Mixing Constraints and Objects: a Case Study in Automatic Harmonization

François Pachet & Pierre Roy
LAFORIA-IBP, Université Paris 6, 4, Place Jussieu, 75252 Paris Cedex 05, France
e-mail: pachet/roy@laforia.ibp.fr
www: http://www-laforia.ibp.fr/~fdp

Abstract

We propose an extension of Smalltalk with finite-domain constraint satisfaction mechanisms. Our system, called BackTalk, allows the definition of constraints over arbitrary Smalltalk objects, and implements efficient algorithms for constraints satisfaction. We exemplify the use of BackTalk on a problem known to be complex, automatic harmonization. We outline several previous attempts to solve the problem with similar mechanisms, and stress on their inefficiencies, mainly the lack of structure of domain objects. We propose to solve the problem by a combination of constraints and objects that fully benefits from object structures. This is achieved in practice by separating the constraint satisfaction process in two steps. By comparison, our system yields excellent results, both in term of efficiency and readability. We discuss the generality of our approach to problems involving numerous and heterogeneous object structures.

Key-words: constraints, finite-domain constraint satisfaction, embedded constraint systems, automatic harmonization.

1. Yet an other object + constraint 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 software* rather than rewriting everything from scratch. This paper is a report on a successful experiment in building a constraint satisfaction mechanism embedded in Smalltalk, with an application on a complex problem.

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 smoother 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.

An interesting approach is the LAURE system [Cazeau 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 [Berlandier & Moisan 88], 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 94]. On the commercial scene, the system IlogSolver also proposes finite-domain mechanisms embedded in C++ [Puget 92]. 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.

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 report on an application of our system on a problem known to be complex: four-voice automatic harmonization. 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.

The paper is organized as follows. In the first part, we briefly describe the kernel of our integrated finite-domain solver (BackTalk, for Backtracking in Smalltalk). In the second part we introduce the (hard) problem of automatic harmonization, and show the results of preceding attempts to solve it with constraint-based systems. Note that this article does not require any knowledge of harmony or music whatsoever to be understood. In the third part, we introduce the object-oriented system we reused (the MusES system), and propose a solution to solve the harmonization problem with MusES and BackTalk. We discuss our result, and particularly the generalization of our technique to complex objects + constraint problems involving many different object structures.

2. Overview of BackTalk

BackTalk stands for "Backtracking in Smalltalk". The aim of this system is to provide a set of classes to state and solve constraints on arbitrary Smalltalk objects. This work follows the same spirit as previous backtracking extensions to Smalltalk such as [Lalonde & Van Gulik 89], [Lalonde 87], in that it does not require any kernel support, and constraints may be expressed on arbitrary Smalltalk objects. The system is entirely written in Smalltalk. Considerations on efficiency may be found in 2.3.

2.1. Definition of a CSP

A 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 a solution, or all its solutions. i.e. assignments of values to variables that satisfy all the constraints. 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 to reduce this complexity 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 found 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. The most recent algorithm is AC-5 [Deville & Hentenryck 91], which is better than AC-4 on specific CSPs, but not on average cases. 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].

In BackTalk, we use a generalization of AC-3 to n-ary and functional constraints [Berlandier 92] for arc-consistency. Enumeration of solutions is implemented by forward-checking. Extensions to AC-4 and AC-5, as well as look-ahead strategies will be eventually considered. However, after several experiments of BackTalk on classical problems, we were not convinced of the urgency of implementing much more sophisticated algorithms.

2.2. Implementation of BackTalk

BackTalk is a simple system, with straightforward representations of variables and constraints. BackTalk introduces one class for defining variables, and several classes to define constraints. Each constraint class represents a particular type of constraint, with redefined methods for improving efficiency, as well as for deducing variables when it is possible. For example, class `Constraint` defines general n-ary methods for filtering variable domains. In the case of `BinaryConstraint`, this procedure is redefined to implement the more efficient procedure "revise" of [Mackworth 77]. BackTalk includes a library of constraint classes including constraints for most common arithmetic computations (Cf. Fig. 1).

```
Object ()
"general constraint (abstract class)"
  Constraint ('attributes' 'arity')
"X1+...+Xn=Y"
  AdditiveConstraint ('orderedAttributes')
    BinaryConstraint ('left' 'right')
"X <> Y"
  BinaryDifferentConstraint ()
"X = Y"
  BinaryEgalityConstraint ()
"X > Y"
  BinarySuperiorConstraint ()
        BinarySuperiorOrEqualConstraint ()
        ...
    BlockConstraint
        ('relationBlock' 'orderedAttributes')
    TernaryConstraint ('first' 'second' 'third')
        TernaryAdditiveConstraint ()
        TernaryEgalityConstraint ()
        ...
    UnaryConstraint ()
```

Figure 1. A part of the hierarchy of constraint classes in BackTalk.

Class `BlockConstraint` is used to specify arbitrary constraints, whose test is represented by a Smalltalk block. The only requirement is that the block yields a Boolean result. Testing the constraint is done by evaluating the block on his arguments.

The user may either use a predefined class of constraint among the classes of the library, or define a new one. This new constraint must be a subclass of `Constraint`, and only one method has to be redefined to make it a concrete class. This method represents the actual test of the constraint, when all the variables are instanciated. In the case the new constraint class represents a functional constraint, a second method must be defined, that computes the deduced variable from the value of the other variables. This method is used to compute infinite domains when needed.

2.3. Example

Here is a simple example of a CSP, that gives an idea of the look and feel of BackTalk, with the n-queens problem (Cf. Fig. 2). We define n variables whose domain is the list of all possible n^2 squares on the chessboard. The squares are represented by instances of class `Square`, who understand message `attacks:`. We define n(n+1)/2 constraints between each pair of variables, to represent all the constraints between the queens. No arc-consistency here is needed since the problem is by definition already arc-consistent. The result of the enumeration is a dictionary which associates a value to each variable. Methods allows more sophisticated exploitations of the results, such as `allSolutionsDo:`, which successively applies a block to each solution. Of course, the n-queen problem may be solved a lot faster , e.g. by imposing the row of queens, but what we want to show here is that the domain of variables may be a list of arbitrary Smalltalk objects, here instances of class `Square`.

```
| p queens d|
p := CSP new.                    "The CSP"
queens := (1 to: n) collect: [:i |
  "The n variables"
  d :=  OrderedCollection new.
  1 to: n do: [:i | 1 to: n do:
      [:j | d add: (Square newAt: (i @ j))]].
"The domain"
  Variable new domain: d].

1 to: n - 1 do: [:i | i + 1 to: n do: [:j |
  p addConstraint:
      (BlockConstraint new
          "The constraints"
      variables: (Array with: (queens at: i)
                        with: (queens at: j))
      relationBlock: [:a :b | (a attacks: b)
                               not])]].
"the first solution"
p firstSolution -> aDictionary
"all solutions"
p solutionsDo: [:d| p printSolution: d]
```

Figure 2. The n-queens problem in BackTalk.

2.4. Efficiency

Efficiency was not our primary concern in building BackTalk, but we ended up with decent results for a language not known to be the most efficient (Cf. Fig. 3). Of course, compared to extremely efficient object + constraint solvers such as LAURE [Cazeau 94] for instance, our results are poor. However, as we will see on the problem example, we think that the main gain in efficiency is not to be found in the implementation of general-purpose algorithms. To quote Freeman-Benson and Borning [Freeman-Benson & Borning 92], we think that "Providing a general solver for arbitrary constraints over arbitrary domains is a completely unreasonable goal". This remark led us to search for better ways of organizing constraints and objects that are constrained, rather than in algorithm optimizations. As we will see, the results obtained by our system on the harmonization problem are an order of magnitude better than similar approaches using more sophisticated solver engines, which makes the BackTalk system, although not optimized, a fast enough engine for our experiments.

Problem	Nb. of variables	Nb. of constraints	Time (Sparc 10)
send+ more = money	11	35	5 s.
n-queens n = 8 n = 100	n 8 100	n.(n+1)/2 36 5050	 300 ms. 40 s.

Figure 3. Some results of BackTalk on classical problems.

3. The musical problem

Music analysis and generation has 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, especially systems oriented towards synthesis (from 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).

3.1. Harmonization as a 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 proposed to solve the AHP using CLP, a constraint extension to Prolog [Jaffard & Lassez 87]. The results of Tsang 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. He 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.

3.2. 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 structuration 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.

4. 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.). From the technical point of view - the one we follow here - MusES can be seen a big repository of consensual knowledge about harmony. One of the goal of MusES is to study how complex object-oriented programs may be extended with AI techniques, for effective reuse. To understand this point, we must say a little bit more about MusES.

MusES is entirely written in Smalltalk, using only the basic mechanisms of Object-Oriented Programming (instanciation, encapsulation, message passing and polymorphism, metaclasses). 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 contains specific methods to compute intervals given two notes, or notes given an interval, such as the following:

```
(1)
PitchClass C intervalWith: PitchClass F sharp
        ->          an augmented fourth
(2) Interval diminishedFifth topIfBottomIs:
PitchClass G ->          C#
```

MusES also includes a representation of scales as classes in the same spirit (ex. (3). These scales can generate so-called *scale-tone chords*, which are at the root of harmonic analysis (example (4)), and conversely, MusES can compute the list of plausible tonalities for a given chord (example (5)):

(3) PitchClass C harmonicMinor notes ->
#(C D Eb F G Ab B)

(4) PitchClass C harmonicMinor scaleToneChordsPoly: 4 ->
([C min maj7] [D halfDim] [Eb aug5 maj7] [F min 7] [G 7] [Ab maj7] [B dim7]))

(5) Chord fromString: 'C min' allPossibleTonalities ->
AnalysisList ({II of Bb MajorScale}
{II of Ab MajorScale} {VI of Eb MajorScale}
{I of C HarmonicMinorScale}
{IV of G HarmonicMinorScale}
{I of C MelodicMinorScale}
{II of Bb MelodicMinorScale}))

MusES also contains a representation of temporal objects (notes in a melody), and more abstract structures, such as melodies. A set of editors are included in MusES to define and edit melodies graphically. MusES contains around 50 classes. 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.

Several extensions are currently developed to MusES to test various ideas in musical representation. The first one is a system that performs automatic analysis of Jazz chord sequences [Pachet 91]. This systems studies the integration of first-order rules in Smalltalk, and proposes a model that captures the human reasoning process involved [Pachet 94b]. Another one is a model for musical memory, aimed at producing an automatic improviser [Ramalho & Ganascia 94].

5. BackTalk and MusES: stating constraints on objects that do not exist yet

Given MusES, the main problem we face in writing the constraints that represent the rules of harmony, is that there are constraints that should be stated on objects that do not exist yet! Typically, chords - the most abstract objects of all - do not exist yet at the beginning of the problem solving. Their existence is dependent on the existence of the notes of intermediary voices, which are themselves computed by constraints. We cannot simply write all the constraints and launch BackTalk on them, since the domains of the chords are empty. The solution we propose consists in separating the problem solving in two phases (Cf. Fig. 6):

1) Management of the constraints at the note level only.
Input of the n-note melody.
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. This corresponds to computing the infinite domains of the chords variable.
Creation of a second CSP including constraints on notes, and constraints on chords (which exist now), as well as constraints on notes and 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) and the preceding 4.n variables (notes)
- Arc-consistency
- Enumeration by forward-checking

Figure 6. Diagram of the architecture.

6. Results

	11 notes	12 notes	16 notes
Tsan&Aiken (CLP)	5 m. (Sparc 1) 70Mg ram	?	?
Ballesta (Pecos)		3 m.	4 m.
BackTalk + MusES	20 sec.	20 sec.	40 sec.

Figure 7. Comparative results of BackTalk+MusES.

7. Discussion, conclusion

We introduced BackTalk, a finite-domain constraint solver in Smalltalk, having decent but not extraordinary performances. The performance of the BackTalk kernel have been showed to be sufficient for our purposes on a problem known to be complex. The relative loss in efficiency, compared to more efficient implementations, is in our view, largely compensated by the possibility of building very easily editors for the constraints (not described here for reasons of space), and by allowing the dynamic modification of the CSP. Future works on the kernel include adding mechanisms to handle constraints hierarchies, [Borning & al. 87], [Berlandier 94].

We solved the automatic harmonization problem by reusing an existing system, and adding a set of constraints of top of it. By reusing the existing object structures, we were able to reduce drastically the number of constrained variables, and thus to get results one order of magnitude better that previous approaches on the same problem. The resulting system is not only faster than the other proposed solutions, it is also much simpler (less constraints). Compared to other approaches using constraints, we achieve better results by finding a compromise between two extremes (Cf. Fig. 8): 1) rich objet structures, with no flexibility (the MusES system), and 2) lots of flexibility, but no structure (The Tsang & Aitken system).

This experiment shows that the combination of objects with finite-domain constraints may increase drastically the performance, if the architecture is able to take into account objects structures adequately. In particular, our good results are obtained because, in a way, some constraints are "compiled" into methods of corresponding classes. This compilation forces the problem solving process to be cut into two phases. This separation in two phases is to be seen as a representation of a particular kind of knowledge about the types of objects. More generally, we replace "inter-object" constraints (for instance constraints defining chord structures or intervals), by exploiting rich object structures (here, essentially the chords), represented as "ghost" objects from the point of view of the constraint mechanism.

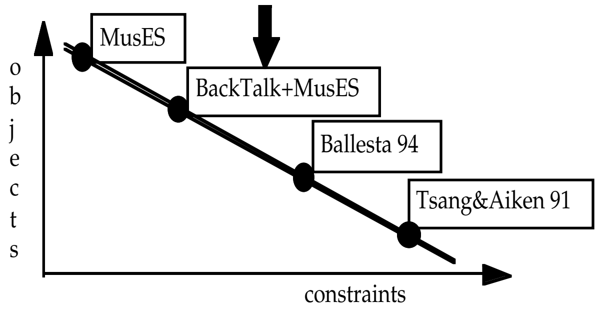

Figure 8. Position of BackTalk+MusES among other approaches.

This successful experiment is now followed by more formal research to find properties that could help automatically detect cases when the separation of the problem solving in distinct phases could help reducing the complexity of large constraint+object systems. Our hypothesis is that this separation may be particularly useful when several ontologically distinct categories of objects are being handled simultaneously. More precisely, we think that such situations should arise when the domain objects contain individuals (analogous to notes), groups of individuals (analogous to chords), groups of groups of individuals (analogous to musical phrases), etc. Experimentation on large configuration problem for instance, should prove interesting to test the validity of these ideas.

References

Avesani, P. Perini, A. Ricci, F. (1990) COOL: An Object System with Constraints. *TOOLS'2*, June 1990.

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.

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, October 1981.

Borning, A. Duisberg, R. Freeman-Benson, B. Kramer, A. Woolf, M. (1987) Constraint hierarchies. *OOPSLA '87*, pp. 48-60. (1987).

Cazeau, 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. Kaleidoscope: mixing objects, Constraints, and Imperative Programming. *Proceedings of ECOOP/OOPSLA 90*, pp. 77-88.

Freeman-Benson, B. Borning, A. (1992) Integrating constraints with an object-oriented Language. *Proceedings of ECOOP '92*, pp. 268-286.

Fron, A. (1994) Programmation par contraintes. Addison-Wesley, 1994.

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

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

Lalonde, W. (1987) A novel rule based mechanism in Smalltalk. ECOOP '87.

Lopez, G. Freeman-Benson, B. Borning, A. (1994). Constraints and Object Identity. *Proceedings of ECOOP '94*, 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, 1988.

Nudel, B. (1983) Consistent labeling problems and their algorithms: expected complexity and theory-based heuristics. *Artificial Intelligence*, vol. 21, n. 1 & 2, pp. 135-178, 1983.

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) An object-oriented representation of pitch-classes, intervals, scales and chords. *Journées d'informatique Musicale*, Bordeaux, march 1994.

Pachet, F. (1994b) A Refined Framework for Representing Knowledge Based on Simulation. *Colloque Langages et modèles à objets*, Grenoble, October 1994, to be published.

Pachet, F. (1995) On the embeddability of production systems in object-oriented languages. *Journal of Object-Oriented Programming* , Jan. 1995. To appear.

Puget, J.-F. (1992) Intensional and cardinality constraints. *Internal report*. Ilog, 1992.

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.

Tsang, E. Foundations of constraint satisfaction. *Computation in Cognitive Science*. Academic Press, 1993.

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

Winograd, T. (1993) Linguistics and the Computer Analysis of Tonal Harmony. In *Machines Models of Music*, Edited by S. M. Schwanauer and D.A. Levitt, MIT Press.