Integrating constraint satisfaction techniques with complex object structures

François Pachet, Pierre Roy
Laforia-IBP, Université Paris 6, Boîte 169
4, place Jussieu,
75252 Paris Cedex
Tel: (33) 1 44277004
Fax: (33) 1 44277000

E-mail: pachet{roy}@laforia.ibp.fr

Abstract:

Integrating constraint satisfaction techniques with complex object structures is highly desirable. Several libraries are now available to use algorithms off-the shelf and embed them in large object-oriented systems. However, the design of complex object + constraint problems is an open issue that severely limits the applicability of available libraries. We compare two radically different approaches in designing systems integrating objects and finite domain constraints. In the first approach, constraints are defined within classes and constrain attributes of the class, thereby introducing "partially instanciated" objects in the reasoning. In the second approach, constraints are defined outside classes, and constrain fully instanciated objects. We show that, for a particular class of problem, the second solution yields a simpler design while being more efficient. We exemplify our claims by comparing these two approaches on a hard object + constraint problem: four-voice harmonization of tonal melodies, seen as a representative complex configuration problem.

1. Introduction

Finite domain Constraint Satisfaction Programming (CSP) is a well know and powerful representation paradigm for solving complex combinatorial problems. This domain of activity emerged from the confluence of several research fields : constraint logic programming [Jaffard & Lassez 87], [Van Hentenryck 89] operation research and optimization, and artificial intelligence [Laurière 78]. On the other hand, object-oriented programming and representation languages have become commonplace in knowledge representation. We are interested in combining the two paradigms for building large knowledge-based systems. This integration raises two kinds of issues: technical issues, related to the adaptation of existing algorithms to complex object structures in place of the usual atomic values, and conceptual issues related to the design of problems using the combination of OOP and finite CSP.

The paper is organized as follows. In section 2, we briefly introduce the BackTalk system, a canonical integration of objects, in the sense of object-oriented programming, with finite CSP. In section 3, we distinguish between two main approaches in designing an object + CSP problem and compare their advantages and disadvantages on a simple example. We propose a scheme (section 4) that combines the advantages of both solutions, and we report (section 5) on a concrete complex problem: automatic harmonization.

2. Objects and finite domain CSP : the BackTalk system

We are interested in building large knowledge-based systems by combining traditional and sound artificial intelligence techniques with object-oriented technology. Our previous work on

NéOpus [Pachet 95], an extension of Smalltalk to first-order production rules, provided us with valuable experience on the building of such large hybrid systems integrating different, if not orthogonal mechanisms. The key goal of our work is to be able to reuse as much as possible *existing representations* rather than rewriting everything from scratch. Before introducing our approach, we will first review briefly previous attempts in combining objects with various kinds of constraints mechanisms, then introduce the field of *finite domain* CSPs, and finally describe our system BackTalk.

2.1. Integrating objects and constraints

Since the seminal works of Borning on integrating constraints with objects - embodied in the ThingLab system [Borning 81] - many ideas have been proposed to enforce a smooth integration of constraints mechanisms within object-oriented programming languages. This evolution of ideas has materialized for instance in the *kaleidoscope* system [Freeman-Benson & all 90] and its various extensions [Lopez & al 94]. Kaleidoscope integrates several mechanisms for constraint satisfaction including local propagation and simplex for real numbers, and finite-domain solver for Booleans. As [Freeman-Benson & Borning 92] recall, most of the early constraint-based systems were based on a *perturbation model* of constraints. In this model, constraints are used to restore the state of a system after an external perturbation, such as a user interaction. Most of the mechanisms used to enforce this model are based on *local propagation techniques*. Interest has now shifted to the so-called *refinement model* in which the set of possible values of variables is progressively refined through the execution of the program, but never altered by outside events. This motivated our interest in embedding *finite-domain* constraints satisfaction mechanisms with objects.

2.2. Definition of finite domain CSPs

A finite domain CSP (Constraint Satisfaction Problem) is a problem defined by 1) a set of variables taking their value in a finite set of values (the *domain*), and 2) a set of constraints on these variables. A *solution* of a CSP is an instanciation of the variables that satisfies all the constraints. Solving a CSP consists in finding one or all possible solutions, if any. The standard algorithm to solve a CSP is backtracking. Backtracking instanciates progressively the variables, and after each instanciation, checks the partial solution against all the instanciated constraints.

This algorithm is, of course, very inefficient on average. The inherent inefficiencies of backtracking have led to the development of techniques to reduce its complexity. The most widely used technique is *arc-consistency*. It consists in reducing the domains of the variables before or during the actual enumeration, by considering each constraint individually. 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 algorithm. Unfortunately AC-4 is slower than AC-3 on a lot of CSPs, because is requires too complex data structures [Wallace 93]. The recent algorithm AC-5 [Deville & Van Hentenryck 91], has been shown to be better than AC-4 on specific CSPs, but not on average cases. More recent algorithms like AC-6 [Bessière 94] and AC-7 [Freuder et al. 95] yet improve the theoretical average complexity of arc-consistency.

Arc consistency may be applied before the actual backtracking, as well as *during* the enumeration. In this latter case, each instanciation is followed by a more or less complete arc-consistency process. The first one in this family of algorithms (called tree search algorithms) is *Forward-Checking* [Nadel 88]. In forward-checking the consistency process is limited to the constraints involving the currently instanciated variable. In so-called *look ahead* strategies, the consistency process checks all the constraints [Nadel 88]. Another strategy is backjumping [Prosser 93a]. Both strategies can be combined in several ways [Prosser 93b], yielding even more efficient enumeration algorithms.

2.3. *Objects and finite domain CSPs*

An interesting approach is the LAURE system [Caseau 94], which proposes a very efficient implementation of constraint-satisfaction mechanisms embedded with objects. However, LAURE includes a particular object model (called an *object-oriented knowledge representation language*), which, although interesting in many points, is not usable in our context, since we want to reuse existing object-oriented programs. Among the constraint-solvers built as extensions of existing object-oriented languages, our approach is to be compared to systems such as COOL [Avesani & al 90], which integrates a finite-domain solver to the KEE programming environment. KEE objects, however, are closer to frames than to objects in the sense of object-oriented programming. Similarly, the Prose system, integrates finite-domain constraint satisfaction mechanisms on top of Smeci, an object-oriented extension to the Le_Lisp language. The system presented here bears a lot of resemblance in principle to the Prose system, and owes much to the work of Berlandier on algorithms for finite-domain constraint satisfaction [Berlandier 92]. On the commercial scene, the system IlogSolver also proposes finite-domain mechanisms embedded in C++ [Puget 94]. IlogSolver uses proprietary algorithms clearly aimed at efficiency, while providing many hooks for inserting user-specific procedures. IlogSolver as a library has much of the desired features for smooth embeddability, except its lack of a powerful interface and programming environment. As far as Smalltalk is concerned, no finite-domain solver has, to our knowledge, yet been developed.

2.4. *The BackTalk system*

We designed BackTalk [Pachet & Roy 95], an environment to test and compare existing algorithms and their integration with Smalltalk object structures. The main idea of BackTalk is to build a library of existing algorithms and make them available as extensions to arbitrary Smalltalk applications.

The main algorithms implemented in BackTalk are a generalization of AC-3 to n-ary and functional constraints [Berlandier 92] for arc-consistency, and a parameterized enumeration algorithm inspired by [Kökeny 94]. Extensions to AC-4 and AC-5, as well as look-ahead strategies have also been implemented, as well as algorithms combining backjumping with look-ahead strategies. After several experiments of BackTalk on classical problems, we were not convinced of the urgency of implementing much more sophisticated algorithms, such as AC-6 [Bessière 94].

Our contribution in the field is two-fold: First we propose a finite-domain solver smoothly integrated in the Smalltalk language. This allows to apply constraint mechanisms to arbitrary Smalltalk programs. Second, we show that the integration of objects with finite-domain constraints is not so natural as it first seems, and that, in a way, constraints may be "incompatible" with the object structures of the original program. We propose a practical solution to effectively achieve integration while preserving the object structures of the original program.

3. Two main designs for finite CSP + objects problems

We claim that there are two main approaches for the design of problems integrating complex objects structures and finite CSP. In order to exemplify this claim, we will take a simple example in the domain of planar geometry. The example is the following:

Problem Statement (P)
Find all pairs of non trivial quadrilaterals satisfying the following set of constraints (C) :

C1 - All vertices have integer coordinates in {1 .. n}.
C2- For all vertices, the x and y coordinate are different
C3 - All quadrilaterals are straight rectangles.
C4 - The two rectangles do not intersect.

Figure 1 shows a solution of (P) when n = 10.

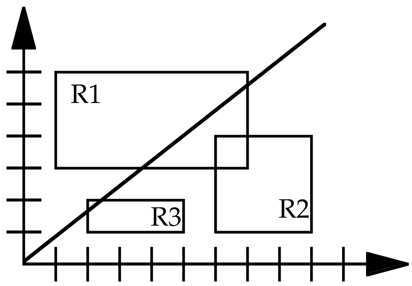

Figure 1. A solution of problem (P) is a pair of rectangles satisfying the set of constraints (C). The pairs of rectangles {R1, R3} and {R2, R3} are possible solutions; pair {R1, R2} is not. In a space with coordinate ranging from 1 to 6, there are 90 solutions.

3.1. Designing the problem as an object + CSP problem

The main task in finding an object + CSP formulation of problem P is to identify the variables to be constrained.

A first formulation consists in specifying the problem only with point variables, and considering points as atomic entities[1]. In this representation the constraints must be stated entirely in terms of point variables. For instance, constraint C4 could be stated as follows:

`x(b) < x(a')`	"R1 on the left of R2"
`or x(a) > x(b')`	"R1 on the right of R2"
`or y(c) > y(a')`	"R1 above R2"
`or y(a) < y(c')`	"R1 below R2"

where a, b, c, d (resp. a', b', c', d') are the coordinates of R1 (resp. R2), and x and y are the functions yielding the x and y coordinate of the points (Cf. figure 2).

In this solution there are :

- 8 variables, representing the 8 vertices, each one with a domain of size (n^2 - n).
- 9 constraints (C1 and C2 are represented as domains, 8 binary constraints for expressing C3, one 8-ary constraint for C4).

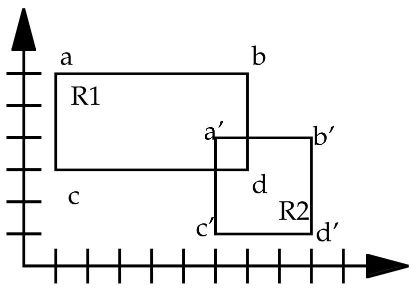

Figure 2. Stating constraint C4 in terms of points.

---

[1] We will see later that our architecture can accomodate the case when only integer are considered atomic, and points considered as aggregates.

The second formulation is based on the remark that the first approach does not make use of existing object structures and methods: since the problem statement contains constraints involving rectangles (C4) and that the domain is defined in terms of `Point` objects, a natural solution consists in specifying a problem with *both* "rectangle variables" (i.e. variables whose domain is collection of rectangle objects) and point variables. The main interest of this solution is to allow the statement of constraints using fully-fledged objects. For instance, constraint C4 could be stated as :

```
not (intersects (r1, r2)),
```

where r1 and r2 are rectangle objects.
In this solution there are :

• 2 variables, representing the 2 rectangles, each one with a domain of size N! / (N-4)!
where $N = n^2 - n$ (the number of possible points).
• 1 binary constraint, for C4.

3.2. Comparison

These two ways of designing the problem lead to various kinds of difficulties in the declaration phase. We now review these difficulties.

In the first solution, constraints are difficult to state, because they involve only "lower-level" objects. In our previous example, imagine a constraint involving three rectangles! Moreover, classes defining the domain objects (here class `Point` and class `Rectangle`) may not be simply reused for the problem solving. In our example, class `Rectangle` is "by-passed" by the definition of constraints (more precisely, constraint C4 bypasses method `intersects`).

In the second solution, constraints are stated using higher-level objects (here class `Rectangle`). These higher-level objects are used as such, (method `intersects` in class `Rectangle`), and play a central role in the formulation. Also, the constraints involve less variables (recall that the arity of constraints is a crucial parameter for performance). In our example, C4 has 8 variables in the first solution, and only 2 in the second one.
The problem is that rectangle variables have no domain a priori. Therefore a lot of objects have to be created: in our case it is roughly equal to the Cartesian product of domain sizes of the variables making up a rectangle. Although the collection of all possible rectangles in a finite 2-dimension space is, of course, finite, it is practically unreasonable to build this collection prior to the resolution.

These two approaches are graphically represented in Figure 3. The first one corresponds to a problem with partially instanciated objects, i.e. objects whose attributes are constrained variables. In the second one, constraints are stated between fully instanciated objects.

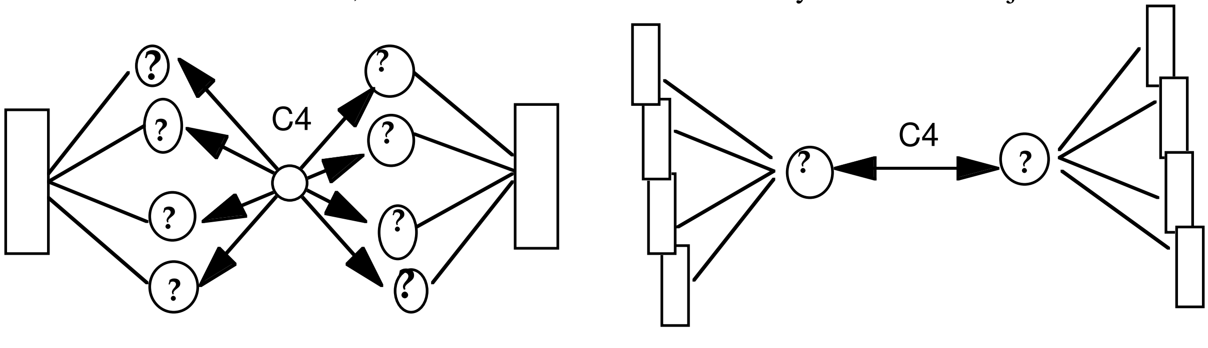

Figure 3. The two design approaches in object + finite CSP problem solving. Solution 1 is on the left, solution 2 on the right.

4. Our approach

Based on the preceding discussion, we propose a scheme that combines the advantages of the second solution (declarativity, possibility of reusing existing classes, constraints with lesser arity), while avoiding the systematic creation of all higher-level objects.

4.1. Separation of the problem in successive resolutions

Our scheme is based on the idea that the number of higher-level objects can be reduced by solving a preliminary CSP with only constraints over lower-level objects. The solutions of this preliminary CSP constitute the domains of the variables representing the higher-level objects.

The first phase has the following steps:
• Creation of a CSP involving only lower-level objects, and constraints between these objects.
• Enumeration of all solutions.

The second phase is:
• Creation of a CSP involving higher-level objects and constraints between these objects. The domain of these higher level variables are the solutions of the preceding CSP.
• Enumeration of all solutions.

In our rectangle example, the first CSP is the following :

CSP 1

- variables: `a, b, c, d` whose domains are the list of all possible points (size n(n-1)),
- constraints: There are four binary constraints, expressing the fact that a, b, c, and d make up a straight rectangle:

```
x(a) = x(c)
and y(a) = y(b)
and x(b) = x(d)
and y(d) = y(c)
```

Let `S` be the set of solutions for CSP1. The size of `S` is n! / (n-4)!. This size is in $O(n^4)$, to be compared with $O(n^8)$ in the naive version of the second solution.

CSP 2

• variables: r_1, r_2 , with domain `S`.
• constraint: one binary constraint expressing the fact that r1 and r2 do not intersect, and using method `intersects` defined in class `Rectangle`.

Finally, CSP2 is solved, yielding all pairs of rectangles satisfying (C). Summing up the two phases, the total complexity of the problem using this approach is therefore in $O(n^4)$, with only binary constraints.

4.2. Generalization

The resolution scheme presented here can be generalized to problems involving more than two levels of composition. For instance, in the problem (P) there are two composition levels if we consider the points as atomic objects, but there are three levels (coordinates, points and rectangles) if we consider points as aggregate structures.

In general, problems may include an arbitrary number, say `c`, of composition levels. In this case we claim that designing and solving the problem into `c` different phases, each phase corresponding to a particular level, leads to considerable improvements:

- When solving a CSP with the first method, the variable ordering heuristic is fixed for the duration of the whole resolution. With our separation scheme, we can use a specific variable ordering heuristic for each phase of the resolution, and more generally exploit available meta knowledge specific to each phase.
- We saw that this separation allows to easily reuse existing classes without any modification.
- The amount of constraints and variables needed, when designing a CSP with our separation scheme, is much less important than with the first solution.

4.3. Execution time on the rectangle example

Computing the exact theoretical complexity of the two approaches is difficult. We give here some experimental results in execution time for each solution. Note that the absolute execution time are not important here (experiments were made on a PC/Pentium), only the relative performance are interesting:

> For n = 4, there are 6 possible straight rectangles, and 6^2 couples of rectangles. There are 14 solutions of (P).
> BackTalk took 15 seconds to enumerate all solutions using the first approach, and 9ms using the second approach with our separation scheme.
>
> For n = 6, there are 90 possible straight rectangles, and 90^2 couples of rectangles. There are 4090 solutions of (P).
> BackTalk took 5 minutes to enumerate all solutions using the first approach, and 2 seconds using the second approach with our separation scheme.

These preliminary results show clearly the advantage of our approach over the first approach, at least in terms of performance.

5. A complex application : Automatic Harmonization

We will now show how the design in the rectangle problem can be applied to a large and complex problem, in the field of musical harmonization.

5.1. The musical problem

Music analysis and generation have long been a favorite domain for researchers in Artificial Intelligence. Within AI, Object-Oriented Programming has traditionally been a favorite paradigm to build complex musical systems: the *Formes* system [Cointe & Rodet 1991] to the *MODE* system [Pope 1991], the Kyma system [Scaletti & Johnson 88]), and *ImprovisationBuilder* [Walker and al. 1992] to name but a few. The "Automatic Harmonization Problem" (hereafter referred to as AHP) is particularly representative of the field. It consists in finding harmonizations of a given melody (such as the melody in Figure 4), or, more generally, an *incomplete* musical material, that satisfies the rules of harmony (and counterpoint, if rhythm is taken into account). The standard AHP is to harmonize four voices (see Fig. 5 for a possible solution).

The constraints needed to solve the AHP are consensual, and can be found in any decent treatise on harmony, such as [Bitsch 57]. The problem is interesting as a benchmark because it involves a lots of complex object structures (notes, intervals between notes, chords, intervals between chords, scales, etc.). Moreover, there are various types of constraints which interact intimately: 1) horizontal constraints on successive notes of a melody, such as: "two successive notes should make an interval of a seventh" or leading note rises to the tonic", 2) vertical constraints on the notes making up a chord, such as "no interval of augmented fourth, except on the fifth degree", or "voices do not cross", and 3) constraints on sequences of chords, such as "two successive chords should not have the same degree".

Figure 4. An initial melody to harmonize (a part of the French national anthem, 18 notes).

5.2. Harmonization as an object + constraint satisfaction problem

Harmonization of a given melody naturally involves the use of constraints, because of the way the rules of harmony are stated in the textbooks. Indeed, several systems have proposed various approaches to solve the AHP using constraints. The pioneer was Ebcioglu [Ebcioglu 92], who designed a specific constraint logic programming language (BSL) to solve this specific problem. His system not only harmonizes melodies (in the style of J.-S. Bach), but is also able to generate new 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. Ebcioglu also uses *real intelligent backtracking*, which is not always more efficient than forward checking algorithms, but a lot more complex to maintain.

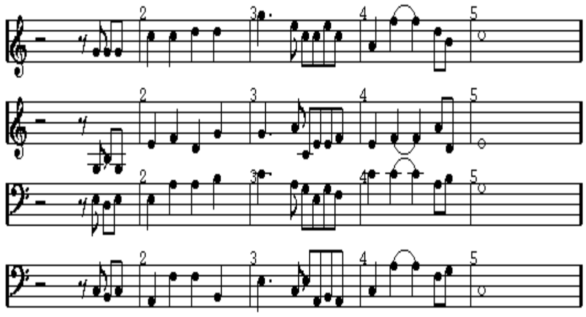

Figure 5. A solution proposed by BackTalk from the initial melody of Figure 4.

More recently, [Tsang & Aitken 91] proposed to solve the AHP using CLP, a constraint extension to Prolog [Jaffard & Lassez 87]. The results of Tsang & Aitken were unrealistic : 5 minutes and 70 Megs of RAM were needed to solve the AHP on an 11-note melody (see Figure 7), making his approach not very encouraging. Ovans [Ovans 92] was the first to introduce the idea of using arc-consistency techniques and CSP to solve the harmonization problem, but his system was very poorly structured, as all the musical concepts had to be represented as number variables, thereby imposing an unnatural bias on the representation of the musical entities.

The system proposed by Ballesta [Ballesta 94] is much more promising. Ballesta uses Pecos (an earlier version of IlogSolver) to solve the AHP, and uses both object structures and finite-domain constraints. The results of Ballesta are listed in Figure 7. The main drawback of this work (in our context) is that Ballesta's system is too radical: everything is stated in terms of constraints, and objects are defined only as structures, designed merely to support the constraints. More precisely, objects are defined by a set of attributes, but all the relations between the attributes are stated in terms of constraints. For instance, to represent one interval instance, 12 attributes are defined, such as the *name* of the interval (e.g. *diminished fourth*), its *type* (e.g. fourth), its two extremities, represented as instances of class `Note`, etc. Nine constraints are then introduced to state the relations that hold between the various attributes of class `Interval`. For instance, a constraint links the name of the interval to the various attributes of its extremities (the octave and name of the note). As a result, his representation is indeed very rich, since any request can be made on any partially instanciated interval. For instance, the user can ask for the set of notes yielding an interval of a fourth with a given note, etc. The same scheme is applied to all the entities of the domain: notes, intervals, scales and chords. One note instance is represented by 6 constrained variables, one interval by 9 constrained variables, and so forth. To solve the AHP on a n-note melody, Ballesta uses (126*n - 28) constrained variables. The total amount of constraints is therefore very high, while the total number of methods is very low.

5.3. Critics of preceding approaches

As Figure 7 shows, the approaches which have been proposed using constraints yield poor results in terms of performance. There are, from our point of view, two lessons to learn from these experiments:

1- There are too many constraints. The approaches proposed so far do not structure the representation of the domain objects (notes, intervals, chords). When such a structure is proposed (as in Ballesta's system) objects are treated as passive structures.
2- 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 problem, 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 and the constraint satisfaction mechanism 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 propose to reuse a fully-fledged object-oriented program, the MusES system, which contains a set of classes that represent the basic elements of harmony, such as notes, intervals, scales and chords. We start from MusES and add constraints on top of it to represent the rules of harmony. Not only the resulting system will be faster (because methods are faster than constraints), but we claim that it also will be more intelligible.

5.4. The analogy between AHP and the rectangle problem

Seen from this point of view, the AHP is analogous to the rectangle problem discussed above. The analogy is the following: notes are analogous to points, chords to rectangles. As far as constraints are concerned, the analogy is still valid: there are constraints on notes only (range constraints), constraints between notes, and constraints between chord objects (the equivalent of constraint C4). Following this analogy, Ballesta's system correspond to the first approach described in the rectangle example. We will now propose a solution of the same problem using the second approach with our separation scheme.

5.5. The MusES system

The MusES system is a project to represent consensual knowledge about basic harmony in Smalltalk [Pachet 94]. From the musical point of view, the aim of MusES is to study the adequacy of various representation techniques to capture the essence of musical structures, starting from the most simples ones (notes, enharmonic spelling, intervals, scales, and so forth) to the most sophisticated ones (analysis, tonalities, support for improvisation, etc.).

MusES contains around 90 classes and 1500 methods. All these operations are represented using object-oriented programming (usually methods in the associated classes). Of course, this approach is less general than the one using purely constraints. Our approach is indeed much different: instead of proposing a general framework in which relations are stated and arbitrary computations are at the user's hand, we impose a fixed set of "essential" computations which are fast, while being easy to understand and modify. The example of the notion of interval is typical. The MusES approach is based on the remark that only three types of operations are important with intervals: (1) computing an interval given 2 notes, (2) computing the top note given the bottom one, and 3) computing the bottom one given the top one. Several other less important operations are also considered, like adding two intervals. MusES therefore contains specific methods to compute intervals given two notes, or notes given an interval.

Several extensions are currently developed to MusES to test various ideas in musical representation [Pachet 91], [Ramalho & Ganascia 94], thereby validating the construction of MusES as a general purpose library.

5.6. *BackTalk and MusES to solve the AHP*

In accordance with the analogy given above, we apply our resolution scheme to AHP as follows, by breaking the problem into two parts:

1) Management of the constraints at the note level only.
Input of the n-note melody and creation of a CSP with only constraints on notes.
Arc-consistency is applied to reduce the domains of the note variables.

2) Management of the constraints at the chord level.
Computation of the concrete instances representing all the possible chords under each note.
Creation of a second CSP including only the higher level objects, i.e. chords variables and constraints involving chords.
Arc-consistence is applied, followed by the enumeration of solutions with forward-checking.

In this scheme, given a n-note melody, the total CSP contains (4*n) variables for the notes plus n variables for the chords, which are handled in the second phase. The results are given in Figure 7. As we can see, we are an order of magnitude faster than previous approaches. The memory needed is not significant.

1
- Input n-note melody
- Build CSP with constraints on notes (4n variables)
- Arc-consistency
- Computation of chord domains

2
- Creation of second CSP with n constrained variables (chords)
- Arc-consistency
- Enumeration by forward-checking

Figure 6. Diagram of the architecture.

	11 notes	**12 notes**	**16 notes**
Tsang&Aiken (CLP)	5 m. Sparc 1, 70Mg ram	?	?
Ballesta (Pecos)		3 m.	4 m.
BackTalk + MusES	12 sec.	13 sec.	20 sec.

Figure 7. Comparative results of BackTalk + MusES.

6. Conclusion

We are interested in the integration of finite domain CSP with object oriented programming languages, from a technical and a conceptual point of view. We introduced BackTalk, a testbed for studying the integration of current algorithms with Smalltalk. We showed on a simple example drawn from geometry that two main approaches are possible for the design of complex objects + CSP problems. We compared these solutions and showed their respective advantages and drawbacks. We propose a scheme based on the separation of the problem in successive phases that allows to combine the advantages of both approaches. Each phase corresponds to a level in the composition hierarchy of the domain objects. We illustrate our scheme on a problem known to be complex (automatic harmonization) and show that our approach clearly outperforms preceding attempts at solving the same problem. These results encourage us to continue studying our approach. In particular, we are trying to find formal specifications of complex object + CSP problems for which our scheme can be safely applied.

7 . Reference

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

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.

Berlandier, P. (1992) Etude de mécanismes d'interprétation de contraintes et de leur intégration dans un système à base de connaissances. *Ph.D. Thesis*, INRIA, 1992.

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

Bitsch, M. (1957) Précis d'Harmonie tonale. *Ed. Alphonse Leduc*.

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.

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

Cointe, P. Rodet, X. (1991) Formes: Composition and Scheduling of Process. In *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology* , S. T. Pope, ed. MIT Press.

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

Ebcioglu, K. (1992) An Expert System for Harmonizing Chorales in the Style of J.-S. Bach, In M. Balaban, K. Ebcioglu & O. Laske (Ed.), *Understanding Music with AI: Perspectives on Music Cognition*, The AAAI Press, California.

Freeman-Benson, B. (1990) Kaleidoscope: mixing objects, Constraints, and Imperative Programming. *Proceedings of ECOOP/OOPSLA 90*, Ottawa, pp. 77-88.

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

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

Jaffard, J. Lassez, J.-L. (1987). Constraint logic programming. *14th POPL*, Munich, 1987.

Kökeny, T. (1994). Yet another object-oriented constraint resolution system (YAFCRS) : an open architecture approach, *Proceedings of TOOLS USA 94*, Prentice-Hall, Santa Barbara, pp. 103-114.

Laurière, J.L. (1978). A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, Vol. 10, pp. 29-127.

Lopez, G. Freeman-Benson, B. Borning, A. (1994). Constraints and Object Identity. *Proceedings of ECOOP '94*, Bologna (Italy), Springer-Verlag Lecture Notes in Computer Science, vol. 821, pp. 260-279.

Mackworth, A. (1977). Consistency on networks of relations. *Artificial Intelligence*, (8) pp. 99-118, 1877.

Nadel, B. (1988) Tree search and arc-consistency in constraint satisfaction algorithms. *Search in Artificial Intelligence*, Springer-Verlag, pp. 287-340.

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

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.

Pachet, F. (1991) A meta-level architecture for analyzing jazz chord sequences. *International Conference on Computer Music*, pp. 266-269, Montreal, Canada.

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.

Pachet, F. (1995) On the embeddability of production systems in object-oriented languages. *Journal of Object-Oriented Programming* , Vol. 8, n. 4, July/August 1995, pp. 19-24.

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

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) Programmation logique sous contraintes en C++. *Proceedings of "Langages et modèles à objets" LMO'94*, Grenoble (France), October 1994.

Pope, S. (1991) Introduction to MODE: The Musical Object Development Environment. In *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology* , S. T. Pope, ed. MIT Press.

Ramalho, G., Ganascia, J.-G. (1994) Simulating Creativity in Jazz Performance. *Proceedings of 12th AAAI conf.* Seattle, Aug. 1994.

Scaletti, C. Johnson, R. E. (1988) An interactive environment for object-oriented music composition and sound synthesis. *Proceedings of OOPSLA '88*, pp. 222-233, San Diego.

Tsang, Chi Ping & 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. MIT Press, Cambridge, MA, USA.

Walker, W., Hebel, K., Martirano, S., Scaletti, C. (1992) ImprovisationBuilder: improvisation as conversation, *Proc. of ICMC* , 1992.

Wallace, R.J. (1994) Why AC-3 is almost always better than AC-4 for establishing arc-consistency in CSP. *Proceedings of IJCAI 93*, Chambéry, France, pp. 239-247.

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