). Each concrete class has its own creation method (see example below).

For the general case, class `BTBlockConstraint` allows to define a constraint whose predicate is an arbitrary Smalltalk Boolean block. Although `BTBlockConstraint` is the most general type of constraint, new classes of constraints can be added by the user for reasons of efficiency (see section 3.4).

The following hierarchy shows a subset of the predefined BackTalk constraints:

```
BTConstraint ('name' 'variables' 'arity')   "abstract class"
    BTBinaryConstraint ('left' 'right' )
       BTBinaryBlock ('block')
       BTBinaryDifference ()                "x <> y"
       BTBinaryEquality ()                  "x = y"
       BTGreaterOrEqualThan ()              "x ≥ y"
       BTGreaterThan ()                     "x > y"
    LinearCombination ('coefs' 'constant') "e.g. 2x + 3y - z = 7"
    BTBlock ('block')
       BTBinaryBlock ()
    BTExtensionConstraint ('extension')    "constraint in extension"
    ...
```

3.1.3. Problems

CSP *problems* themselves are represented by instances of class `BTCSP`. The class defines two instance variables: *variables*, holding the set of variables and *constraints*, the set of constraints. Note that actually only the list of variables suffices to define a CSP: BackTalk deduces the list of constraints from the list of variables, since each variable knows the list of constraints in which it is involved. Problems also have a couple of bells and whistles such as a *print pattern*, a specification of how solutions should be displayed, specially during execution, for tracing purpose (see Figure 3).

Whereas the notion of problem is not relevant to the philosophy of constraint propagation, having problems as objects is particularly interesting in constraint satisfaction. It allows defining several problems simultaneously, each one having its own set of constraints and variables. As a comparison, in IlogSolver, constraints are stated in the object's classes themselves, and are thus diluted in the object space. Having problems as objects allow us to encapsulate the definition of a problem, and enforce a clear separation between the problem and the definitions of the classes involved in the problem.

3.2. Stating a problem with BackTalk

The specification of a CSP in BackTalk consists in creating an appropriate instance of class `BTCSP`, or of one of its subclasses. A typical specification consists in:

1 • Defining the variables of the problem, by instantiating class `BTVariable` with a name and a domain.

2 • Defining the constraints holding between the variables. These constraints are instances of one of the concrete constraint classes.

3 • Instantiating a `BTCSP`, with the list of variables, and optionally a print pattern.

The definition of the crosswords problem proposed in section 2.1 could be represented as a method as follows in BackTalk:

```
perfectCrossWord:  n
"AllEnglishWords is a list of English words such as /usr/words/dict"
   | allVars p allWords |
   allWords:= AllEnglishWords select: [:w | w size = n].
"Definition of the variables with their name and domain"
   allVars := (1 to: n)
         collect: [:c | BTVariable  name: ('h', c printString)
                                    domain: allWords copy].
"Definition of the constraints"
   BTCSP allDifferent: allVars.
   1 to: n do: [:h |
         (h + 1) to: n do: [:v |
               BTBinaryBlock
                     on: (allVars at: h)
                     and: (allVars at: v))
                     block: [:a :b | (a at: v) = (b at: h)]]].
"Instantiation of the CSP"
   ^BTCSP onVariables: allVars
```

We will now see how to solve this problem, using a particular solving strategy.

3.3. Solving a CSP in BackTalk

The Solving phase of CSPs is based on a further reification of the *algorithms* themselves. The whole collection of solver algorithms is represented by class `BTSolver`, an abstract class. Each algorithm is represented by an intance of a concrete subclass of `BTSolver`. Within each family, all the algorithms respond to the same execution messages, defined as abstract methods in root classes. Prefiltering algorithms only understand message `run`. Enumeration algorithms understand various messages such as `firstSolution`, `allSolutions`, and `allSolutionsDo:` an iterator method which takes a block as a parameter, and executes it for each successive solution.

What is a solution ?

Therefore, solving a CSP is done by 1) instantiating a particular solver class, 2) assigning the solver to a CSP as defined above, and 3) sending the solver adequate messages for actually solving the problem. Since values of variables are assigned as a side-effect of the solving procedures, and that variables "know" their value, a solution of the problem is simply materialized by the list of variables. However, since we often want to compute and compare several solutions, the results of enumeration methods (`firstSolution`, `allSolutions`, etc.) is not the list of variables, but a dictionary associating variables to values. This allows the coexistence of several solutions in the environment.

BackTalk currently implements the following consistency algorithms (see the hierarchy of solvers in Figure 1): Mackworth's AC-3 and Deville & Van Hentenryck's AC-5 for binary CSPs, a special version of AC-3 for n-ary CSPs. It implements two different back-jumping algorithms: classical backjumping and conflict-directed backjumping. Standard forward-checking is also available, as well as an algorithm combining forward-checking with backjumping. All algorithms can be run according to any static or dynamic variables instantiation order. Most of them are generalized to handle n-ary constraints.

For instance, the first solution of our preceding problem using forward-checking can be computed as follows (the result is a dictionary representing the first solution):

```
(BTForwardChecking on: aBTCSP) firstSolution
```

To further illustrate the use of solvers as encapsulator objects, let us suppose we want to perform a filtering using AC-5 arc-consistency, followed by a Backjumping enumeration. This would naturally be expressed as:

```
(BTAC5 on: aBTCSP) run.                          " the filtering phase"
(BTBackjumping on: aBTCSP) firstSolution    " the enumeration phase"
```

3.4. Extending BackTalk with user-defined constraints

Class `BTBlockConstraint` allows to specify constraints with any predicate you can express as a Smalltalk block. This gives the user the ability to define new constraints. For instance, a constraint forcing two rectangles to have the same area can be defined as follows:

```
BTBinaryBlock
    on:   rect1                                  "the first rectangle"
    and:  rect2                                  "the second one"
    block: [:r1 :r2 | r1 area = r2 area]     "the predicate"
```

Notice that any Smalltalk Boolean expression (using any Smalltalk message, here message `area`) can be used in the block.

However, it may be interesting, for particularly hard problems, to define specific constraint classes in BackTalk, to increase efficiency. In this case, specific subclasses of class `BTConstraint` may be defined, which redefine the methods responsible for reducing domains during arc-consistency, as well as methods for implementing the predicate. Similarly, one may want to define specific variable classes, to implement more efficient methods for solving, such as interval reasoning. This is done by defining a subclass of `BTVariable` which overrides inclusion test methods.

3.5. The BackTalk Interface

BackTalk provides two main interfaces. The first one is useful to browse CSPs. The second one is used to visualize the execution of a solving algorithm.

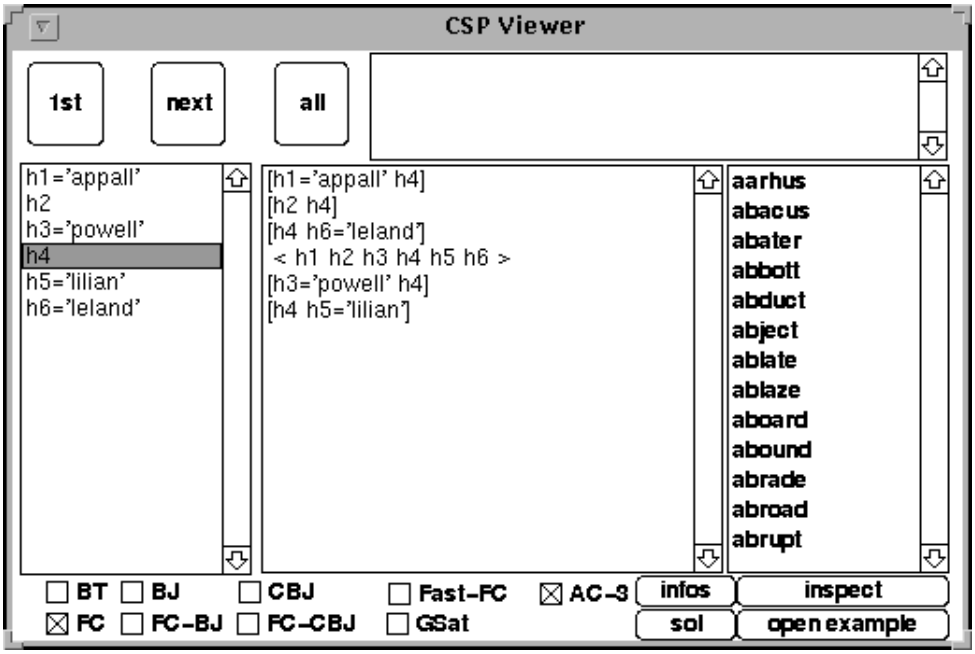

Figure 3. *The BackTalk's CSP-viewer interface, opened on the crosswords example.*

The first interface (called a CSP Viewer, see Figure 3) shows the list of variables, each one with its domain, and the constraints of the problem holding in this variable. Various menus allow to edit constraints, variables or values (removing, filtering, inspecting...). Several buttons trigger the solving algorithms. Variables ordering heuristics may also be defined using specialized fields.

The second interface (called a `SolutionView`) dynamically shows the evolution of a solution as it is built by the solver (a question marks represents a non instantiated variable). The Figure 4 shows an example of an execution snapshot, taken during the solving of the example problem followed in this paper.

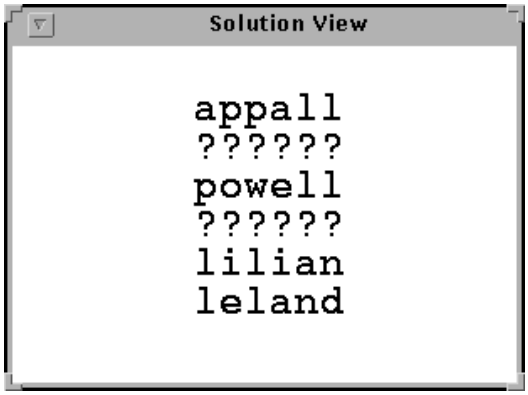

Figure 4. A solution view window opened on an execution of the example problem. Here the variables v1, v3, v5 and v6 are instantiated, the other two are unknown.

4. A musical example

In this section we give an overview of a substantial application written in BackTalk. The application is an automatic harmonization program that has been developed as an extension of a library of musical objects.

A traditional harmonization problem in music teaching consists in composing polyphonic melodies by harmonizing a monophonic melody given as input. The difficult part of the exercise is to ensure that the resulting melody satisfies a set of musical rules. These rules are commonly found in a treatise of Harmony. A typical example is (R1) "two consecutive notes in a same voice must not be more than one octave apart", (R2) "the leading note should rise to the tonic" or (R3) "two successive chords should have different degrees".

It is well-known that this problem can be seen as a case of constraint satisfaction (see e.g. [Ballesta 94], [Tsang 91]; or [Ovans 92]). Each variable corresponds to a note or chord or any other musical object. Musical rules are represented by constraints holding on these variables. The fact that each voice is limited to a finite interval of notes - its range - implies that domains may be represented by finite sets. Hence, the automatic harmonization perfectly fits the conceptual framework of finite-domain CSP.

In our case, we could have solved this problem from scratch, by defining one by one the musical concepts useful for stating the constraints: notes, chords, melodies, degrees, and so forth. Instead, our goal was to reuse as much as possible existing structures, and more precisely, the specialized MusES library [Pachet 94]. This library provides the definition of the most common musical structures in Smalltalk. The size of MusES is around 90 classes and

1500 methods. Of course, the MusES library was not designed with the harmonization problem in mind, but rather as a domain-independent source of information. MusES is used today in a variety of musical applications, ranging from automatic harmonic analysis to simulation of improvization [Ramalho & Ganascia 95].

In MusES all concepts are represented using object-oriented programming (usually methods in the associated classes). For instance, notes are Smalltalk objects (instances of class `Note`), that respond to messages such as yielding its degree within a given scale. Similarly, intervals are represented as objects, and may be computed by sending the message `intervalBetween:` to its lowest extremity, with the other extremity as argument. Constant intervals are represented by special creation messages (e.g. `MusicalInterval octave` yields an object representing the abstract interval of an octave). Chords, melodies, degrees are represented in the same fashion.

Therefore, the ability of BackTalk to handle variables that represent any Smalltalk object, and constraints defined by arbitrary Smalltalk predicates is crucial here, to allow reusing MusES. Using BackTalk, variables are defined with domains being collections of note or chord objects. Constraints are defined using both predefined constraints and block constraints. For instance, the constraint corresponding to the rule: “two consecutive notes, in a same voice, must not be distant of more than an octave”, may be defined simply as follows, where `i` represents a running index on the input melody:

```
BTBinaryBlock                              "Rule R1"
   on: (noteVariables at: i)
   and: (noteVariables at: (i + 1))
   block: [:n1 :n2| (n1 intervalBetween: n2) <= MusicalInterval octave]
```

Similarly, rule R3 can be expressed as a `BinaryBlockConstraint` holding on two instances of class `Chord`:

```
BTBinaryBlock                              "Rule R3"
   on: (chordVariables at: i)
   and: (chordVariables at: (i + 1))
   block: [:ch1 :ch2| ch1 degree <> ch2 degree].
```

The whole harmonization problem has been implemented in BackTalk, by simply reusing MusES as is, and writing the set of constraints corresponding to each musical rule. Details on the actual implementation of the system can be found in [Pachet & Roy 95a], including results on the efficiency of the overall system, which proved to be far better than previous non object-oriented approaches.

5. Discussion

Based on our experience with the system, we discuss here what we claim are the four most important characteristics of BackTalk.

5.1. Reusing class libraries as they are

The first original feature of BackTalk is to allow the reusability of any library of classes, and any system class. In example 4.1 we saw how the MusES library was used to design an efficient harmonization system in BackTalk, without modifying any line of the library. Example 4.2 illustrates the reuse of system classes (here class `String`). Class reuse has also methodological advantages, such as reducing the number of constrained variables when expressing constraints on complex objects (see [Pachet & Roy 95b] for details).

5.2. Providing a library of algorithms

The second original feature concerns the representation of algorithms. BackTalk provides several algorithms to solve CSP, organized in a class hierarchy. Because algorithms are represented as objects, they are easily interchangeable, and can be combined, for instance when different sub-problems in an application require different solving procedures. Typical extensions of the solver framework in progress include solvers to implement radically different approaches such as the GSAT strategies [Selman & al 92], particularly efficient for e.g. Boolean CSPs.

This materialization of algorithms as fully-fledged objects corresponds to the design pattern called *Strategy* , a kind of so-called *behavioral pattern* in the terminology of [Gamma & al 94]. The main advantage of this design is to define a library of interchangeable algorithms, and to allow the library to be extended easily, by creating subclasses of the available classes.

Other patterns can be found in the design of BackTalk. For instance, the execution of algorithms is represented by top-level methods in class `BTSolver` such as `firstSolution`, `allSolutions`, or `allSolutionsDo:` These methods themselves call more specialized methods such as `forward`, `backward`, and `propagateValue:`, which are defined as abstract methods, and defined in concrete subclasses. This design corresponds to the *template method* pattern.

Similarly, the mechanism to save and restore values of variables for backtracking purposes may be described by the *Memento* pattern.

It is interesting to note here that BackTalk is not a framework in the sense of e.g. [Johnson & Foote 88], because algorithms are designed to be used off-the-shelf, and no programming is required from the user to state and solve a constraint problem. But looking closer at the design of the solver hierarchy, this representation of an abstract thread of control specialized by subclasses - using a combination of *strategy* and *template method* patterns - actually perfectly matches the definition of a framework (in e.g. [Gamma & al. 94] p. 26). In other words, BackTalk may be seen as a library from the user point of view, and as a framework from the point of view of a specialist in CSP algorithms.

5.3. Efficiency

Efficiency is an important concern when dealing with constraint satisfaction algorithms. BackTalk was redesigned several times to achieve optimum efficiency while keeping a simple and reusable design. We give here the performance of the system on classical benchmarks: crypto-arithmetics, n queens, and the zebra problem. These benchmarks show that the performance of BackTalk, admittedly not as good as recent versions of IlogSolver (written in C++), is reasonable, and allow scaling-up to real world problems.

Results on a PC Pentium 90 Mhz, using VisualWorks 1.0	
n queens	n = 8 : 0.04s n = 40 : 0.4s n = 100 : 7s
zebra problem	0.3s
send + more = money	0.2s
donald + gerald = robert	1.5s
magic square, size n	n = 3 : 0.3s n = 4 : 12s

5.4. Integration of the system in the object language

In BackTalk, the domains of variables can contain any Smalltalk object. Moreover, we saw that constraints can be defined with any Smalltalk expression (the class `BTBlockConstraint`). These two properties are essential to avoid the language mismatch so frequent in hybrid environments.

5.5. Easy definition of a CSP in BackTalk

As we saw in 3.2 the declaration of a problem in BackTalk is straightforward. Only a small amount of Smalltalk code needs to be written. Specifying a constrained variable only requires

expressing its domain using Smalltalk constructs. Defining a problem only requires specifying the list of its variables, and stating the constraints may be done using all the power of the Smalltalk language.

6. Conclusion

We introduced the BackTalk system, a seamless integration of finite domain constraint satisfaction techniques in Smalltalk[1]. The system brings the power of efficient combinatorial exploration to the hands of the average Smalltalk programmer. The design of the system, seen as a library for defining, combining and using different CSP algorithms, makes it easy to define CSPs integrated with existing class libraries and frameworks, thereby extending them with powerful problem solving capacities. We indeed believe that the CSP paradigm fits particularly well with object-orientation, and that the result is more than the sum of its parts !

7. Acknowledgements

We wish to thank Prof. Jean-François Perrot for his constructive remarks in earlier drafts of the paper and his encouragement in studying integration of AI paradigms with OO techniques.

7. References

Avesani, P. Perini, A. Ricci, F. (1990) COOL: An Object System with Constraints. Proceedings of*TOOLS'2*, pp. 221-228, Bézivin, J. Eds, Angkor, Paris.

Ballesta, P. (1994) Contraintes et objets : clefs de voûte d'un outil d'aide à la composition ? Ph.D. Thesis, INRIA, Sophia Antipolis, November 1994.

Bessière, C. (1994) Arc consistency and arc-consistency again. Artificial Intelligence, 65, pp.179-190.

Borning, A & Freeman-Benson, B. (1995) The OTI Constraint Solver: A Constraint Library for Constructing Interactive Graphical User Interfaces. CP'95, Springer-Verlag, Lecture Notes in Computer Science n. 976. U. Montanari & F. Rosi Eds, pp. 624-628.

[1] The BackTalk system is available on request from the authors.

Borning, Alan, H. (1981) The programming language aspects of ThingLab, a constraint oriented simulation laboratory. ACM transaction on Programming Languages and Systems, 3 (4) pp. 353-387, October 1981.

Briot, J.-P. (1989) Actalk: a testbed for classifying and designing actor languages in the Smalltalk-80 environment, Proceedings of ECOOP'89, pp. 109-130. Cambridge University Press, Cambridge (UK).

Caseau, Y. (1994) Constraint Satisfaction with an Object-Oriented Knowledge Representation Language. *Journal of Applied Artificial Intelligence*, 4, pp. 157-184.

Colmerauer, A. (1990) An introduction to Prolog-III. Communications of the ACM, 33 (7).

Dechter, R. Pearl, J. (1988) Network-based heuristics for constraint-satisfaction problems. Artificial Intelligence, Vol. 34, pp. 1-38.

Deville, Y. Van Hentenryck, P. (1991) An efficient arc-consistency algorithm for a class of CSP problems. Proceedings of IJCAI '91, pp. 325-330.

Freeman-Benson, B. (1990) Kaleidoscope: mixing objects, Constraints, and Imperative Programming. *Proceedings of ECOOP/OOPSLA 90*, Ottawa (Canada), ACM-SIGPLAN notices, Vol. 25, n. 10, pp. 77-88.

Freeman-Benson, B. Borning, A. (1992) Integrating constraints with an object-oriented Language. *Proceedings of ECOOP '92*, Utrecht (NL), Springer-Verlag Lecture notes in Computer Science, n. 615, pp. 268-286.

Freuder, E.C (1978) Synthesizing Constraint Expression. communication of the ACM, nov 1978, vol. 21 (11), pp. 958-966.

Freuder, E.C. Bessière, C. Régin, J.C. (1995) Using inference to reduce arc-consistency computation. *Proceedings of IJCAI'95*, Montréal, pp. 592-598.

Gamma E, Helm R, Johnson R, Vlissides J. (1994) *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Haralick, R.M. Elliot, G.L. (1980) Increasing Tree Search Efficiency for Constraint Satisfaction Problems, Artificial Intelligence, vol. 14, pp. 263-313.

Jaffard, J. Lassez, J.-L. (1987) Constraint logic programming. *14th POPL'87*, Principles Of Programming Languages, Munich, Germany.

Johnson R E and Foote B. (1988) Designing Reusable Classes. *Journal of Object-Oriented Programming* 1 (2): 22-25.

Lalonde, W. Van Gulik, M. (1988) Building a backtracking facility for Smalltalk without kernel support. *Proceedings of OOPSLA '88*, pp. 105-123.

Laurière, J.L. (1978) A Language and a Program for stating and solving combinatorial problems. Artificial Intelligence, Vol. 10, n. 1, pp. 29-127.

Mackworth, A. (1977) Consistency on networks of relations. Artificial Intelligence, Vol. 8, pp. 99-118.

Mohr, R. Henderson, T. C. (1986) Arc and path-consistency revisited. Artificial Intelligence, vol. 28, n. 2, pp. 225-233.

Nadel, B. (1988) Tree search and arc-consistency in constraint satisfaction algorithms. In Search in Artificial Intelligence, Eds. L. Kanal, and V. Kumar, Springer-Verlag, pp. 287-340.

Ovans, R. (1992) An Interactive Constraint-Based Expert Assistant for Music Composition. Proc. of the Ninth Canadian Conference on Artificial Intelligence, University of British Columbia, Vancouver, 1992, pp. 76-81.

Pachet, F. (1994). The MusES system : an environment for experimenting with knowledge representation techniques in tonal harmony. First Brazilian Symposium on Computer Music - SBC&M '94, August 3-4, Caxambu, Minas Gerais, Brazil, pp. 195-201, 1994.

Pachet, F. (1995) Embedding production rules in object-oriented programming languages. JOOP, Vol. 8, n. 4, July/August 1995, pp. 19-24.

Pachet, F. Roy, P. (1995a) Mixing constraints and objects: a case study in automatic harmonization. *Proceedings of TOOLS Europe '95*, Versailles, Prentice-Hall, pp. 119-126.

Pachet, F. Roy, P. (1995b) Integrating constraint satisfaction techniques with complex object structures. *15th Annual Conference of the British Computer Society Specialist Group on Expert Systems, ES'95*., Bramer, M. A. Nealon J. L., Milne, R. Eds, pp. 11-22, Cambridge, 1995, SGES Publications (awarded best technical paper).

Prosser, P. (1993a) Domain filtering can degrade intelligent backtracking search. *Proceedings of IJCAI'93*, Chambéry (France), pp. 262-267.

Prosser, P. (1993b) Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, Vol. 9, pp. 268-299.

Puget, J.-F. (1994) A C++ implantation of CLP. Ilog Solver collected papers. ILOG technical report.

Ramalho, G. & Ganascia, J,-G. (1994) *Simulating Creativity in Jazz Performance*. 12th AAAI Conference, Seattle, AAAI Press, pp. 108-113.

Selman, B. Levesque, H. Mitchell, D. (1992) A New method for solving hard satisfiability problems. Proceedings of AAAI '92, pp. 440-446.

Tsang, C.P. & Aitken, M. (1991) Harmonizing music as a discipline of constraint logic programming. Proceedings of ICMC '91, Montréal, pp. 61-64.

Van Hentenryck, P. (1989) Constraint satisfaction in Logic programming. Logic Programming Series, MIT Press, Cambridge, MA, USA.

Wallace, R.J. (1993). Why AC-3 is almost always better than AC-4 for establishing arc-consistency in CSP. Proc. IJCAI 93, Chambery France. pp. 239-247.

Waltz, D. (1972) Generating semantic descriptions from drawings of scenes with shadows. *MIT Technical report*, AI271, 1972.

7.1. Annex 1*: *Illustration of solving algorithms

To illustrate some of the enumeration algorithms described above, we give here an execution of each one on a simple problem in BackTalk. The problem is the following:

To dress somebody with a pair of pants, a shirt, a hat and a pair of shoes, we dispose of several colors for each item: black, brown and white shoes, black and blue pants, blue, white and brown shirts and a brown hat. We want the final choice to satisfy the following constraints: the hat and the shoes should have the same color; the shoes, the shirt and the pants should have different colors.

We define four variables and four constraints:

4 variables (domain)

1. shoes (black brown)
2. shirt (brown blue white)
3. pants (black brown blue white)
4. hat (brown)

4 constraints

1. shoes = hat
2. shoes <> pants
3. shoes <> shirt
4. shirt <> pants

Chronological backtracking	**Backjumping**	**Forward-checking**
#black #black #black #black #brown #black #brown #brown #black #brown #blue #black #brown #blue #brown #black #brown #blue #green #black #brown #white #black #brown #white #brown #black #brown #white #green #black #blue #black #blue #brown #black #blue #brown #brown #black #blue #brown #green #black #blue #blue #black #blue #white #black #blue #white #brown #black #blue #white #green #black #white #black #white #brown #black #white #brown #brown #black #white #brown #green #black #white #blue #black #white #blue #brown #black #white #blue #green #black #white #white #brown #brown #black #brown #black #brown #brown #black #blue #brown #black #blue #brown	#black #black #black #black #brown #black #brown #brown #black #brown #blue #black #brown #blue #brown #black #brown #blue #green #brown #brown #black #brown #black #brown #brown #black #blue #brown #black #blue #brown	#brown #brown #black #brown #black #blue #brown

This execution trace gives an overview of the relative efficiency of the algorithms on a given problem. Chronological backtracking performs a huge number of instantiations and backtracks to find a solution. Back-jumping goes more directly to the solution thanks to the “intelligent” backtracking heuristic. Finally, forward-checking goes naturally straight to the solution without backtracking at all. If forward-checking is clearly a winner here, the relative performances of algorithms depends heavily on the problem, hence the necessity to provide all of them in a unified environment, and let the user chose the most adapted one.