Contents

Chapter 1

Variable Orderings for Solving Musical Constraint Satisfaction Problems

1.1. Motivation

Constraint programming [DEC 03, ROS 06] is an established approach for implementing computational models for various music theories. Many compositional tasks have been addressed by constraint programming: [PAC 01] survey constraint-based harmonisation, other examples include Fuxian counterpoint [SCH 89], Bach choral harmonisation [EBC 92], purely rhythmical tasks [SAN 03], Ligeti-like textures [CHE 01], and instrument-specific writing [LAU 01].

The attraction of constraint programming for modelling music theories is easily explained. Music theories are often expressed by a set of rules that forms a combinatorial problem. Constraint programming allows users to model such problems in a relatively simple way. A problem is stated by a set of *variables* (unknowns), a *domain* for each variable (a set of its possible values), and *constraints* (relations) between these variables. For example, the note durations and pitches in a music representation can be variables whose domains consist of sets of note values and pitches allowed for these variables. Compositional rules on durations and pitches can be implemented by constraints. In the terminology of constraint programming, the modelled problem is referred to as a *constraint satisfaction problem* (CSP). In a solution to a CSP, every variable is assigned to a single value from its original domain that is consistent with all its constraints. The solution to a musical CSP is typically a fully-determined score (or score excerpt such as a chord progression expressed by Roman numerals), and

Chapter written by Torsten ANDERS.

the solution is consistent with all constraints expressed by the rules. Existing constraint systems can efficiently solve a CSP – a fact that has greatly contributed to the popularity of constraint programming.

Several music constraint systems have been proposed in which musicians can define their own musical CSPs. The seminal systems are PWConstraints [LAU 96], and Situation [RUE 98, BON 99]. These systems – alongside other music constraint systems – are surveyed in detail by [AND 11].

Musical CSPs can be extremely complex: the search space of these problems – the set of all their partial solutions – can be huge. To be useful in practice, a constraint system must be reasonably efficient. It makes a big difference whether a CSP takes seconds or hours to solve. However, the search process implemented by existing systems is optimised for specific problem classes to the detriment of others. Problems for which these systems are not optimised either perform poorly or cannot be defined at all. For example, Situation was originally designed for solving harmonic problems and has been extended for rhythmical problems, whereas the PWConstraints subsystem Score-PMC is intended for contrapuntal problems. However, Situation is ill-suited for contrapuntal problems, and Score-PMC makes it impossible to constrain the rhythmical structure. Due to their search strategy (besides other aspects such as their music representation), none of these systems support problems in which the rhythmical, harmonic, and contrapuntal structure is constrained at the same time.

Experience in constraint programming in general has shown that the order in which variables are visited during the search process can have an immense impact on the size of the search space that is explored before a solution is found, and hence the efficiency of the search process [KUM 92]. This general experience has been confirmed for music constraint programming. For example, Score-PMC applies a sophisticated variable ordering, which is optimised for polyphonic music and is well-suited for arbitrarily complex rhythmical structures [LAU 96].

A suitable variable ordering is highly problem-dependent. For example, contrapuntal and harmonic CSPs are generally more efficiently solved by different variable orderings. A contrapuntal CSP results in two or more simultaneous voices that accompany each other, but which are rhythmically and melodically relatively independent. Such a CSP is often solved best by processing from left to right in the score and completing all voices more-or-less in parallel. Section 1.4.5 explains further why this is the case. By contrast, the pitch structure of a harmonic CSP is based on an underlying harmonic progression (commonly expressed by Roman numerals, e.g., $I\ ii\ V\ I$). Harmonic CSPs are often solved best by first deciding upon this underlying harmonic progression, and then for the actual notes. Also, the bass and soprano voices are more important in a homophonic texture, where all voices more or less share the same rhythm, and it is therefore beneficial to decide for a bass and soprano note before deciding for simultaneous notes of other voices.

Most existing music constraint systems apply a *static variable ordering*: the order in which variables are visited during the search process is fixed before the search actually starts. However, musical CSPs can be extremely complex and it can be impossible to find an efficient static variable ordering. Consequently, only specific CSP classes are solved efficiently by a single static variable ordering, and some systems even make the definition of other CSPs impossible. For instance, Score-PMC only supports CSPs that fully determine the rhythmical structure in the problem definition: Score-PMC depends on this rhythmical structure to compute its efficient static variable ordering [LAU 96].

The search of Score-PMC proceeds 'from left to right' in the polyphonic score as described above: basically, notes with a smaller start time are visited earlier. However, to compute such a variable ordering before the search starts, it is of course essential that the temporal structure of the score is fixed.

A *dynamic variable ordering*, on the other hand, is not fixed before the search begins. Instead, a dynamic variable ordering can make use of all information available in a partial solution to decide which variable to visit next. For example, a dynamic variable ordering allows a system to apply the variable ordering of Score-PMC even if the temporal structure is not known in the problem definition: the search always continues with the 'most-left' note in the score that is still undetermined.

After carefully analysing underlying problems of existing systems [AND 11], complemented by the experience from designing an earlier system [AND 00], the author concluded that a music constraint system, which should support a wide range of CSPs, requires a very flexible search algorithm. Users should be able to select search strategies suitable for their problems, or – if no suitable predefined strategy exists – define search strategies themselves, and the system should facilitate such definitions. Also, the search algorithm should allow for dynamic variable orderings. This paper demonstrates that music theory models, which where considered extremely complex for existing systems, become solvable in a reasonable amount of time when a suitable dynamic variable ordering is used.

Strasheela [1] [AND 07] is a highly generic music constraint system, where users can define a wide range of musical CSPs, including rhythmic, harmonic, melodic and contrapuntal problems. Strasheela comes with various examples that demonstrate these features, a selection of them is presented at the website of the software. Strasheela is based on a constraint programming model that supports dynamic variable orderings, and also allows users to freely define of such orderings. Strasheela is implemented in the programming language Oz [ROY 04], which provides build-in support for such a constraint programming model.

1. Strasheela is freely available at `http://strasheela.sourceforge.net/`.

Plan of Paper

The rest of this paper is organised as follows. Section 1.2 introduces a constraint model that supports both dynamic and user-defined variable orderings. Section 1.3 explains how this constraint model is customised to solve musical CSPs. Section 1.4 presents a number of special search strategies (more specifically, distribution strategies) for different musical CSPs. A contrapuntal CSP example is presented in Section 1.5. Note that both the pitch structure and the rhythmic structure are constrained in this problem, which makes it difficult or even impossible to solve in existing systems. The paper ends with a summary (Section 1.6).

1.2. Background: The Constraint Model Based on Computation Spaces

Strasheela is founded on a constraint programming model that introduces the notion of computation spaces [SCH 02], a programming construct for encapsulating speculative computations, where a solution is found by searching. This model makes the search process programmable at a high-level.

Two aspects of this constraint model are particularly apparent: (i) the model performs local inferences (constraint propagation) automatically, whereas (ii) speculative decisions during the search process (branching, distribution) are programmed by the user. This observation led to the slogan *propagate-and-search* [ROY 04] (also known as *propagate-and-distribute* [SCH 08]).

Propagate-and-search constitutes a general search approach, but for brevity the explanation here will focus on CSPs over variables whose domains consist of natural numbers (finite domain integers). For the same reason, this explanation is restricted to binary choice points (alternatives) in the search tree, although the approach supports n-ary choice points.

The space-based constraint programming model is only outlined in the present text. [SCH 02] describes in detail the notion of computation spaces and the consequences of their introduction. The programming textbook by [ROY 04] explains how this model is defined on top of more fundamental programming models such as declarative programming (i.e., functional and logic programming without the notion of search) and concurrent programming. [SCH 08] provide a tutorial on constraint programming with finite domain integers using this model, and [MÜL 08] a tutorial on finite set constraints. [DUC 98] apply this model to natural language processing and explain many aspects of the constraint model.

The following Section 1.2.1 will introduce the propagate-and-search approach. Section 1.2.2 explains propagate-and-search by an example. Finally, Section 1.2.3 explains how users define a search strategy (more specifically, how users define a

variable ordering – other aspects of the search such as how the resulting search tree is explored are left out for simplicity).

1.2.1. *Propagate and Search*

The propagate-and-search approach represents explicitly what is known about a CSP – at any stage during the search process. There are two types of information on variables: explicit information about the values of variables, and explicit information about the constraints applied to these variables.

A *computation space* encapsulates the information available at a certain stage during the search process. The information is expressed by (i) a constraint store, and (ii) a number of constraint propagators that are contained in a space (see Fig. 1.1). The *constraint store* stores partial information about the values of variables. A *propagator* represents a constraint on these variables. A computation space also contains a distributor (explained below). [2]

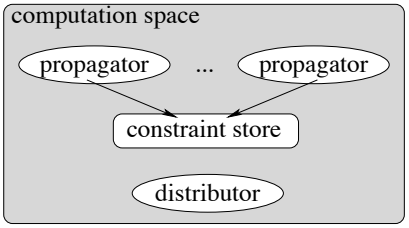


Figure 1.1: Propagate and search: a computation space encapsulates the information available on a CSP at a certain stage during the search process

The constraint store represents the domain information of all constrained variables, and also represents which variables are equal. The store has the form of a conjunction of basic constraints. A *basic constraint* is a piece of information that is restricted to very few simple forms.

In the case of finite domain constraints, there are two forms. $X \in \mathbf{D}$ denotes that $\mathbf{D}$ is the domain of the constrained variable X. In case the domain contains only a single value as in $X \in \{n\}$, then X is determined (i.e., $X = n$). The second form $X = Y$ expresses that two variables are equal (unified). Neither X nor Y must be determined here. An example of the information contained in a constraint store is the following conjunction.

2. A computation space contains further entities, which are not discussed in this introduction (e.g., a mutable store, cf. [ROY 04]).

$$X \in \{1, \ldots, 5\} \wedge Y = 7 \wedge Z = X$$

All constraints that don't fit into this form of basis constraints are *non-basic constraints*. Most constraints are non-basic constraints. Examples include $X < Y$ or 'all values in $[X_1, \ldots, X_n]$ are distinct'. Non-basic constraints are represented by propagators. [3] Each propagator implements an algorithm highly optimised for its particular constraint that strives to reduce the domain of the variables to which it is applied (and that way to reduce the size of the search space). These domain reductions are *inferred* from the current domains of these variables (and the constraint expressed by this propagator). For example, for a constraint store

$$X \in \{1, \ldots, 5\} \wedge Y \in \{1, \ldots, 5\}$$

the propagator $X < Y$ narrows the domains of the variables to which it is applied (X and Y) so that the constraint store is updated to

$$X \in \{1, \ldots, 4\} \wedge Y \in \{2, \ldots, 5\}$$

Each propagator is a software agent that runs in its own thread. Therefore, multiple propagators run in parallel and can reduce the domain of variables in parallel. Domain reductions of one propagation commonly trigger further reductions of another propagation.

Constraint propagation performs these inferences without actually searching (i.e., without performing any decision that may be wrong and requires undoing) and without excluding any solution. A propagator disappears when it is entailed by the constraint store (e.g., all its variables are determined). A space is *solved* when it contains no propagator (usually, all its variables are determined at that stage). A space is *failed* when it contains a failed propagator, that is a propagator that is inconsistent with the constraint store.

However, constraint propagation does not necessarily lead to a solution (or a fail). For example, for the store

3. Oz and Strasheela allow users to define complex constraints by wrapping sub-CSPs into an abstraction (procedure). Compositional rules are often defined as such abstractions in Strasheela. For simplicity we will refer to these abstractions also simply as constraints, although they are actually a network of constraints (propagators).

$$X \in \{1,2\} \wedge Y \in \{1,2\} \wedge Z \in \{1,2\}$$

the propagators $X \neq Y$, $X \neq Z$, and $Y \neq Z$ cannot reduce the problem further. Each propagator only 'sees' the variables, which it constrains, but it does not directly communicate with other propagators. When no further propagation is possible, the hosting computation space is said to be *stable*.

In this situation, constraint *distribution* makes a decision that results in two new computation spaces, which are easier to solve (see Fig. 1.2). A distributor is also a software agent, and runs concurrently with the propagators. It waits until its hosting computation space becomes stable. Then, the distributor creates two child spaces of the stable parent space. Both child spaces inherit all information available in the parent space (e.g., the constraint store and the propagators). Additionally, the distributor applies an arbitrary constraint C to any variable in one child space and its complement $\neg C$ (not C) to the corresponding variable in the other child space. Thus, the distributor executes a nondeterministic choice operation where C and $\neg C$ denote alternatives. For example, the constraint C may determine some variable to a certain value (e.g., to the first value in its domain), in which case $\neg C$ excludes this value from the variable domain.

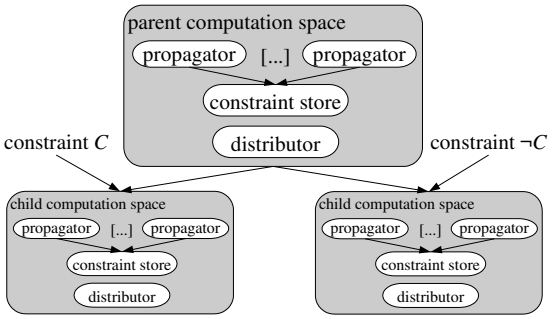


Figure 1.2: Constraint distribution creates two child spaces that are the result of two complementary decisions (expressed by the two added constraints C and $\neg C$)

Adding a constraint and its negation as two alternatives does not change the set of solutions. Either the constraint C leads to a solution – or it does not. It will, however, often restart propagation.

A *distribution strategy* defines the decision that is performed by the distributor. A common strategy is *first-fail*: this strategy determines some variable with the smallest domain to its leftmost domain value (and removes this value from the variable domain in the second child space). The following Section 1.2.2 explains this distribution strategy with an example.

The branching caused by constraint distribution results in a search tree. The example in the following Section also depicts the corresponding search tree in Fig. 1.4. This search tree is investigated by an exploration strategy for solutions (e.g., by the depth first search algorithm).

The combination of constraint propagation and constraint distribution (together with an exploration strategy) yields a complete method to solve a CSP.

1.2.2. *An Example*

The following paragraphs illustrate constraint propagation and distribution by computing an all-distance series. An all-distance series is a musical CSP very similar to an all-interval series, but its definition is more simple. Twelve-tone series (or tone rows) have been used by many composers (in particular in the second third of the 20th century) as a device to achieve coherency in music, even if the music abandons conventional harmony, by shaping a composition from only a single series and transformations of it (e.g., series transposition and inversion). Because a series plays such an important part for this music, composers design them with great care, and the notion of an all-interval series is proof for that.

Both the all-interval and the all-distance CSP define a series that consists of unique pitch classes and unique intervals between these pitch classes. The CSPs only differ in the way the pitch classes and the intervals are related. In an all-distance series, the intervals between the pitch classes are absolute distances (instead of inversionally equivalent intervals as in an all-interval series). For example, a fourth upwards and a fourth downwards are regarded as the same interval (instead of a fourth upwards and a fifth downwards in the case of an all-interval series).

Figure 1.3 shows the definition of the all-distance CSP of length 4 (i.e., the solution series consists of only 4 pitch classes). This short length results only an a small search tree, where the search for the first solution can be discussed in full detail. The definition uses common mathematical notation instead of Strasheela code to make it more easy to read.

Figure 1.4 shows the full search tree for all solutions of this CSP. Each tree node represents a computation space. Solved spaces are drawn as diamonds $\Diamond$, failed spaces as boxes $\Box$ and distributable spaces as circles $\circ$.

Only 17 nodes are necessary to find all 4 solutions – without propagation many more nodes would be necessary. The table below the tree shows the basic constraints on the elements of the solution series xs and the intervals dxs before their respective spaces get distributed.

/ Declaration of the solution series of pitch classes (xs) and intervals (dxs). */*

$xs \stackrel{def}{=}$ a list of 4 undetermined integers, each with the domain $\{0, \ldots, 3\}$

$dxs \stackrel{def}{=}$ a list of 3 undetermined integers, each with the domain $\{1, \ldots, 3\}$

/ Constraints between elements of xs and dxs. */*

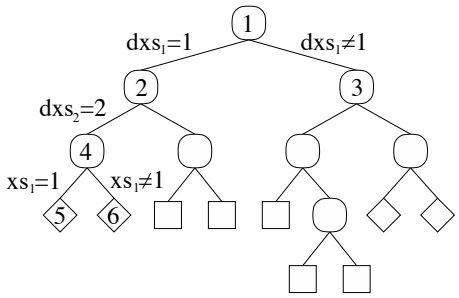
$$\bigwedge_{i=1}^{3} dxs_i = |xs_i - xs_{i+1}|$$

/ All elements in the solution xs as well as the intervals dxs between them are pairwise distinct. */*

$\wedge\ distinct(xs)$

$\wedge\ distinct(dxs)$

Figure 1.3: All-distance series definition of length 4



Space	xs	dxs
1	$[\{0,\ldots,3\},\ldots,\{0,\ldots,3\}]$	$[\{1,\ldots,3\},\{1,\ldots,3\},\{1,\ldots,3\}]$
2	[{0, …, 3}, …, {0, …, 3}]	[1, {2, 3}, {2, 3}]
3	[{0, …, 3}, …, {0, …, 3}]	[{2, 3}, {1, 2, 3}, {1, 2, 3}]
4	[{1, 2}, {1, 2}, {0, 3}, {0, 3}]	[1, 2, 3]
5	[1, 2, 0, 3]	[1, 2, 3]
6	[2, 1, 3, 0]	[1, 2, 3]

Figure 1.4: All-distance series: search tree for all solutions of length 4

The constraints added by the first-fail distribution strategy are shown next to the tree arcs. The strategy always affects some variable with minimal domain size (in the case where multiple variables share the same minimal domain size, the leftmost variable is chosen). In the first space it creates a choice point, which either binds the first element of dxs to 1 (in space 2) or removes this value from the domain (in space 3).

After each distribution step propagators adjust the domains of all variables accordingly. In space 2 the propagator constraining all intervals to be distinct removes 1 from the other two interval domains in dxs. In space 4 the propagator determines the third interval to be 3, which awakens the distance propagator to reduce the domain of last two series elements in xs to $\{0, 3\}$. This again reduces the domain of the first two series elements as well, because all elements are constrained to be different.

1.2.3. *Distribution Strategy Definition*

As mentioned above, the propagate and search approach allows the user to program how the search process is conducted. The distribution strategy defines the shape and content of the search tree. For example, it controls how many nodes exist and what constraints are added at each branching of the search tree. The user can define simple distribution strategies 'from scratch' with a few lines of code (cf. [SCH 02], p. 37). However, a high-level interface can make the definition of distribution strategies more convenient. [4]

Essentially, distribution strategies differ in two aspects: (i) in which order are variables visited and (ii) which domain value is selected by the distribution strategy. The high-level interface presented here expects a formal specification for these two aspects.

Order: Which variable is distributed? This aspect is specified by a binary boolean function. Internally, the distributor uses this function to compare pairs of variables: the variable that performs best with respect to this order relation is distributed (ties are broken by choosing the first of multiple variables that perform equally well). For example, the first-fail strategy distributes some variable with the smallest domain size. This variable ordering is specified as follows:

$$myOrder(X, Y) \stackrel{def}{=} getDomainSize(X) \leq getDomainSize(Y)$$

Value: How does the distribution strategy affect the domain of the selected variable? This aspect is specified by a unary function expecting a variable and returning a reduced domain specification for this variable (most often a single value). For example, the first-fail strategy decides for the minimal domain value:

$$myValue(X) \stackrel{def}{=} getMinimalDomainValue(X)$$

4. This section presents an interface that is a distinctly simplified version of the Oz procedure `FD.distribute` [DUC 08], which is the standard means for solving finite domain integer CSPs in Oz. For an implementation of this interface in Oz see [AND 07].

A suitable distribution strategy results in a relatively small search tree. This requires careful design. Any distribution step makes a decision, which possibly leads to a fail that can prolong the search process. Constraint propagation, on the other hand, is never wrong when it reduces variable domains. This observation leads to an important rule-of-thumb for designing distribution strategies: a good strategy keeps the number of required decisions at a minimum and distributes in such a way that propagation does most of the work.

It depends on the actual CSP as to which distribution is suitable (e.g., efficient) for this problem. A variable ordering that follows the well-proven *first-fail* principle first determines those variables, which are particularly hard to solve [BAR 98]. The previously discussed first-fail strategy first distributes some variable with the smallest domain. An alternative approach distributes some variable to which most constraints are applied (both aspects can also be combined, c.f. [BEE 06, p. 106 ff]).

Common decisions for a particular domain value in addition to the domain minimum include the median or the maximum. Alternatively, the domain of a variable can be reduced to its lower half instead of deciding on a specific domain value (this technique is sometimes called domain splitting). A well-proven rule of thumb for value orderings is the *succeed-first* principle, which decides on a promising domain value first [BAR 98].

A distribution strategy can define heuristics to place a better solution earlier in the search tree. Such a *best-first* principle can be interpreted as a variation of the succeed-first principle. A typical heuristic for a musical CSP avoids repetition and uniformness. For many musical CSPs it is appropriate to randomise the decision for a domain value. [5]

Please note that a distribution strategy only decides which constraint to add when this decision is actually required during the search process (dynamic variable and value ordering). The distribution can therefore make use of all the information available at the current state of the search. In contrast, most existing music constraint systems apply search strategies in which the order of decisions is determined before the search starts (static variable ordering). Section 1.3 will discuss the consequences of static versus dynamic variable orderings for musical CSPs.

5. It is important that a distribution strategy is deterministic, that is, if executed multiple times a distribution strategy must make the same decision in each computation space. This is particularly important for large CSPs, which require recomputation, a technique that trades memory for computation time [SCH 02]. A distribution strategy can be seemingly random but still deterministic when based on pseudo-random values.

1.3. Specialising the Constraint Model for Music

The present section first briefly introduces the music representation of Strasheela (Section 1.3.1), and then explains how Strasheela customises the space-based constraint model for a generic music constraint system (Section 1.4).

1.3.1. *Music Representation and Score Contexts*

As mentioned before, Strasheela supports a wide range of musical constraints, including constraints on the rhythm, harmony, contrapuntal structure and the musical form.

In order to facilitate such a range of CSPs, Strasheela features a comprehensive symbolic music representation. The representation predefines various musical concepts important for many musical CSPs. For example, concepts such as notes, rests, voices and polyphonic staffs are modelled. Such score objects include parameters and the values of these parameters are variables that can be constrained (e.g., the start time, duration or pitch of a note can be constrained). Also, the Strasheela music representation supports extensive analytical information (e.g., information on the underlying harmony or motivic relations). Again, objects modelling such information contain parameters whose values can be constrained (e.g., the root, type and inversion of a chord object can be constrained). Further, the relation between parameters – either within the same object or parameters of multiple objects – is specified by constraints (e.g., a harmonic constraint can define the relation between the pitch class set of a chord object and the pitch classes of simultaneous notes). Most importantly, the representation can be extended by users. For example, all entities are defined by classes in the object-oriented programming sense, and users can extend these classes by inheritance. Strasheela's music representation is explain in detail in [AND 07].

Strasheela is very flexible with regard to which variable sets of the music representation can be constrained. Musical CSPs often apply constraints to multiple sets of variables. For example, a melodic constraint may constrain the pitches of all consecutive note pairs. This paper refers to sets of variables or score objects that are interrelated in the same way as instances of the same *score context*.[6] Examples include the sets of consecutive note pairs in a melody, sets of simultaneous notes in a score, sets of intervals between simultaneous notes, sets of all score objects that belong to a single bar and so forth. Arbitrary contexts can be constrained in Strasheela: the system supports a constraint application technique based on functional programming that combines generality with convenience [AND 10].

6. The term score context is inspired by Lilypond [NIE 03] and PWConstraints [LAU 96], however the term is redefined here with a much more general meaning.

However, this flexibility introduces a potential problem for solving CSPs efficiently: score contexts can be inaccessible in the CSP definition. For example, the sets of simultaneous notes is inaccessible if the rhythmical structure is unknown. As long as contexts are inaccessible, constraints on these contexts propagate very weakly – if at all. If simultaneous notes are constrained by a harmonic constraint then this constraint can only propagate for notes that are known to be simultaneous. As long as the rhythmical structure for the notes in question is undetermined – and hence the context 'simultaneous notes' is inaccessible – propagation blocks.

A distribution strategy for a CSP that constrains inaccessible score contexts should therefore take special care to resolve these contexts at an early stage in order to make propagation possible. In analogy to the *first-fail* principle for variable orderings (Section 1.2.3), the present research calls this guideline the *resolve-inaccessible-contexts* principle. For instance, in many musical CSPs the temporal structure should be determined relatively early in case-dependent contexts (for example where simultaneous notes are inaccessible but constrained).

1.3.2. *Adapting the Space-Based Constraint Model*

The score distribution strategies introduced below often make use of score information available at the time a distribution decision is due. In order to make arbitrary score information accessible for score distributors, Strasheela adapts the space-based constraint model for musical CSPs.

While the space-based constraint model originally distributes plain variables, Strasheela distributors distribute parameter objects (the value attribute of a parameter object is a variable). The hierarchic Strasheela music representation defines bi-directional links between all nesting relations in the hierarchy. For example, the pitch parameter of a note can access the note to which it belongs (see Figure 1.5), and the note can access its container (e.g., its voice). By distributing parameter objects instead of plain variables and because of the bi-directional links between score objects, a Strasheela distributor can freely traverse a score hierarchy to collect information for its decision.

To make the discussion of score distribution strategies more concise, the high-level interface for the definition of distribution strategies introduced in Section 1.2.3 has been adapted for score distribution strategies. The original interface expects two functions: a binary *order* function, which defines the variable ordering and a unary *value* function, which defines how the domain of the distributed variable is reduced. In the adapted interface, the *order* function expects two parameter instances, but the *value* function remains the same. [7]

7. For an implementation of this interface in Oz see [AND 07].

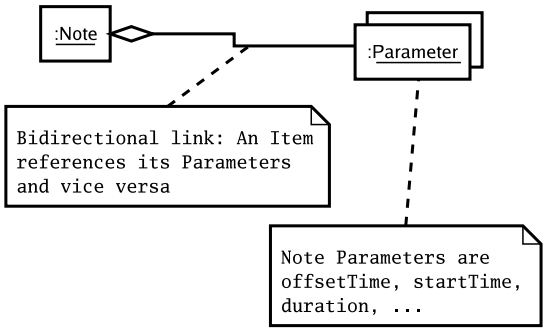


Figure 1.5: A single note object and its parameters (e.g., start time, duration and pitch) are bidirectionally linked (UML notation); such bidirectional links are featured by all other hierarchic relations in the music representation as well

1.4. Score Distribution Strategies

This section proposes several score distribution strategies that together cover a large number of musical CSPs. Further strategies can be defined by users according to needs.

1.4.1. *Adapting the First-Fail Principle*

Section 1.2.3 discussed the first-fail principle as a rule-of-thumb used to formulate distribution strategies. This guideline is also well-suited to many musical CSPs. Typical strategy examples either distribute a variable with the smallest domain or alternatively a variable on which most constraints are applied.

The definition of the first-fail variable ordering function for parameter objects differs only slightly from the ordering function for plain variables. The order expressed by the function $isParamWithSmallerDomainSize$ (Fig. 1.6) results in a variable ordering that first distributes the value of some parameter with the smallest domain size. The only difference to the first-fail strategy for plain variables presented above is that the version for parameter instances has to access the parameter values by $getValue$.

$$isParamWithSmallerDomainSize(param_1, param_2) \stackrel{def}{=}$$
$$getDomainSize(getValue(param_1)) \leq getDomainSize(getValue(param_2))$$

Figure 1.6: Variable ordering specification of first-fail strategy for parameters

A full distribution strategy specification complements the variable ordering with a decision as to how the variable domain is reduced (the term value ordering denotes a slightly less general concept but will still be used here for brevity). Commonly, the first-fail strategy decides on the smallest domain value. For musical CSPs it is commonly desired to avoid (too much) repetition, and therefore a heuristics that randomly selects a domain value is often preferable.

Order: $isParamWithSmallerDomainSize$
Value: For example $getMinimalDomainValue$ or $getRandomDomainValue$

In the remainder of this chapter, several variable orderings suitable for various musical CSPs are presented. All these variable orderings can be complemented by any value ordering.

1.4.2. *Resolving Inaccessible Score Contexts*

The general first-fail principle is primarily suitable for CSPs in which all constrained score contexts (Section 1.3.1) are already accessible in the problem definition. Otherwise, inaccessible contexts that are constrained should be resolved early to make the propagation of these constraints possible (resolve-inaccessible-contexts principle, see above).

A typical example of an inaccessible but constrained score context is the context of simultaneous score objects in case the temporal structure of the music is undetermined in the problem definition. For example, Figure 1.7 shows several notes with unknown start time, duration and pitch. Because the temporal parameters of these notes are undetermined, their context of simultaneous notes is inaccessible. One variable ordering that addresses this inaccessible context determines all temporal parameters first and other parameters only later.

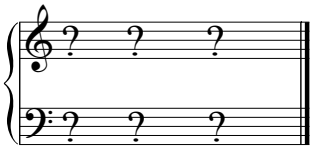


Figure 1.7: Several notes with unknown start time, duration and pitch (common music notation does not support such unknown information and question marks are used for notating these notes). The context of simultaneous notes (i.e., the set of the notes that are simultaneous) is inaccessible until enough temporal parameters are determined

It is sufficient to test whether the first of two parameter objects is a temporal parameter in order to express a variable ordering that determines temporal parameters

first (Fig. 1.8). In the case where $isTemporalParameter$ [8] returns $true$ for the first parameter, then the ordering function $preferTemporalParameter$ returns $true$ and this parameter will be distributed. Alternatively, if *preferTemporalParameter* returns $false$ then this expresses a preference for $param_2$ – whether it actually is a temporal parameter or not. In the case where $param_2$ is not a temporal parameter, then this preference will be overruled as long as there is still any undetermined temporal parameter. In the case where there is no undetermined temporal parameter left, it is adequate to distribute a parameter that is not a temporal parameter.

$$preferTemporalParameter(param_1, param_2) \stackrel{def}{=} isTemporalParameter(param_1)$$

Figure 1.8: Variable ordering specification that determines all temporal parameters first in order to make score contexts based on the temporal structure (e.g., simultaneous notes) accessible

A variable ordering that first determines the temporal structure of the score is only a special case for variable orderings that first resolve some inaccessible but constrained score context. The resolve-inaccessible-contexts principle is applicable to any inaccessible score context. For instance, a harmonic CSP may explicitly represent analytical harmonic information (e.g., chords denoted by Roman numerals) and constrain this harmonic information as well as the actual note pitches (which depend on the harmonic information). For such CSPs, it is usually suitable to distribute the harmonic information first before distributing the actual note pitches.

1.4.3. *Resolving Multiple Inaccessible Score Contexts in Order*

The preceding subsection addressed how a constrained but inaccessible score context can be resolved at an early stage of the search process. Doing so facilitates constraint propagation. Such a variable ordering approach can be generalised for multiple constrained but inaccessible score contexts. During the search process, these contexts are then determined in a certain order. For example, such a variable ordering can be applied to solve a harmonic CSP, which explicitly represents analytic harmonic information and constrains this harmonic structure, the actual note pitches, and the rhythmical structure at the same time. A distribution strategy for a such a CSP may first determine the temporal parameters of the chords and notes to resolve contexts that depend on the rhythmical structure. Then it may determine all chord parameters and finally the actual note pitches.

---

8. $isTemporalParameter$ tests class membership of parameter objects in the object-oriented programming sense.

While this variable ordering can be suitable for harmonic CSPs, other CSPs require their own ordering. This depends on the CSP. Therefore, a generic means is desirable to easily define variable orderings suitable for a given CSP.

The function $determineInOrder$ defines such a generic means. It expects a specification stating the parameters affected and the order in which they are determined. This specification is given as a list of unary boolean functions – the order of the test functions specifies the order in which the corresponding parameters are determined. For example, Fig. 1.9 defines the variable ordering for the harmonic CSP proposed above. This ordering first determines all temporal parameters, then all chord parameters, and finally all note pitches.

$$\begin{aligned} determineInOrder([&isTemporalParameter,\\ &isChordParameter,\\ &isPitch]) \end{aligned}$$

Figure 1.9: A variable ordering for harmonic CSPs that first determines the temporal structure of the score, then the harmonic structure and finally the actual pitches

The function $determineInOrder$ is defined in Fig. 1.10. The function expects a list of test functions and returns a variable ordering that compares two parameter objects and decides accordingly. This definition uses the functional programming paradigm [ABE 85]: it expects functions as arguments and it creates and returns a function. The functions f and g are defined using *where-notation* [LAN 66]: this notation uses conventional function notation for effectively defining anonymous functions. The colon (:) is read as *where*.

$$\begin{aligned} &determineInOrder(tests) \stackrel{def}{=}\\ &\quad \textbf{let} \ \textit{/* Append a default test function that always returns } true. \ \textit{*/}\\ &\qquad allTests \stackrel{def}{=} append(tests, [f : f(x) \stackrel{def}{=} true])\\ &\quad \textbf{in} \ \ g : g(param_1, param_2) \stackrel{def}{=}\\ &\qquad\quad getTestIndex(param_1, allTests) \leq getTestIndex(param_2, allTests) \end{aligned}$$

Figure 1.10: The function $determineInOrder$ returns a variable ordering function that determines parameters in the order specified by a list of test functions

For its decision, $determineInOrder$ makes use of the function $getTestIndex$. This function expects a value and a list of Boolean functions, and it returns the position of

the first function that returns $true$ for the given value.[9] The variable ordering function created by $determineInOrder$ calls $getTestIndex$ with both its parameters and decides on the parameter for which the smaller index is returned (in case $getTestIndex$ returns equal values for two parameters, the distribution opts for the first parameter).

1.4.4. *Combining the Principles Resolve-Inaccessible-Contexts and First-Fail*

The variable orderings discussed above distribute selected parameters (i.e., parameters for which some test function such as $isTemporalParameter$ returns $true$) in an arbitrary order. For example, the variable ordering that first determines the temporal structure (Fig. 1.8) does not express any preference as to which temporal parameter to distribute in case there are multiple undetermined temporal parameters (or which non-temporal parameter to distribute in case all remaining undetermined parameters are non-temporal parameters).

For many CSPs, a more suitable strategy combines the resolve-inaccessible-contexts principle with the general first-fail principle. For example, instead of determining all temporal parameters (or non-temporal parameters) in no particular order, some temporal parameter with smallest domain size is distributed first. Figure 1.11 defines a variable ordering that first tests whether one of two parameters is a temporal parameter, and then decides for the temporal parameter. If either both or none of the parameters is a temporal parameter, then the parameter with smaller domain size is distributed. The function $isParamWithSmallerDomainSize$ is defined in Fig. 1.6.

$$
\begin{array}{l}
preferTemporalParamOrParamWithSmallerDomainSize(param_1, param_2) \stackrel{def}{=} \\
\quad \textbf{let} \;\; b \stackrel{def}{=} isTemporalParameter(param_1) \\
\quad \textbf{in} \;\; \textbf{if} \;\; xor(b, isTemporalParameter(param_2)) \\
\qquad \textbf{then} \;\; b \\
\qquad \textbf{else} \;\; isParamWithSmallerDomainSize(param_1, param_2)
\end{array}
$$

Figure 1.11: Variable ordering specification that in general determines all temporal parameters before the non-temporal parameters, but also determines temporal (or non-temporal) parameters with smallest domain size first.

The first-fail principle can be applied in a similar way when multiple contexts are resolved in a certain order. In case multiple parameter objects are of the same 'rank', some parameter with the smallest domain (or most applied constraints) is distributed first.

9. The function $getTestIndex$ is defined in [AND 07].

1.4.5. *The Dynamic Left-to-Right Variable Ordering*

For many musical CSPs, in particular contrapuntal CSPs, a suitable distribution strategy determines parameter objects 'from left to right'. Such an approach distributes parameters in the order of the start times of the score objects to which these parameters belong.

Contrapuntal CSPs commonly apply constraints between 2 or more consecutive notes of the same voice (e.g., melodic constraints), as well as constraints between 2 or more simultaneous notes from different voices (e.g., harmonic constraints). By contrast, less common are constraints between notes of the same voice that are not directly following each other, as well as constraints of different voices that are not simultaneous.

For an efficient search it is important to detect violations of these common constraints as early as possible. A variable ordering that completes individual voices in the order of their notes, and simultaneous voices more or less in parallel does exactly that. Further, a variable ordering that proceeds 'from left to right' (i.e., starting at the beginning of a score) instead of 'from right to left' (starting at the end) is better for those CSPs where we do not know the end time of the score, which is commonly the case, Commonly, we already know the start time of the first note of each voice – or their domain is rather small, but the domain of the end time of the last note of each voice is rather large. Starting at the beginning therefore follows the first-fail principle.

The resulting variable ordering is very similar to the variable ordering applied by Score-PMC [LAU 96]. [10] However, Score-PMC estimates a static variable ordering before the search starts and therefore requires that the rhythmical structure of the music is fully determined in the problem definition. A distribution strategy, on the other hand, allows for a dynamic variable ordering. A distribution strategy can be defined in such a way that it proceeds 'from left to right' whether the rhythmical structure of the music is determined or not.

Figure 1.12 defines a dynamic left-to-right variable ordering. The main criteria for the variable ordering are the start times of the score objects to which the parameter objects in question belong. The function $isLeftmostParam$ first accesses the parameters' associated start times and checks whether these start times are already determined (i.e whether $getDomainSize$ returns 1) and then bases its decision mainly on

10. The variable ordering of Score-PMC takes not only the start time but also the duration of notes into account. In case two notes share the same start time, then the longer note is visited first.

The distribution strategy presented here ignores the note durations. Still, this strategy can be extended accordingly.

the values of these start times. In case only one of these two start times is already determined, the respective parameter is preferred (else-clause of outer if-expression). [11] In case both start times are determined, $isLeftmostParam$ opts for the parameter to which the smaller start time belongs (else-clause of inner if-expression). Finally, in case both start times are equal then $isLeftmostParam$ prefers a temporal parameter. This condition causes an early determination of temporal parameters.

$isLeftmostParam(param_1, param_2) \stackrel{def}{=}$

let $start_1 \stackrel{def}{=} getStartTime(getItem(param_1))$
$start_2 \stackrel{def}{=} getStartTime(getItem(param_2))$
$isStart_1Bound \stackrel{def}{=} (getDomainSize(start_1) = 1)$
in **if** $isStart_1Bound \wedge (getDomainSize(start_2) = 1)$
then **if** $start_1 = start_2$
then /* If the items of both parameters start at the same time, then prefer a temporal parameter (e.g., determining a duration may propagate and other item start times get determined). */
$isTemporalParameter(param_1)$
else /* Prefer the parameter whose item has a smaller start time */
$start_1 \leq start_2$
else /* If only one parameter has a determine start time, then prefer that parameter. */
$isStart_1Bound$

Figure 1.12: A left-to-right dynamic variable ordering

The dynamic left-to-right variable ordering presented in this section turns out to be highly suitable for a wide range of musical CSPs. In particular, it allows for solving contrapuntal problems that are very hard or even impossible to solve using existing systems. In addition, this variable ordering is applicable for arbitrarily nested score topologies and therefore is a good candidate for a default variable ordering for musical CSPs in general. Section 1.5.2 will compare the efficiency of this strategy with other strategies for a specific contrapuntal CSP.

1.5. An Example: Florid Counterpoint

This example demonstrates Strasheela's capabilities for contrapuntal CSPs where both the pitch structure as well as the rhythmical structure is constrained by rules.

11. In case no start time is determined, $isLeftmostParam$ decides for $param_2$. The distribution strategy assumes that the start time of the leftmost item in the score is always determined before the search starts. Otherwise, the search would start with an arbitrary parameter.

Previous systems either hardly support contrapuntal CSPs at all or require that the temporal structure is determined in the problem definition (cf. Score-PMC, Section 1.1).

This example was designed to be relatively simple. Therefore, it compiles rules from various sources instead of following a specific author closely. For example, several rules are variants of Fuxian rules [FUX 25], but rhythmical rules were inspired by [MOT 81]. Accordingly, the result does also not imitate a particular historical style (but neither does Fux, cf. [JEP 30]).

1.5.1. *The Music Theory Model*

The music representation for this example consists of two parallel voices ($voice_1$ and $voice_2$). Each voice is represented by a *sequential container* – a Strasheela score object that implicitly constrains contained objects (in this case note objects) to form a temporal sequence. In turn, both voices (i.e., both sequential containers) are nested in a *simultaneous container*, which implicitly constrains these voices to run in parallel. In this specific example, $voice_1$ contains 17 notes and $voice_2$ 15 notes.

The start time and end time of each voice is further restricted. $voice_1$ begins one bar before $voice_2$. This is expressed by setting the offset time of these two sequential containers with respect to their surrounding simultaneous container to different values: the offset of $voice_1$ is 0, and the offset of $voice_2$ is a semibreve, represented by the duration value 16. [12] Both voices end at the same time (the end time of both sequential containers is constrained to be the same value).

All note pitches and also all note durations are searched for. In this specific example, each note duration has the domain $\{quaver, crotchet, minim\}$. The pitch domain for each note in $voice_1$ is set to $\{f3, \ldots, g4\}$, [13] the domain for the note pitches of $voice_2$ is slightly greater ($\{f3, \ldots, c4\}$).

The example defines constraints for various aspects of the music. The example applies rhythmic constraints, melodic constraints, harmonic constraints, contrapuntal constraints and constraints concerning the formal structure.

Rhythmic Constraints:

– Each voice starts and ends with a minim note value.

12. The actual meaning of any Strasheela parameter is defined by its unit of measurement. In this example, all temporal parameters are measured in beats, where each beat corresponds to a crochet of duration 4.

13. The unit of measurement of all pitch parameters in this example is MIDI note number.

– Note durations may only change slowly across a voice: neighbouring note values are either of equal length or differ by 50 percent at maximum (e.g., a quaver can be followed by a crotchet, but not by a minim).

– The last note of each voice must start with a full bar.

Melodic Constraints:

– Each note pitch is restricted to the diatonic pitches of the C-major scale.

– The first and last note of $voice_1$ must start and end with the root c.

– The melodic intervals between consecutive notes in a voice are limited to a minor third at maximum (i.e., the melodic intervals are more restricted here than in Fuxian counterpoint).

– An important rule constrains melodic peaks: the maximum and minimum pitch in a phrase occurs exactly once and it is not the first or last note of the phrase. In this example, a phrase is defined simply as half a melody. Also, the pitch maxima and minima of phrases must differ. This rule concerning the melodic contour – inspired by Schoenberg – has great influence on the musical quality of the result (subjectively evaluated) but also on the combinatorial complexity of the CSP.

Harmonic Constraints:

– Simultaneous notes must be consonant.

– The only exception permitted here are passing tones, where $note_1$ is a passing tone (i.e., the intervals between the note and its predecessor as well as between the note and its successor are steps and both steps occur in the same direction) where a simultaneous $note_2$ started before $note_1$, and this $note_2$ is consonant with the predecessor of $note_1$.

Contrapuntal Constraint:

– Open parallel fifths and octaves are not allowed. However, hidden parallels are unaffected here – in contrast to the Fuxian counterpoint.

Form Constraint:

– Both voices form a canon at the fifth: the first n notes of both voices form (transposed) equivalents. In the case here, $n = 10$.

1.5.2. *Search Process and Results*

This section compares the performance of different distribution strategies for the example and presents a solution.

Note that this example constrains a score context that cannot be accessed in the problem definition. The problem definition does not provide enough information to

access simultaneous notes in the music representation. Nevertheless, the relation between simultaneous notes is constrained by the harmonic and contrapuntal constraints.

The distribution strategy should therefore determine this score context at an early stage to support propagation of the constraints applied to this context (Section 1.3.1). The left-to-right score distribution strategy (Section 1.4.5) was applied. This distribution strategy found the first solution in about 4 seconds (189 distributable spaces, 175 failed spaces, search tree depth 47). [14]

For comparison, a distribution strategy that first fully determines the temporal structure and then searches for the pitches (Section 1.4.2) was applied as well. To determine the temporal structure, the distribution only needs to determine the note duration values. Note offset time values are already determined in the problem definition (all 0). Start time and end time values are determined by propagation once the durations are known. This distribution strategy did not find any solution within an hour (i.e., not even within about 900 times more time than needed by the left-to-right strategy)! After that time period, the search process was interrupted. [15]

The distribution strategy that first fully determines the temporal structure finds solutions for the rhythmical structure, which indeed fulfil all rhythmic constrains, but which are in conflict with constrains concerning the pitch structure. However, these conflicts are detected very late – this causes much redundant work, and slows down the search by orders of magnitude.

An analysis of the CSP revealed that the complexity of the problem was greatly reduced when the rule that demands unique melodic minima and maxima was left out. With such a reduced CSP, a solution can be found with both distribution strategies in a reasonable amount of time.

The left-to-right strategy clearly outperforms the other strategy on this reduced CSP as well. To find the first solution for the reduced CSP, the left-to-right strategy required about 1.7 seconds (92 distributable spaces, 70 failed spaces, search tree depth 53) and the strategy that first fully determines the temporal structure required about 14 seconds (630 distributable spaces, 601 failed spaces, search tree depth 62). The left-to-right strategy is thus almost ten times faster than the other strategy for this simplified example.

---

14. All performance measurements are the rounded average of 10 runs and were conducted on a Pentium 4, 3.2 GHz machine with 512 MB RAM (Fujitsu Siemens Scenic P320 i915G), running Linux with kernel 2.6.12 (Fedora Core 3) and Mozart 1.3.1.

15. For this CSP, the first-fail distribution strategy (Section 1.4.1) is equivalent to a distribution strategy that first fully determines temporal structure. In this CSP, the duration parameters have the smallest domain (only three duration domain values vs. at least 14 pitch domain values) and are therefore determined first by first-fail.

These figures demonstrate the great importance of a suitable search strategy for a musical CSP. This requirement is particularly pressing for complex problems that constrain inaccessible score contexts.

The figures indicate the suitability of the left-to-right distribution strategy for contrapuntal CSPs. The importance of user-definable distribution strategies is underlined by these figures: different CSPs require different distribution strategies and a system designer cannot foresee every CSP defined by a user.

Figure 1.13 shows one solution for this CSP. $voice_1$ is depicted as the lower voice. A randomised value ordering has been used (whose search process was comparably efficient to the mid-domain-value ordering used for the measurements).

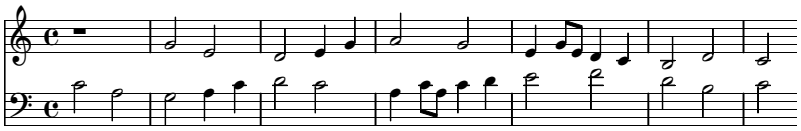


Figure 1.13: One solution of the florid counterpoint CSP (using random value ordering), $voice_1$ is depicted in the lower staff

In conclusion, it should be mentioned again that previous systems such as PWConstraints and Situation do not support musical CSPs like the one presented in this section due to their computational complexity. The dynamic left-to-right distribution strategy, proposed by the present research, solves problems like this in a few seconds and thus in a reasonable amount of time for practical use.

1.6. Summary

This paper presented the search process of a highly generic music constraint system, called Strasheela. Strasheela allows its users to define a large set of musical CSPs, including rhythmic, harmonic, melodic and contrapuntal problems, and to solve them in a reasonably efficient way.

Strasheela supports solving its wide range of musical CSPs efficiently by making the search process adaptable to these problems. Strasheela employs the space-based constraint model, because this model makes the search process programmable at a high-level (Section 1.2).

A particular important problem-dependent aspect of the search process is the variable ordering. The effect of different variable orderings on the size of the search space for a CSP – and thus on the efficiency of the search process – is well known from the constraint programming literature, as is the fact that different CSPs perform best with

different variable orderings. This paper argues that musical CSPs often require variable orderings custom-made for musical CSPs in order to be solved efficiently. This is particularly true for complex CSPs that constrain at the same time various musical aspects such as the rhythmic structure, melodic structure, harmonic structure, contrapuntal structure and musical form. By defining a distribution strategy, Strasheela users define a dynamic variable and value ordering that is independent of the CSP definition.

Strasheela customises the space-based constraint model for music constraint programming: Strasheela allows a score distribution strategy to exploit any information available in the score whenever it performs a distribution step (Section 1.3). Strasheela preserves all other features of the model such as constraint propagation, user-defined exploration strategies, reified constraints, and recomputation [SCH 02].

This research presented special score distribution strategies suitable for a large range of musical CSPs (Section 1.4). For example, the text revises the left-to-right search strategy of Score-PMC such that the revised strategy efficiently solves contrapuntal CSPs including problems that Score-PMC explicitly made impossible to define due to their computational complexity. The effect of this distribution strategy is also demonstrated in a musical example (Section 1.5).

1.7. Bibliography

[ABE 85] ABELSON H., SUSSMAN G. J., SUSSMAN J., *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.

[AND 00] ANDERS T., "Arno: Constraints Programming in Common Music", *Proceedings of the 2000 International Computer Music Conference*, San Francisco, International Computer Music Association, 2000.

[AND 07] ANDERS T., Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System, PhD thesis, School of Music & Sonic Arts, Queen's University Belfast, 2007.

[AND 10] ANDERS T., MIRANDA E. R., "Constraint Application with Higher-Order Programming for Modeling Music Theories", *Computer Music Journal*, vol. 34, num. 2, 2010.

[AND 11] ANDERS T., MIRANDA E. R., "A Survey of Constraint Programming Systems for Modelling Music Theories and Composition", *ACM Computing Surveys*, vol. 43, num. 4, 2011.

[BAR 98] BARTAK R., "On-Line Guide to Constraints Programming", `http://kti.mff.cuni.cz/~bartak/constraints/` (accessed 4 January 2011), 1998.

[BEE 06] VAN BEEK P., "Backtracking Search Algorithms", ROSSI F., VAN BEEK P., WALSH T., Eds., *Handbook of Constraint Programming*, Elsevier B.V., Amsterdam, 2006.

[BON 99] BONNET A., RUEDA C., OpenMusic. Situation. version 3, IRCAM, Paris, 3rd edition, 1999.

[CHE 01] CHEMILLIER M., TRUCHET C., "Two Musical CSPs", *Seventh International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*, Paphos, Cyprus, 2001, available at `http://recherche.ircam.fr/equipes/repmus/cpws/chemillier.ps` (accessed 6 January 2010).

[DEC 03] DECHTER R., *Constraint Processing*, Morgan Kaufmann, San Francisco, CA, 2003.

[DUC 98] DUCHIER D., GARDENT C., NIEHREN J., "Concurrent Constraint Programming in Oz for Natural Language Processing", `http://www.ps.uni-saarland.de/~niehren/Web/Vorlesungen/Oz-NL-SS01/vorlesung/` (accessed 4 January 2011), 1998.

[DUC 08] DUCHIER D., KORNSTAEDT L., HOMIK M., MÜLLER T., SCHULTE C., ROY P. V., Mozart Documentation. System Modules, Mozart Consortium, 2008, `http://www.mozart-oz.org/documentation/system/index.html` (accessed 4 January 2011).

[EBC 92] EBCIOGLU K., "An Expert System for Harmonizing Chorales in the Style of J.S. Bach", BALABAN M., EBCIOGLU K., LASKE O., Eds., *Understanding Music with AI: Perspectives on Music Cognition*, Chapter 12, p. 295–332, MIT Press, Cambridge, MA, 1992.

[FUX 25] FUX J. J., *The Study of Counterpoint. from Johann Joseph Fux's Gradus ad Parnassum*, W.W. Norton & Company, London, 1965, orig. 1725, translated and edited by Alfred Mann.

[JEP 30] JEPPESEN K., *Kontrapunkt*, Breitkopf & Härtel, Leipzig, 4th edition, 1971, orig. 1930.

[KUM 92] KUMAR V., "Algorithms for Constraints Satisfication Problems: A Survey", *AI Magazine*, vol. 13, num. 1, 1992.

[LAN 66] LANDIN P. J., "The Next 700 Programming Languages", *Communications of the ACM*, vol. 9, num. 3, p. 157–166, 1966.

[LAU 96] LAURSON M., PATCHWORK: A Visual Programming Language and some Musical Applications, PhD thesis, Sibelius Academy, Helsinki, 1996.

[LAU 01] LAURSON M., KUUSKANKARE M., "A Constraint Based Approach to Musical Textures and Instrumental Writing", *Seventh International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*, Paphos, Cyprus, 2001, available at `http://recherche.ircam.fr/equipes/repmus/cpws/laurson.ps` (accessed 6 January 2010).

[MOT 81] MOTTE D. D. L., *Kontrapunkt*, Bärenreiter-Verlag, Kassel/Basel, 1981.

[MÜL 08] MÜLLER T., Problem Solving with Finite Set Constraints in Oz. A Tutorial, Mozart Consortium, 2008, `http://www.mozart-oz.org/documentation/fst` (accessed 4 January 2011).

[NIE 03] NIENHUYS H.-W., NIEUWENHUIZEN J., "Lilypond, a System for Automated Music Engraving", *Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*, Firenze, Italy, 2003, available at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.6160&rep=rep1&type=pdf` (accessed 6 Januaray 2010).

[PAC 01] PACHET F., ROY P., "Musical Harmonization with Constraints: A Survey", *Constraints Journal*, vol. 6, num. 1, p. 7–19, 2001.

[ROS 06] ROSSI F., BEEK P. V., WALSH T., *Handbook of Constraint Programming*, Elsevier, Amsterdam, 2006.

[ROY 04] VAN ROY P., HARIDI S., *Concepts, Techniques, and Models of Computer Programming*, MIT Press, 2004.

[RUE 98] RUEDA C., LINDBERG M., LAURSON M., BLOCK G., ASSAYAG G., "Integrating Constraint Programming in Visual Musical Composition Languages", *ECAI 98 Workshop on Constraints for Artistic Applications*, Brighton, 1998, available at `http://www.cs.vu.nl/~eliens/poosd/@online/@share/archive/ecai98/rueda.ps` (accessed 7 January 2010).

[SAN 03] SANDRED O., "Searching for a Rhythmical Language", *PRISMA 01*, EuresisEdizioni, Milano, 2003.

[SCH 89] SCHOTTSTAEDT W., "Automatic Counterpoint", MATHEWS M. V., PIERCE J. R., Eds., *Current Directions in Computer Music Research*, The MIT Press, Cambridge, MA, 1989.

[SCH 02] SCHULTE C., *Programming Constraint Services*, vol. 2302 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Heidelberg/Berlin, 2002.

[SCH 08] SCHULTE C., SMOLKA G., Finite Domain Constraint Programming in Oz. A Tutorial, Mozart Consortium, 2008, `http://www.mozart-oz.org/documentation/fdt` (accessed 4 January 2011).