Sonic Arts Research Centre
Queens University Belfast, Northern Ireland

Essay for the Differentiation Process

Composing Music by Composing Rules: Computer Aided Composition employing Constraint Logic Programming

Torsten Anders

21st November 2003

Supervisors:
Prof. Michael Alcorn
Dr. Christina Anagnostopoulou

Contents

1. Introduction

Music is multidimensional. When listening to music, we perceive various aspects such as rhythm, melody, harmony, or instrumentation simultaneously. Also during the compositional process as well, the composer more or less consciously builds up a complex network of relationships between musical entities in a piece. Furthermore when analysing a piece, the music analyst searches for relations between score entities and describes them explicitly.

Some of the dimensions of music are notated explicitly in the score (e.g. note parameters, as duration and pitch). Other dimensions (one may also call them derived parameters) are only contained implicitly in the score (e.g. the interval between two successive pitches). There are also a number of musical dimensions which only become apparent after some kind of analysis. Such dimensions can be a harmonic progression, for instance, a cadence. These 'hidden' dimensions play a crucial role in musical comprehension.

In an attempt to formalise the compositional process, a single compositional rule may describe a single musical or derived parameter. However, to describe a specific musical style, usually multiple rules hold at the same time. To state a very simple example: only two rules are needed to describe an all-interval series (see figure 1.1): One rule ensures that all pitches – an explicitly represented parameter – are pairwise distinct. The other rule ensures that all intervals between successive pitches – an implicit information – are pairwise distinct. An all-interval series fulfils both rules, which is a conjunction of rules.

The subject of the present text is constraint based computer aided composition, a particular computer aided composition approach. Using this approach the composer generates a musical score by describing it on a level of abstraction similar to the definition of the all-interval series example. The composer defines compositional rules like the two above mentioned rules. A rule definition applies constraints, in other words, the restrictions

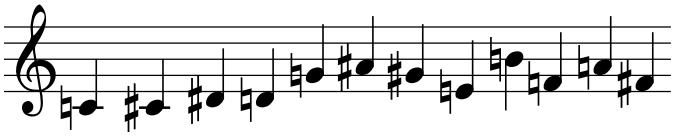


Figure 1.1.: An all-interval series

which the result of the program must fulfil.

The composer using constraint based computer aided composition does not describe a score in a theoretical way, in fact his score description is a computer program. When the program is executed, it searches for a solution to satisfy the given constraints. Thus constraints programming frees the composer to concentrate on *what* he wants to do in a musical sense; the *how* is left to the computer.

Compositional rules, implemented by constraints, may affect musical parameters as note durations, note pitches etc. and that way cover musical dimensions as rhythm, harmony, voice-leading etc. Compositional rules interrelate score parameters which are either represented explicitly or implicitly.

Compositional rules are defined as modular entities which can be logically connected in the score description. For instance, there may be many rules on voice-leading which are connected by conjunction.

Compared to other computer aided composition approaches, the attitude of constraint based computer aided composition is particularly close to the way many musicians and music textbooks are thinking or talking: musicians often describe, for instance, a certain compositional style by a set of compositional rules. The concept of constraint based computer aided composition is hence intuitively understood by many musicians.

The text proposes a generalisation of of some notions of constraint based computer aided composition. For instance, how can we formulate compositional rules which only hold in certain situations in the score? To give a traditional counterpoint example: usually all notes shall be consonant to simultaneous notes, however in certain circumstances there may be dissonances (e.g. passing notes on easy beat). Can we formulate this as a constraint problem as abstract as in counterpoint textbooks: this is the default constraint, however, in that particular situation that constraint holds instead.

Plan of the Paper

Chapter 2 introduces the notion of constraint based computer aided composition. The general area of computer aided composition is discussed in section 2.1. Section 2.2 explains the concept of constraint logic programming in the context of other programming paradigms, and section 2.3 surveys existing constraint based computer aided composition systems.

Chapter 3 discusses the research aims. These aims lead to constraint based computer aided composition systems which are more general than the systems described in section 2.3 and therefore provide a wider compositional freedom for the user.

Chapter 4 outlines design principles for an example of a constraint based computer aided composition system based on this research. The decision for a particularly potent programming platform for such a system, the Oz programming language is explained in section 4.1. This section also introduces constraint logic programming capabilities of Oz and sketches the search strategy applied. Principles for a score data representation of the constraint based computer aided composition system are sketched in section 4.2. Section

4.3 introduces in more detail the constraint based computer aided composition system by outlining levels of abstraction the system supports. An efficient full search strategy for scores is discussed in section 4.4.

Appendix A discusses issues of the score data representation of the constraint based computer aided composition system in more detail.

2. Background

2.1. Computer Aided Composition

The subject of the present text is constraint based computer aided composition. In order to introduce the reader to the subject, we first compare two related subjects *automated composition* and *computer aided composition* 2.1.1. The section 2.1.2 presents a number of computer aided composition strategies and introduces the notion of *deterministic computer aided composition*. Finally, section 2.1.3 surveys a few existing computer aided composition systems.

2.1.1. Computer Aided Composition vs. Automated Composition

Computer aided composition and automated composition often apply the same technical means. The main differences lay in the goal and the attitude of each subject. The following comparison may overemphasise theses differences for clarification – the actual work of researches and composers is often situated between these extremes.

2.1.1.1. Goal of Automated Composition

The goal of automated composition is to formalise the composition process of music as a subject for scientific research. The result of the research often results in a model which automatically composes music in a certain musical style.

Usually automated composition formalises the composition process of conventional music to allow scientific verification by comparing the original and the model-composed music. The verification can, for instance, be conducted by a variation of the Turing test: a human listener must recognise which music was composed by a human and which by the automatic model.

The automated composition model is usually expected to compose music as fully as possible. That is, the music will not be edited by humans afterwards, because such an editing makes the scientific verification of the model more difficult.

There exist various strategies for automated composition. Two main approaches are rule-based strategies and machine-learning bases strategies.

2.1.1.1.1. Rule-Based Automated Composition Rule-based strategies of automated composition are closely related to constraint based computer aided composition. From a musicians point of view, a rule-based approach appears a natural way to instruct a computer for automated composition. The first attempts to realise such an approach have been conducted in the 1950s – about as early as the beginning of computer science itself.

Hiller and Isaacson [1993] transformed compositional rules into computer instructions to compose the *Illiac Suite for String Quartet* (1956). For example, in their experiments one and two, Hiller and Isaacson realise counterpoint by defining melodic rules and harmonic rules.

In the following decades, computer scientists, mathematicians and musicologists worked on the formalisation of music as described in music textbooks on, for example, counterpoint and harmony. A programming strategy that has proven particularly useful for this task is constraint logic programming (see section 2.2.2).

For instance, constraint programming has been applied extensively to automatic music harmonisation. Pachet and Roy [2000] provide a survey on the subject constraint based music harmonisation. Ebcioglu [1992] developed a particular complex constraint system which produces choral harmonisations in the style of Johann Sebastian Bach.

2.1.1.2. Goal of Computer Aided Composition

The goal of computer aided composition is to provide means which assist the creative composition process of a human composer. In opposite to automated composition, composing with computer aided composition means is an artistic act. A composer using computer aided composition applies certain technical means, however this technical means are not the actual purpose. Instead, the purpose is the artistic output. Like any other artistic act, composing with computer aided composition means has no scientific method. There is also no scientific method to evaluate or even verify the outcome of an artistic act. However, research on the subject of computer aided composition, that is research on tools and strategies for computer aided composition may well be based on scientific grounds.

In contrast to automated composition, the composer who applies computer aided composition does rarely want to copy a traditional style. Instead, the composer wants to explore new musical means and develop his own musical style(s).

Also in contrast to automated composition, a composer who applies computer aided composition does not need to model the whole compositional process. He uses the software to assist him in solving subtasks, providing unexpected musical ideas and music transformations. The composer may use the software to generate the full score. However, often only sections of a composition are generated with the help of the computer.

The composer may apply an already existing computer aided composition environment (see below, section 2.1.3). Nevertheless, the composer usually writes the actual composition program himself and the process of programming is an integral part of the

compositional process. The composer judges the output of his program and often edits his program to better meet his musical needs. The composer may also further edit the computer output according to his musical needs. Finally, the composer composes the output sections of the program to the actual piece. It is the human composer who controls the actual composition process.

2.1.2. Deterministic Computer Aided Composition

The subject of computer aided composition is a rich field which developed diverse strategies to create music. For instance, computer aided composition strategies are often based on a certain mathematical model. Examples of such strategies include the application of stochastic models (e.g. [Xenakis, 1992]), or of mathematical models of chaos or self-similar systems to generate a musical score (e.g. [Supper, 2001]). Other strategies transform and map data on a score which is won, for instance, by analysing the changing spectrum of a recorded sound. Some strategies of computer aided composition implement an already existing composition strategy which developed outside the actual domain of computer aided composition, for instance serial composition (e.g. [Koenig, 1991–2002; Laske, 1981]). Finally, some computer aided composition strategies are based on a specific programming concept. Constraint based computer aided composition is of this latter kind. Many strategies work offline, that is the composition is generated in a non-realtime process. However, in recent years a growing number of composers work in the field of interactive composition (e.g. [Rowe, 2001, 1994]).

A systematic survey of strategies in computer aided composition – with historical excursuses – is provided by Roads [1996], Taube [2003], Miranda [2001], Alpern [1995] and Pope [1995]. Assayag [1998] outlines a history of computer aided composition. Papadopoulos and Wiggins [1999] offer a systematic overview with a focus on systems based on techniques from Artificial Intelligence.

Most approaches of computer aided composition apply *deterministic* programming techniques. In a deterministic program, each computational step only depends on the current state Usually, a deterministic program always returns the same result – however, even a (pseudo-)random operation is a deterministic operation.

Most deterministic computer aided composition strategies win data in some way and map the data into the score representation. For example, let us assume a composer uses a score representation which consists of a flat list of musical events, and each event consists of a number of parameters such as start time, duration, amplitude, frequency and so on. The composer combines a few computer aided composition strategies in a simple way: the composers binds all event start times directly to data generated randomly, all event durations to data generated by a pattern stream, all event frequencies to data won by a sound analysis strategy and so on.

One may understand the data generation and parameter binding as a compositional rule. In the example above, the rule for all start times says: “Generate a random number and bind it to the start time”.

The strategies to win the data as well as the techniques to map the data into the score – that is the compositional rule – may be arbitrarily complex. For instance, a more complex data generation strategy may make each generated datum dependent on the datum generated previously by the generation strategy (in fact, the random number example above may already be generated this way, because pseudo random number generators often depend on their previous output). A more complex mapping binds the frequency of every second event by one compositional rule and the remaining event frequencies by another rule.

Nevertheless, although a compositional rule (the data generation as well as the mapping strategy) may be arbitrarily complex, there is a principle limitation: each event parameter (including all its derived parameters, that is, implicit parameters as the implicit interval between two explicit pitches) is always bound by a single rule only. Using deterministic computer aided composition it is not possible, for instance, to bind the pitches of some notes by one rule and to bind the intervals between these pitches by another rule as it is necessary to generate an all-interval series (see section 1). Of course there exists an algorithm to generate an all-interval series deterministically, but to apply this algorithm means to apply a single and more complex 'rule'.

In a further example a composer wants to implement the following score description: "A piece consists of two voices. The pitch of the first note in the first voice is $b4$, the pitches of the following notes of that voice are random pitches in a range a fifth above or below their predecessor pitch." This description of a piece states a dependency of pitches to their respective predecessor note pitch.

The example resembles a single traditional counterpoint rule. However, to describe a score in a complex musical style (e.g. contrapunctual music) one often needs to combine several compositional rules.

Outside the domain of computer aided composition, musicians and textbooks on composition tend to describe music by multiple compositional rules, which are independent from each other. For instance, tonal music shows a very dense woven net of relations between score entities. It is more easy to describe such complex music by multiple individual rules than by describing it with a single rule, which will be much more complex.

Most computer aided composition environments only allow a pure deterministic score description. Pure deterministic computer aided composition does not support a level of abstraction in which multiple rules on the same parameter can be freely combined. This limitation is not necessarily felt in the domain of deterministic computer aided composition. Composers adopted their way of working and thinking accordingly. Nevertheless, music with a net of mutual relationships as dense as, for instance, in tonal music is very hard to realise by pure deterministic computer aided composition.

2.1.3. Computer Aided Composition Systems

Many systems have been developed as tools for composers using computer aided composition strategies. Often composers develop their own tools according to their needs,

but there are also systems that have been used by hundreds of composers. Roads [1996] lists many historical systems. The following paragraph mentions a few systems which are currently in use.

OpenMusic [Assayag et al., 1999; OpenMusic] is a graphical computer aided composition programming language with many library extensions for various composition strategies. PatchWork [Laurson and Kuuskankare, 2002], the predecessor of OpenMusic, is further developed in parallel. Common Music [Common Music; Taube, 1997, 2003] is a computer aided composition programming language with specifically flexible support for nested pattern streams. Haskore [Hudak; John Peterson and Olaf Chitil] is a computer aided composition programming language which explicitly supports both the composition and the interpretation of a score. Max [Cycling74; Puckette, 1991] is a graphical and real-time computer aided composition programming language. Max/MSP [Zicarelli, 1998], an extension for Max, supports sound synthesis too. SuperCollider [McCartney, 1996, 2002; SuperCollider] is a real-time sound synthesis and composition programming language. Siren [Pope] is a sound synthesis and composition programming language.

2.2. Programming Paradigms

The current section introduces principle computer science concepts for programming. These different concepts form the foundation for different programming languages and are usually called *programming paradigms*. Section 2.2.1 surveys some particular significant paradigms. Section 2.2.2 introduces the paradigm *constraint logic programming*, which is the foundation of constraint based computer aided composition. Finally, section 2.2.3 sketches how different programming paradigms can interact in *multi-paradigm programming*. The notion of multi-paradigm programming will become important later in the text.

2.2.1. Programming Paradigm Survey

Various programming paradigms are used for computer aided composition, for instance, functional programming, object-oriented programming, concurrent programming, procedural programming, and constraint logic programming. Different programming paradigms are appropriate for different compositional tasks and problems.

Often, a composition environment supports a specific programming paradigm particularly well and encourages the composer to employ this paradigm. For instance, Haskore supports functional programming, SuperCollider and MODE/siren support object-oriented programming, while Max respectively Max/MSP support concurrent programming.

For an introduction to programming paradigms and how they can interact see Abelson and Gerald Jay Sussman with Julie Sussman [1985], Friedman et al. [2001], or van Roy and Haridi [2003]. The following paragraphs briefly depict the benefits of some programming paradigms.

In *functional programming*, each computation is an evaluation of an expression: a function gets arguments and returns a result; no side effects occur. The paradigm introduces full compositionality, lexical scoping, and higher-order programming (i.e. functions as first-class citizens). Functional programming can be studied by the literature on functional programming languages, as Haskell, SML and Scheme [Abelson and Gerald Jay Sussman with Julie Sussman, 1985; John Peterson and Olaf Chitil; Mark Kantrowitz and Barry Margolin; Standard ML].

Stateful programming – that is procedural programming and, usually, object-oriented programming – explicitly represents data that changes over time, while in stateless programming – that is pure functional and pure logic programming – data never changes. Stateless programming is usually easier to reason about, while stateful programming adds memory to computations. Stateful programming can be studied by the literature on stateful programming languages, as for example C [Summit].

Object-oriented programming organises computational activities into entities called objects. Objects encapsulate state and methods to manipulate it. This abstraction simulates how we perceive our external world. Inheritance supports incremental development and code reuse. Object-oriented programming can be studied by the literature on object-oriented programming languages, as Smalltalk, CLOS, Java, and C++ [Cline; Elliotte Rusty Harold; Keene, 1989; Kiczales et al., 1991; Malik; Paepcke, 1993; Stroustrup, 3rd edition, 2000].

Concurrent programming supports the execution of computations in independent threads. In a concurrent program, the execution order of computations in different threads is undetermined. If the computer hardware allows it, computations may even run concurrently in time. Concurrent programming can be studied by the literature on concurrent programming languages, as Erlang and Java [Armstrong et al., 1996; Elliotte Rusty Harold].

The *logic programming* paradigm is a declarative programming paradigm. Logic programming supports a programming style in which the user declaratively states a symbolic relation which involves unknowns (i.e. variables in the mathematical sense). The user asks the computer to find one or more solutions for the unknowns depending on all relations given to the computer.

Two main concepts of logic programming are *logic variables* and *unification*. Logic variables represent unknowns. Logic variables can handle partial information. For example, a variable may be known to be a list, but the content of the list is unknown. When two logic variables are unified they become equal. The unified variables share the (potentially partial) combination of information of both variables before unification. This results in a potential increase of information. A logic variable is said to be determined if it is fully known.

Computations in logic programming are (simplified) symbolic deductions. A logic relation can be understood as a procedure with arguments but no explicit return value. Any relation argument can be an undetermined logic variable. To find a solution (i.e. one or more suitable determinations for all relation arguments) possibly a search must be

performed. Logic programming can be studied by the literature on logic programming languages, as Prolog [Bratko, 1987; Prolog].

The constraint logic programming paradigm, the foundation for constraint based computer aided composition, is discussed in the following section.

2.2.2. Constraint Logic Programming

Logic programming can only handle symbolic information (i.e. symbols and combinations of symbols in, for instance, a list). *Constraint logic programming* is a generalisation of logic programming which complements logic programming by efficient means to solve numerical problems as well.

The constraint logic programming paradigm extends the concept of logic variables to, for instance, *finite domain variables* which represent undetermined numbers or *finite set variables* which represent undetermined sets. As logic variables allow to compute with undetermined symbolic values, so finite domain variables allow to compute with undetermined numbers. Relations between these variables are called *constraints*. The set of possible values which a finite domain variable can take is called the *domain* of the variable. The search engine which solves a constrain problem is often called a *constraint solver*.

For instance, a constraint program may state the following two equations:

$$\begin{aligned} x + y &= 7 \\ x + 1 &= y \end{aligned}$$

The constraint solver is able to conclude the value of the two variables ($x = 3, y = 4$).

For an introduction to constraint logic programming see one Marriott and Stuckey [1998], Frühwirth and Abdennadher [1997]; Frühwirth et al. [1993], or Bartak [1998]. Constraint programming languages are discussed in section 4.1.

2.2.3. Multi-Paradigm Programming

For computer aided composition a combination of paradigms is often used. This allows the user to deal more flexibly with the complexity of the field.

The expressiveness of multi-paradigm programming is probably also one of the reasons why Common Lisp [Pitman] is often employed as the foundation for computer aided composition environments (e.g. PatchWork/OpenMusic [Assayag et al., 1999], Common Music [Taube, 1997, 2003], see section 2.1.3). Lisp supports programming in different programming paradigms and even allows the user to add new paradigms. Within

such a multi-paradigm programming style the user may separately choose the adequate paradigm for each given sub-problem.

Peter van Roy, Per Brand, Denys Duchier, Seif Haridi, Martin Henz and Christian Schulte [2003] as well as van Roy and Haridi [2003] show how logic programming becomes more expressive if combined with other programming paradigms in a multi-paradigm programming language.

2.3. Constraint Based Computer Aided Composition

Computer aided composition offers means for the composer to generate a musical score by programming means. Deterministic computer aided composition (see section 2.1.2) offers the composer powerful new means to approach the composition process. However, the level of abstraction found in deterministic computer aided composition is limited if compared to the level of abstraction musicians use when describing a musical style.

Music descriptions as, for instance, found in composition textbooks often state rules on single score objects (e.g. notes) or parameters (e.g. durations or pitches) or mutual dependencies between multiple score objects and parameters. Such restrictions and dependencies don't necessarily result in only a single solution score. Instead, the restrictions and dependencies only reduce the domain of all possible scores.

In computer science, programs which define restrictions or mutual dependencies with multiple solutions are often called *non-deterministic* programs. One, multiple, or all solutions of such programs are found by search (which is implemented by deterministic programming means). The restrictions or mutual dependencies are called *constraints*. A programming paradigm which supports the user to entail such restrictions is constraint logic programming (see section 2.2.2). Computer aided composition which employs the constraint logic programming paradigm is called *constraint based computer aided composition* in this text.

The power of constraint programming for computer aided composition lies in the fact that the composer defines multiple compositional rules (implemented by constraints) which affect the same parameter values in a modular fashion. For instance, the composer constraints the note pitches of a score by multiple rules on voice-leading as well as on the harmonic structure. Each of these separate rules affects the same parameter values, namely the pitches. However, no rule necessarily determines the parameter values fully. Search finds a solution which fulfils all compositional rules.

For a composer, it is particularly intuitive to describe a score or a musical style by modular compositional rules. That way, different aspects of the score or style can be described in a modular way. Textbooks on music composition also teach the student this way.

The following sections on PWConstraint (2.3.1), Arno (2.3.2), and OMCoulds (2.3.3) introduce three systems for constraint based computer aided composition more closely.

Each system allows the composer to define his own compositional rules and that way to describe his own style of music. The section lists similarities and differences between the systems and points out their respective strengths and weaknesses.

Each of the three systems is integrated in a computer aided composition environment. All systems are implemented in Common Lisp.

Other systems are often less expressive than the three mentioned above. These are briefly described in section 2.3.4.

2.3.1. PWConstraints

PWConstraints is one of the first constraint based composition environments. The system allows the user to define compositional rules – restictions a solution score shall fulfil – in a concise and expressive way.

PWConstraints is a constraint programming language on top of PatchWork (see section 2.1.3). This language consists of two main parts. A general constraint programming language offers means to constrain, for instances, a list of integers. A special constraint programming language for musical scores offers means to constrain the pitches of a score.

In the graphical programming language PatchWork, programming constructs are graphical boxes. Consequently, each main part of PatchWork comes as such a box. The user interface of the general constraint programming language is the box `PMC`. The box `score-PMC` is the interface for the special constraint programming language for musical scores. However, PWConstraints complements its graphical interface by a textual interface: the main input to the PWConstraints boxes – the actual constraints – are defined by textual Lisp code, using special functions/macros provided by PWConstraints.[1]

The current section describes PWConstraints from a users point of view. To introduce the reader into the subject of constraint based computer aided composition, this introduction of PWConstraints is relatively elaborate. PWConstraints demonstrates the power of constraint based computer aided composition, but the system demonstrates also some principle problems which are addressed later in the text.

2.3.1.1. The general constraint programming language: PMC

The next paragraph explains the basic concepts of PWConstraints by sketching a simple example. Let us assume a composer wants to create a choral melody. The composer decides that the melody consists of 9 notes, all notes are of equal duration. All melody pitches are situated in the interval between $a3$ (an octave below concert pitch) and $a4$ (concert pitch).

[1] The main purpose of the graphical interface to programming offered by PatchWork is to make the programming more easy for the composer. However, not every program becomes more easy to write in a graphical way. In particular, the pattern matching mechanism provided by PWConstraints (see section 2.3.1.1) can only awkwardly be expressed graphically.

Because the composer is only interested in the note pitches, he can represent the melody by a plain list of pitches. Therefore, the general constraint programming language of PWConstraints is sufficient for this example. In PWConstraints, pitches are represented by their respective MIDI number. The MIDI pitch representation of the concert pitch $a4$ is 69, $a3$ is represented by 57.

The above mentioned composer defines a list of 9 finite domain variables to define a melody according to his initial decisions (for constraint programming related concepts and terms see section 2.2.2).[2]. Each variable is defined with the interval $[57, 69]$ as domain. Each combination of nine pitches in the specified domain is a solution to this initial definition (9 variables – each with a domain of 12 values – have $12^9 = 5159780352$ solutions in total).

The composer further specifies his desired melody by imposing certain restrictions – compositional rules, implemented constraints – on the solution melody.[3] Let us assume the composer wants to restrict the possible interval between successive pitches to a fifth at maximum. Because the pitches are encoded numerically, the composer can define numeric relations between them. In PWConstraints, a compositional rule is a boolean test. A possible definition of such a rule which restricts successive intervals to a fifth as maximum looks like this (the example uses a pseudo-code representation for an example from the PWConstraints documentation).

$$\text{let } interval = pitch_{predecessor} - pitch_{successor}$$
$$isMember(interval, [-5, \ldots, 5])$$

However, the rule definition above is unfinished: the rule definition must be applied to the choral melody notes such that it is defined for all successive pitches (the variables $pitch_{predecessor}$ and $pitch_{successor}$ are free in the example).

To apply a rule to a score, `PMC` offers the composer a pattern matching mechanism. Pattern matching may be known to the reader, for instances, from the UNIX shell. In the UNIX shell, a pattern is used to express a character sequence: for instance, a pattern as '*test.txt' matches the file names 'mytest.txt' as well as 'my-other-test.txt'.

The pattern matching mechanism of `PMC` supports the matching of elements in a list. For instance, a wild card (`*`) matches zero or more list elements. The pattern matching language of `PMC` supports variables, which are used to bind the free variables of rule definitions. The pattern matching expression which matches all successive pitches in the choral melody looks like this:

$$[* \; pitch_{predecessor} \; pitch_{successor}]$$

[2] In the PWConstraints documentation, the term *search variable* is used for a finite domain variable.
[3] In the PWConstraints documentation, the terms *rule* and *predicate* are used for a compositional rule.

In PWConstraints, a rule definition consists of a pattern matching part and the actual rule definition. The system makes sure that in a solution every compositional rule returns true for every match of its corresponding pattern matching expression.

By applying the above mentioned voice-leading rule to the choral melody, the composer reduces the number of possible solutions for his melody. All solutions will fulfil this rule. However, the advantage of constraint programming is that the composer may define further rules for his melody to describe the desired result more accurately.

Lets assume the composer wants to restrict his melody such that the highest pitch occurs only once in the melody. A possible definition says that the current pitch must not equal the maximum of all its predecessor.

The composer defines each rule as a modular entity. This fact is of particular importance. In the composition process the composer will constantly change the set of rules applied and the definition of the rules. Because each rule definition is independent of the other rules, the composer can add, remove, or change a rule of his melody description without affecting the other rules.

PWConstraints performs a full search. The user may ask for a single solution or for multiple solutions. Besides, the user may also predefine certain values of the solution. For example, the user may specify that a solution begins and ends with a certain pitch.

The example above only defines strict rules. To avoid over-constrained situations, PW-constraints allows to define heuristic rules as well. Heuristic rules must not hold under all circumstances, however, the user can rate their importance. In case of a conflict between heuristic rules, the more important heuristic rule is preferred to hold.

2.3.1.2. Score constraints: score-PMC

In case the composer does not only want to define a monophonic melody but a polyphonic score, he needs to switch to the special score constraint language of PWConstraints, represented by the PatchWork box `score-PMC`. `score-PMC` supports the definition of compositional rules which rely on different *contexts*.

Melodic Context : For each note in a melodic succession (i.e. in a voice), a compositional rule may refer to its predecessors and successors.

Harmonic Context A rule may refer to the notes sounding at the attack time of the current note.

Harmonic Slice A rule may refer to the notes sounding simultaneously with the current note.

Metric Context A rule may refer to a user defined metric pattern and the position of the current note in that pattern.

By using the melody context, the composer defines voice leading rules in a way very similar to the approach shown in the choral melody example. This context also supports a pattern matching mechanism to apply compositional rules to the score.

Harmonic rules are defined by accessing simultaneous pitches using, for instance, the harmonic context. Such a rule may, for instance, restrict simultaneous pitches to be consonant. Each rule is defined as a boolean test, which can be arbitrarily complex. For instance, a rule controlling the harmony may cosntrain the intervals between some voice and the bass voice differently than an interval between non-bass voices.

2.3.1.3. The Search Strategy

PWConstraints employs a constraint solver developed explicitly for the system. The solver performs a backtracking search over finite domains of integers.

Most compositional rules are simply defined as boolean predicates. During the search process, the system first binds the currently visited finite domain variable with a value of its domain. Then the search process checks whether all rules hold for this value. If the conjunction of all rules returns true, then the search proceeds to the next variable. If the check fails, then the search tries another domain value of the current variable. If all domain values fail, the search goes back to a variable which has already been determined earlier in the predetermined search order (i.e. the search backtracks). The search tries another domain value of this variable.

The solver can also perform forward-checking constraint-propagation for rules which are specifically defined for forward-checking. Forward-checking rules reduce the domain of variables not yet visited during search.

The score-PMC performs the same search strategy. However, the system first computes an efficient predetermined search order in which the notes in the score are visited during search. To this end, score-PMC expects a score as argument which predefines the rhythmic structure of a solution besides the rule definitions. Using this rhythmic structure, a search order is established which progresses in score-time: notes with lower start-time are visited more early then notes with higher start-time.

2.3.1.4. Limitations of PWConstraints

PWConstraints is a complex system for constraint based computer aided composition, which allows to define complex rule sets to generate music. In the PhD thesis of Laurson [1996] on his composition environment PatchWork however, PWConstraints is only an example for a major PatchWork application. Therefore, the system is also limited in the rules it can define.

The first limitation is that rules can only describe the pitches of a score. In particular, the rhythmic structure of the score must be fully predetermined.[4] This includes, that

[4]This limitation is caused by the underlying search strategy of PWConstraints (see page 18).

the score data structure – how many voices, how many notes per voice etc. – must be predetermined. PWConstraints searches only for the score pitches.

In newer extended versions of PWConstraints other note parameters can also be set, rhythmic values for instance. However, the program still requires a rhythmical input. Internally, search variables are still encoded as pitches. These may be converted to change the rhythmic structure.

The pitches themselves are limited to MIDI note numbers to reduce the size of the search space – PatchWork itself supports a pitch resolution of cent accuracy, called MIDI-cent.[5]

PWConstraints offers the user to apply rules to the score by a pattern matching mechanism, which is a very flexible and convenient way for many rules. For the designer of PatchWork, this pattern matching mechanism was an integral part of the system design. The acronym in the name of the main boxes of PWConstraints, `PMC` and `score-PMC`, stands of 'Pattern Matching Constraints'.

One of the main restriction of the pattern matching approach is, that the system defines only a finite number of elements in the pattern matching syntax. For instance, there is not syntax element to express 0 or more list elements, which become bound to a variable and can be referred to in a rule (i.e. there is no corresponding variable-expression for the wildcard '*'). Unfortunately, the user can not extend the number of predefined pattern syntax elements.

Even more restrictive is the notion that a pattern matching language is only appropriate for sequences of pitches — as supported by the general constraint programming language PMC. In a tree-like (or even graph-like) data structure as a score is, the sequence oriented nature of the pattern matching approach meets severe limitations. Consequently, the score-PMC language supports pattern matching only for the melodic context. For the other contexts (as well as in cases where the pattern matching syntax is limited, see proceeding paragraph) PWConstraints offers many predefined macros, which serve as accessors to data of the internal score representation.

The number of predefined contexts is limited to the number of the four contexts mentioned above.[6] In the definition of a canon, for example, the user will wish to access the all notes which are at the same position in different voices. The user will, for instance, define that theses notes are of the same pitch. In PWConstraints, the user can not add a new context. Also the score data structure can not be extended. For example, the user may wish to represent the harmonic structure of a score in an explicit way to control the harmony on a higher level. PWConstraints does not allow the user to extend its score data representation.

All rule definitions are connected in PWConstraints by an implicit conjunction. The system does not support any other logical connective.

[5] in PatchWork, 6000 corresponds to middle c and 6033 is a $\frac{1}{6}$-tone above.

[6] This restriction is caused by the internal score data representation. For the search process, the score is represented as a flat bidirectional linked-list of notes. All contexts not directly accessible in this representation are represented as additional data slots to notes.

In PWConstraints, the declarative side of a rule definition and the procedural side of the search process are interwoven. Compositional rules are defined as boolean predicates. When a predicate is applied during search, all values involved in the predicate must be determined. Particularly, the predicate can only rely on context values which already have been visited during search and are therefore already determined.

2.3.2. Arno

The overall goal of Arno [Anders, 2000a,b] is very similar to the goal of PWConstraints. Arno allows the user to describe a musical score by defining compositional rules. However, the system was designed to overcome a few of the restrictions the user meets with PWConstraints: Arno supports arbitrary domain values, allows the user to define constraints on arbitrary score parameters, and supports a more general notion of context dependent constraints.

2.3.2.1. Screamer

Arno employs Screamer [Screamer Resposity; Siskind, 1991; Siskind and McAllester, 1993] as constraint solver. Screamer complements Common Lisp to make a language efficient in solving numeric and symbolic constraints. Choice-point generators and the special operator `fail` are the main construct which Screamer adds to Common Lisp. A choice-point generator (such as `either`, `a-member-of`, and `an-integer-between`) returns a single value out of a set of alternatives. `fail` is used to prohibit some situations. That way Screamer supports a non-deterministic programming style as popularised by Prolog. Advantages of Screamer are is portability across Lisp implementations and its efficient implementation of non-deterministic Lisp.

Certainly, Screamer can be used directly to constrain, for instance, lists of pitches. This level of abstraction is comparable to the constraint language of the PMC box in PWConstraints (see section 2.3.1.1). However, the goal of Arno is to allow the user to constrain a score.

2.3.2.2. Searching for Scores

Arno extends Common Music (see section 2.1.3) by the means of constraint programming. Arno employs the score data representation of Common Music. The Common Music score representation is implemented by the object-oriented programming means (see section 2.2.1). This score representation defines a class hierarchy of score objects, including *events* (e.g. notes) and *containers*. Common Music containers determine the timing structure of events or other containers they contain. A *score object* is a super class of events and containers.

The Common Music score representation does not only define means to represent the actual score data. The representation also defines rich means to access and modify these

data. Arno further extends these score accessors. For instance, the extended Common Music *application programming interface* (API) provides means to access all elements of a container, to access the successor of some object in the container of that object, or to access all events in the score which are simultaneous to a given event.

Using Arno, the user constraints a Common Music score hierarchy, consisting of events and containers.

The Arno user can constrain arbitrary parameters of Common Music events, that is not only pitches but also the timing parameters (e.g. duration) or arbitrary sound synthesis parameters. In PWConstraint, only the pitch parameter can be constrained.

Event parameters are initialised as finite domain variables. In Arno, a finite domain does not only contain integers. A finite domain can consists of arbitrary values, for instance, fractions. Support for fractions for event parameters as, for example, duration and frequency allows more easily and precisely to represent complex rhythmical structures as well as a microtonal harmonic structures. In PWConstraints, the domain of a variable consists of integers only.

The Arno user defines how Common Music containers are hierarchically nested in the solution score. By this, the user defines formal aspects of the music, for instance, the number of simultaneous voices in each section of the score. Common Music containers can be arbitrarily nested.

Compositional rules restrict the parameter values of Common Music objects (an object is a super class of both events and containers). Like in PWConstraints, a compositional rule is a boolean predicate in Arno. However, Arno does not apply a pattern matching mechanism to apply a rule to the score (as PWConstraints does). Instead, each compositional rule is defined as a unary function in Arno. The argument of a rule is a Common Music object to check. The function returns true if the test succeeds.

Compositional rules are applied container-wise to the score: The core programming construct of Arno is a higher-order function in the sense of functional programming (see section 2.2.1). This function returns a Common Music container. The function expects a list of compositional rules, that is functions, as argument (besides initialisation arguments for the container). The elements of the returned container fulfil all compositional rules applied to the container.

Like a rule in PWConstraints, a rule in Arno may depend on the context of its object. Arno uses the full Common Music score representation for the search process (PWConstraints uses a reduced score representation in its search process). Because the full score representation is used for the search process, a rule definition can rely on the full score application programming interface and its extensions supplied by Arno. The context of a score object (e.g. the predecessor of a note in a voice) can be access by this API.

Using the interface to access the context of an object, the user can define n-ary constraints between score parameters which are related by some context. For instance, the definition of a contrapunctual voice-leading rule which limits the pitches of two successive notes into

a certain range (see the example in section 2.3.1.1) may be very similar in PWConstraints and Arno. An unfinished rule in both systems may look like this:

$$\text{let } interval = pitch_{predecessor} - pitch_{successor}$$
$$(-5 < interval) \vee (interval < 5)$$

The main difference is that the free variables $pitch_{predecessor}$ and $pitch_{successor}$ are bound by a pattern matching mechanism in PWConstraints and by Common Music score API functions in Arno.

The main advantage of using score API functions to access the context of a score object is that the API can easily be extended to cover additional contexts.

2.3.2.3. The Search Strategy

Arno employs Screamer, which performs a backtracking search. Similar to PWConstraints (see section 2.3.1.3), compositional rules in Arno are defined as boolean predicates. As in PWConstraints, the search process first binds the currently visited finite domain variable with a value of its domain. Then the search process checks whether all boolean rules hold for this value. The system progresses or backtracks according to this test.

Similar to PWConstraints, Arno performs a search with predetermined search order. However, Arno is developed to generate the rhythmic structure of polyphonic music too. The system can therefore not guess in advance which events will be simultaneous in time. Hence, Arno can not pre-compute an efficient search order which determines score events with smaller start time before later starting score events. Instead, the search proceeds container-wise and performs thus redundant work. Consequently, the performance of Arno degrades for longer, non-monophonic examples. For polyphonic music that can be reduced to monophonic music (e.g. canons) the performance is much better.

The performance can be increased by applying backjumping.[7] Heuristics may sort the domain order of variables and thus increase the search performance or lead to a better solution.

2.3.2.4. Limitations of Arno

Besides its poor performance, Arno has further limitations. Some limitations are very similar to limitations in PWConstraints (see section 2.3.1.4).

Arno connects all rules by an implicit conjunction. The system does not support other logical connective, although negation of single rules is supported too.

[7] Backjumping is is a variant of chronological backtracking. Backjumping skips variables in case of failure and jumps back directly to the variable with causes the conflict. The algorithm is sometimes also called intelligent backtracking.

In Arno, the declarative side of a rule definition and the procedural side of the search process are interwoven: rules are implemented by predicates and all values involved in the predicate must be determined. Therefore, a rule can relate the value currently visited by the search only to values which are already determined.

Arno supports searching for the values of arbitrary parameters of an arbitrarily structured score hierarchy. However, the whole hierarchic structure itself must be fully determined before search.

Although Screamer supports to search for multiple or all solutions of a problem, the Screamer application Arno supports only the search for a single solution.

Arno lacks means to handle over-constrained situations.

2.3.3. OMClouds

OMClouds [Chemillier and Truchet, 2001; Truchet et al., 2001a,b] is a constraint based computer aided composition system which extends the composition environment OpenMusic. OMClouds has similar goals as PWConstraints and Arno: OMClouds allows the user to describe a musical score by defining compositional rules.

The system defines its own constraint solver which performs an efficient local improvement search strategy Codognet and Diaz [2001]; Codognet et al. [2002] . This strategy allows to quickly find approximated solutions to a constraint problem which fulfil many or most of the constraints imposed. The user can observe the refinement of the search and stop the process when the current solution is adequate.

The user expresses the constraint problem in logical form. The constraints are automatically translated into cost functions to guide the search progression.

Instead of supporting the use of a special score data representation, the user defines his own music representation using general data structures like nested lists. The data structures contain finite domain variables which can be constrained. In this respect, OMClouds supports a level of abstraction more simple than the score-PMC language of PWConstraints (see section 2.3.1.2) or Arno (see section 2.3.2.2). Instead, OMClouds is in this respect comparable to the plain PMC language of PWConstraints or the direct application of Screamer (see section 2.3.2.1) to music composition. On the other hand, this simplicity allows the user to freely design his own music representation.

OMClouds supports easy-to-use means to apply constraints homogeneously to multiple variables, for instance to all variables in a list. However, OMClouds does not support more complex means to apply constraints only to specific variables comparable, for instance, to the pattern matching mechanism of PWConstraints.

The heuristic approach of OMClouds is very attractive for the composer, because some complex constraints are relatively easy to express. In systems which truly fulfil every constraint (as PWConstraints and Arno) compositional rules must explicitly handle special cases in which the rule is fulfilled anyway. In OMClouds, rule definitions can be

less carefully formulated: the heuristic search strategy does not reject a solution because some rules are not always fulfilled. On the other hand, OMClouds supports less precision in the description of a score.

Adaptive search does not execute a complete search. If the system is asked multiple times for a solution to the same problem it may come up with different solutions. However, the system can not output all solutions.

Because adaptive search does not execute a complete search, strict constraints as 'all elements of a list are pairwise distinct' are hard to fulfil.

2.3.4. Survey of Other Constraint based Computer Aided Composition Systems

The following paragraphs briefly survey constraint based computer aided composition systems.

During the last decade, interest rose to apply constraint logic programming for computer aided composition. Bonnet and Rueda [1999] developed Situation, another constraint based library for PatchWork as PWConstraints. Situation implements a set of predefined rules by constraints. The user can customise these rules by specifying parameters to the rules. Situation was designed specifically to generate chord sequences. Later versions also support the generation of rhythm. The predefined rules make Situation more easy to use than PWConstraints. However, the predefined rules also restrict the expressiveness of Situation.

Constraint logic programming proved a very successful approach for computer aided composition. Therefore, the subject became an important research subject in the field of computer aided composition. Much research on the subject has been done at IRCAM [Agon et al., 1998; Rueda et al., 1998].

OpenMusic supports several libraries based on constraint programming. OMBT [Truchet, a,b] integrates important functionality of Screamer into the graphical language OpenMusic. OMRC [Sandred, 2000a,b] is a library for controlling rhythm by constraints. OMSituation is the current version of the PatchWork library Situation.

Some research focuses on score representation and score search strategies. Curry et al. [2000] show how a set of score trees is represented by finite domain constraints. Beurivé and Desainte-Catherine [2001] discuss the representation of score hierarchies with constraints. Anders [2002] proposes an efficient and complete search strategy for a score hierarchy with undetermined rhythmic structure but with constraints dependent to the rhythmic structure (see section 4.4).

PiCO [Alvarez et al., 1998] combines concurrent object-oriented programming and constraint programming as primitive entities in a single calculus. The calculus was defined to form a basis for compositional tools. Rueda and Valencia [2001] continue this research.

Constraint programming is not only applied to generate a score. Midispace [Pachet and Delerue, 1998] realises sound mixing and spatialisation using constraint programming.

Ferrand et al. [1999] propose an approach for optical music recognition which uses constraint programming.

3. Research Aims

The previous chapter described some of the most relevant constraint based computer aided composition systems and their respective limitations. Some limitations restrain the compositional freedom of the user. Other limitations result in cumbersome formulations of complex compositional rules.

The motivation behind the development of the current system is to overcome some of these limitations. In particular, the research aims

- to fuse concepts of deterministic composition and constraint based composition such that benefits of both approaches are combinable
- to introduce a more general notion of context dependent constraints to allow a more concise formulation of complex compositional rules
- to allow the user to constrain the score hierarchy as well which is a prerequisite to formulate constraints on the musical form

3.1. Fusing Conventional Computer Aided Composition Concepts and Constraint Based Computer Aided Composition

Deterministic computer aided composition (section 2.1.2) developed expressive concepts for music composition which often support unconventional approaches and result in new musical styles. An example for such an expressive concept are *pattern streams*. A pattern stream allows the user to generate a sequence of values which forms patterns. An examples for a simple pattern is a stream generating repetitions of a value sequence or generating an arithmetic series (that is a series of numbers with a constant difference between two series elements). However, patterns can be much more complex as well. Patterns can also be nested. Because the approach is very general, pattern streams are implemented in several computer aided composition systems, for instance, Common Music and SuperCollider (section 2.1.3). For more details on pattern streams see, for instance, Taube [2003].

Constraint based computer aided composition (section 2.3) allows the composer to describe a score in a more abstract way than deterministic computer aided composition:

the composer can describe different aspects of the composition in modular composition rules. Multiple composition rules can effect the same score parameters. This allows the composer to generate a score which establishes a complex net of mutual relations between score entities.

Computer aided composition will become more expressive if we combine the expressive power of both deterministic computer aided composition concepts and constraint based computer aided composition.

Reformulating a concept – originating from deterministic computer aided composition – by constraints means to transform the deterministic concept into a non-deterministic concept (section 2.3). For instance, a deterministic pattern stream generating an arithmetic series generates a single solution. If the arithmetic series pattern is reformulated by constrains, and some score parameters are constrained to follow this pattern, the total number of solutions of that score is reduced to the scores in which the parameters form an arithmetic series pattern. However, the actual pattern itself may still be undetermined.

Consequently, concepts of deterministic computer aided composition which are reformulated by constraints can be used together with other compositional rules. For instance, the arithmetic series pattern can constrain the pitches of a melody and additional user defined rules may constraint the harmony and thus affecting the melody as well. A traditional musical example which uses a similar technique can be found at the beginning of the first movement in the fourth symphony of Johannes Brahms. All intervals between the first eight pitches of the starting theme are falling thirds (or rising sixth) – thus forming a simple pattern of the pitch classes, which is further controlled by the underlying harmony.

To reformulate a concept of deterministic computer aided composition by constraints therefore means to really fuse the concept with constraint based computer aided composition. The benefits of both worlds are combined, because the user can apply the proven concept (e.g. pattern streams) and further constrain the solution by additional compositional rules.

3.2. Generalised Context Dependent Constraints

PWConstraints supports the definition of context dependent constraints (section 2.3.1.2). In the score-PMC language of PWConstraints, the context of a pitch is, for instance, a list with all proceeding pitches in a voice. In PWConstraints, the user is limited to a few predefined contexts of a pitch.

Arno supports the definition of context dependent constraints as well (section 2.3.2.2). In Arno, a context of a Common Music score object means the set of score objects accessible by some function of the Common Music score application programming interface. That way, Arno generalises the notion of context dependend constraints defined for PW-

Constraints: in Arno, the user may define new contexts by defining new score accessor functions.

For instance, to define a canon the user wants to define a compositional rule which relates notes of different voices which are at the same position. That is, all first notes of each voice, all second notes of ech voice etc. become related, for instance, to have the same pitch. To define a compositional rule which consitutes a canon the user may define a new accessor function which is given a note as an argument and which returns all corresponing notes in the other voices of the canon. This new accessor function definition serves as a definition for a new context, the context of all notes of the canon which are at the same position in their voice.

Nevertheless, in both PWConstraints and Arno the user is limited to the contexts of a score entity which are already fully determined when this entity is visited during search. A more general constraint based computer aided composition system will allow the user to define compositional rules which rely on contexts which may or may not be predetermined.

An even further generalised notion of context understands also the validity of arbitrary compositional rules, implemented by constraints applied to some score objects, as a context of these objects, besides the contexts accessible by score accessor functions. For instance, in Arno a score accessor may return all notes which are (so far) known to be simultaneous to a certain note: the score accessor returns the context of simultaneous notes. In the generalised notion, the user may still follow this approach. Additionally, a compositional rule $IsSimultaneous$, which constraints whether two notes are simultaneous or not, is also understood as a context.

Applying such a notion of a context allows the definition of very concise compositional rules based on other compositional rules. For instance, the user may define that compositional rules only hold in certain circumstances. For instance, a counterpoint textbook usually states first the general (but too simple and restrictive) rule that there shall only be consonances between simultaneous notes. Later, the textbook explains in which specific circumstances there may be dissonances between simultaneous notes, for instance if the notes in one voice happen in a passing notes context.

We may define the first version of the rule very concisely by a logical implication.

$$IsSimultaneous(note_1, note_2) \rightarrow IsConsonant(note_1, note_2)$$

In this logical expression, the predicates $IsSimultaneous$ and $IsConsonant$ are both compositional rules (implemented by constraints). The rules define that two notes are simultaneous respectively consonant if the rule holds. However, the rules allow to constrain their validity (i.e. truth value) as well.

The above implication therefore expresses that the pitches of the notes $note_1$ and $note_2$ are consonant to each other if both notes are simultaneous in time. However, an implication is not simply a procedural `if`-statement but a logical relation. A logical relation

works in both ways: if the pitches of the notes $note_1$ and $note_2$ happen not to be consonant, then the notes must also not be simultaneous.

In a constraint based computer aided composition system which allows this level of abstraction, the extended counterpoint rule may be expressed by a similar logical equation. The equation states that two simultaneous notes, $note_1$ and $note_2$, must be consonant unless either $note_1$ or $note_2$ is situated in a context of passing notes.

$$\begin{aligned} IsSimultaneous(note_1, note_2) \quad &\rightarrow \\ (\neg(IsPassingNote(note_1) \vee IsPassingNote(note_2)) \quad &\rightarrow \\ IsConsonant(note_1, note_2)) & \end{aligned}$$

3.3. Towards Constraining the Musical Form

In both PWConstraints and Arno, the user must predefine the hierarchic structure of the score. For instance, if the user creates a piano piece with one of these systems, he must predetermine the number of voices for each section of the composition. Subsequently, the user may perform a search for the parameters of the piano piece (as durations, pitches etc.).

A more general constraint based computer aided composition system will support to search for the hierarchic structure of the score as well. This will allow the user to describe the hierarchic structure of the composition by compositional rules. By constraining the hierarchic structure the user may constrain the musical form of the composition.

4. Towards a More General Constraint Based Computer Aided Composition System

This section sketches a constraint based computer aided composition environment which will serve as a testbed for research. I will also use this environment as a platform to do actual compositions. This environment is a formal language which allows to describe aspects of music. A music description in this language is a computer program, which returns a single, multiple or all solution scores fulfilling the description.

In other words, this constraint based computer aided composition environment is a programming language, specialised in music descriptions. However, it is already a tradition in the field of computer aided composition to extend a high-level general-purpose programming language by special constructs for music composition instead of defining a new language from scratch. Recent examples of this approach are, Common Music, Haskore, Open Music, and Siren (section 2.1.3) which extend Scheme, Haskell, Common Lisp, or Smalltalk respectively.

In the remainder of the present text, the new constraint based computer aided composition programming language is called *score description language*.

4.1. The Programming Platform

The current section discusses the decision for a high-level general-purpose programming language to be the foundation of the score description language.

4.1.1. A State-Of-The-Art Constraint Programming Language

Environments for computer aided composition rely heavily on the expressive power of the underlying programming language. For the score description language, this text proposes to employ a general programming language specifically designed with constraint logic programming in mind.

Perhaps surprisingly, most constraint based computer aided composition environments available so far come with their own extra developed constraint solver, ignoring the

general constraint solvers already available. The necessary additional effort to design and implement an extra solver results in less general or less efficient solutions.

Instead of designing a new solver, the score description language employs an already existing state-of-the-art constraint logic programming language, whose expressive power outperforms the solvers of existing constraint based computer aided composition environments.

The constraint archive [Eaton and Joslin] provides an overview of existing constraint logic programming languages as well as constraint libraries for programming languages without build-in support for constraint logic programming. Particularly mature examples of constraint logic programming languages are SICStus Prolog [SICStus Prolog Home], ECLiPSe [ECLiPSe Prolog Home], and Oz [Mozart]. Non of these languages has so far been the foundation of a computer aided composition environment. Choosing one of these languages therefore means to put aside all existing computer aided composition environments and to implement all music related programming constructs from scratch. One would wish to combine an existing computer aided composition environment with an existing constraint logic programming library. One must only find a constraint library for the general programming language on which the composition environment is based.

The design of Arno (section 2.3.2) took this approach: Arno combines the composition environment Common Music with the constraint library Screamer. However, the experience with developing Arno demonstrates the priority of choosing a state-of-the-art constraint programming environment over the convenience of employing an already existing composition environment. The fundamental weaknesses of Arno – both in expressiveness and in performance – are causes by weaknesses of Screamer. Unfortunately, Screamer is inferior to state-of-the-art constraint programming environments available today.

A programming language whose constraint programming capabilities are only added later can often hardly compete with programming languages which is specifically designed for constraint logic programming. The score description language therefore extends a programming language with build-in support for constraint programming even if a computer aided composition environment for that language is missing so far.

4.1.2. A Multi-Paradigm Programming Language

Most constraint logic programming languages are descendants of Prolog. This is also the case of the above mentioned languages SICStus Prolog and ECLiPSe. As mentioned earlier, there has been little effort so far to use a special constraint programming platform for music composition, although several systems do apply constraint programming for composition. One reason for this may be the need to tightly embed constraint programming in an already existing, more general composition environment. The limitation to pure logic and constrain logic programming in Prolog may, however, be the more relevant cause.

Experiences with computer aided composition environments shows the importance of a multi-paradigm programming environment (section 2.2.3). Single paradigm programming forces the composer to solve each task in a similar way, which often makes computer aided composition harder then necessary.

Prolog lacks support for programming paradigms which have been proven highly useful for computer aided composition. A Prolog program must be a (constraint) logic program and nondeterministic search is the standard control structure in Prolog – however, one does not always need to perform a search.

Important and well established programming paradigms in computer aided composition are, for instance, object-oriented programming and functional programming. The success of the multi-paradigm programming language Common Lisp for computer aided composition may indicate the importance of supporting various programming models. Besides, the successful music programming languages SuperCollider supports multi-paradigm programming as well: SuperCollider programs can be object-oriented, functional, stateful and concurrent.

The constraint logic programming language Oz offers a programming model including declarative (i.e. functional and logic) programming, stateful programming, object-oriented programming, constraint programming, and concurrency as part of a coherent whole. The designer of the score description language therefore prefers Oz to the Prolog based constraint programming language.

4.1.3. The Oz Programming Language

The Oz programming model and language offer expressive multi-paradigm programming power. A rich set of paradigms is combined in one simple programming model with a sound theoretical foundation. Supported paradigms include functional programming, logic and constraint programming, object-oriented programming, concurrency and distributed programming [Smolka, 1995; van Roy and Haridi, 2003].

The constraint logic programming capabilities of Oz are introduced in section 4.1.4.

The Mozart Programming System is an industrial strength implementation of the Oz programming language, which is based on the Oz programming model. Mozart provides an interactive development environment for Oz which runs under the Emacs editor and comes with an extensive documentation [Mozart].

The Mozart system – the implementation of the Oz programming language and an interactive development environment – is cross-platform compatible: implementations exists for UNIX (including Linux and MacOS X) and MS Windows. Mozart is open and free software. The system is completely free of charge, including full source code and a license for all use including commercial use.

The soft real-time capabilities of Oz (together with its easy-to-use concurrency) will be useful for a composition platform. In addition, there are several ways to interface with

existing music software (e.g. composition environments or sound synthesis languages), because Oz is network aware, has an interface to the underlying system, and a C++ interfaces for developing dynamically-linked libraries. For example, OpenSound Control [Wright and Freed, 1997] (supported by SuperCollider [McCartney, 1996, 2002], Max/MSP [Zicarelli, 1998], Csound [Boulanger, 2000; Vercoe, 1985], and many more sound synthesis programs) implements sendOSC and dumpOSC as UNIX shell commands. These can easily be called form within Oz.

Oz has already been applied to music composition. The projects COMPOzE and OPPELIA [Henz et al., 1996; Zimmermann, 1998, 2001] realise automated composition (which is restricted to four-voiced music).

4.1.4. Constraint Programming in Oz

The logic and constraint logic programming capabilities of Oz subsumes both concurrent logic programming (don't care nondeterminism) and search-based logic programming (don't know nondeterminism). This text discusses only don't know nondeterminism.

The Oz programming language predefines a broad width of constraints. Examples include various numeric relations between integers or constraints on finite sets. For instance, the numeric relations

$$\begin{array}{rcl} x + y & = & z \\ x & < & y * z \\ x, y, z & & \textit{are pairwise distinct} \end{array}$$

all express constraints between the three integers x, y, z – without deterministically specifying only a single solution for them. New primitive constraints can be added to the Oz programming language via a C++ interface.

More complex constraints can be formulated by constraining the truth value of other constraints in various ways, for instance, by logical connectives as conjunction, disjunction, implication, equivalence (these constraints are called reified constraints). Other high-level means for constraint logic programming offered by Oz include first-class computation spaces, deep guards, active objects.

Search is encapsulated in Oz and the search strategy is therefore not hardwired (unlike Prolog). The user can select or define a search strategy appropriate to for an efficient search process or to find a better solution. Oz offers sophisticated control of the search process, for instance, by a graphical user interface for the interactive exploration of search trees [Peter van Roy, Per Brand, Denys Duchier, Seif Haridi, Martin Henz and Christian Schulte, 2003]. High-level means are available to implement search strategies.

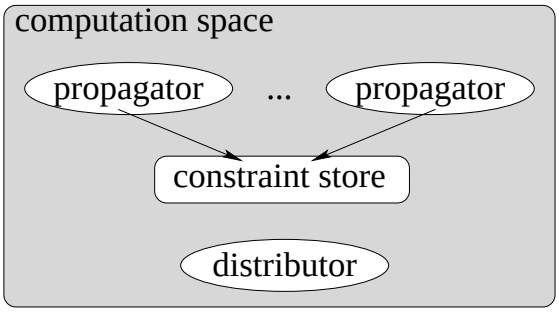


Figure 4.1.: Propagate and Distribute: Concurrent Agents in the Oz Computation Space

4.1.4.1. The Search Strategy: Propagate and Distribute

The following paragraphs introduce how search-based constraint programming is done in Oz. It concentrates on finite domain constraints [Schulte and Smolka, 2003].[1]

The constraint solver of Oz uses a technique called *propagate and distribute*. This name is based on the two inference rules of the method: *constraint propagation* and *constraint distribution*.

Computation with constraints takes place in a *computation space*. The computation space contains a *constraint store* which stores a conjunction of *basic constraints* on logic variables.[2] A basic constraint of a finite domain problem takes either the form $x = n$ (n being an integer or another variable) or $x \in D$ (D is a finite domain).

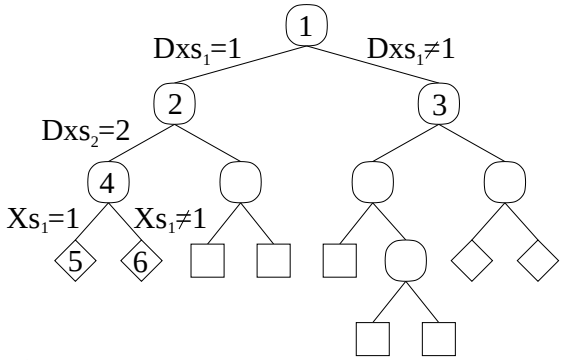
Non-basic constraints (as "$x < y$" or "all values of $x_1, \ldots, x_n$ are distinct") are imposed by propagators. Propagators are concurrent agents. A propagator will narrow a variable domain by propagating a new basic constraint, if the basic constraint is entailed by both the constraint store and the propagator. It thus reduces the search space. Nevertheless, constraint propagation does not necessarily lead to a solution (or a fail).

A *distributor* is a concurrent agent which waits until its hosting computation space S becomes stable (i.e. no further propagation is possible). It then creates two new computation spaces by executing a binary choice statement. The new spaces inherit all basic constraints and propagators of S. Additionally an arbitrary constraint C is added to one and its complement, $\neg C$, to the other space. Adding these constraints does not change the solution. It may however restart propagation. The combination of constraint propagation and distribution yields a complete solution method for finite domain constraint problems.

Distribution techniques differ in what constraint is added to which variable. In Oz, a few standard distribution strategies exist. The user can also define new strategies. The best (e.g. fastest) strategy depends on the problem. A typical distribution technique is *first*

[1] A finite domain consists of non-negative integers in Oz, which can be mapped to arbitrary discrete data.

[2] Logic variables are single-assignment variables and bound by unification as, for example, in Prolog.



Space	Xs	Dxs
1	[{0#3} ... {0#3}]	[{1#3} {1#3} {1#3}]
2	[{0#3} ... {0#3}]	[1 {2, 3} {2, 3}]
3	[{0#3} ... {0#3}]	[{2, 3} {1#3} {1#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 4.2.: all-interval series: search tree for all solutions of length 4

fail: this strategy selects respectively removes the leftmost domain value of a variable with the smallest domain.

Distribution defines the shape of the search tree. This tree can be explored by different search strategies to find a single, several, all or the best solution,[3] according to some criterion. The search can also be interactively guided by tools as, for instance, the Oz Explorer [Schulte, 2003].

4.1.4.2. All-Interval Series: the Search Strategy Demonstrated with an Example

The following paragraphs illustrate constraint propagation and distribution by computing an all-interval series.[4] The figure 4.2 shows the full search tree for all solutions of length 4. Each tree node represents a new computation space, introduced by distribution. Solved spaces are drawn as diamonds ◊, failed spaces as boxes □ and distributable spaces as circles ○.

Only 17 nodes are necessary to find all 4 solutions. 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.[5]

[3] Best solution search uses a binary procedure. After finding a solution it constrains the next solution to be better than the previous according to that relation.

[4] An all-interval series consists of distinctive pitch classes and distinctive intervals between the pitches. For simplicity, the example avoids equal pitch distances instead of equal or complementary pitch intervals.

[5] The notation $m\#n$ is used for the finite domain $m, \ldots, n$.

The constraints added by the first fail distribution strategy are shown next to the tree arcs. The strategy always affects the leftmost variable with minimal domain size. In the first space it creates a choice point by either binding the first element of `Dxs` to its first domain value or removing this value.

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

4.1.4.3. Comparison with other Search Strategies

In the three constraint based computer aided composition systems discussed earlier, constraints where implemented either as boolean predicates (PWConstraints, see section 2.3.1.3, as well as Arno, see section 2.3.2.3) or as boolean comparison (OMClouds, see section 2.3.3). In all these three systems, variable values were always first bound to a domain value and then the validity of the binding was checked.

This is fundamentally different to the constraint programming approach 'propagate and distribute', where constraints are implemented either as basic constraints or as propagators but never as a boolean function.

Propagation does not first decide (i.e. bind a variable to a value) and then check. Instead, constraint propagation reduces the variable domains without introducing any choice point and therefore reduces the total search tree – usually quite drastically. Constraint propagation, complemented by distribution, has a much better performance than a simple backtracking search, as conducted by PWConstraints and Arno.

Depending on task, the combination of constraint propagation and distribution also performs better than adaptive-search as conducted by OMClouds. For instance, a constraint which forces n variables to be pairwise distinct is hard to solve by means of a local search only. Constraint propagation is optimally suited to this task.

Like backtracking search, 'propagate and distribute' performs a full search: the approach finds a single, multiple or all solutions to a constraint problem. Adaptive-search, on the other hand, may find multiple solutions if the search is conducted multiple times. However, adaptive-search can not find all solutions.

Besides, the Oz approach offers a best-solution-search which finds the best solution according to an arbitrary user-defined criterion. Nevertheless, a best-solution-search does not need to traverse all solutions. The system constraints each solution to be better than the previous and constraint propagation reduces the search tree accordingly.

4.2. Definition of a Score Data Representation

The current section outlines some general guidelines and criteria important for the design of a generic score data representation for the score description language. Much research has been done already in the domain of score representation. The score data representation principles presented here are mainly a compilation of ideas presented in the literature and in existing implementations of computer aided composition environments. Different music representation issues and systems are discussed by Hewlett and Selfridge-Field [2001]; Selfridge-Field [1997], Dannenberg [1993], as well as Wiggins et al. [1993]. Special issues relating to time and tempo discusses Honing [2001].

A program in the score description language uses and therefore depends on the score data representation defined for the score description language. Also, a score generated by the score description language is expressed by that score data representation. In other words, the score description language can only describe and output music expressible by its score data representation. Therefore, the definition of a general and powerful score data representation is vital for the success of the score description language itself.

4.2.1. Symbolic Representation

The score representation for the score description language is a symbolic representation. This means that not the sound of the music itself is represented, but an instruction to create the sound. For example, the representation can not represent sound samples – a sound file path, on the other hand, may be part of the score. The actual sound rendering may later happen either by human musicians or by some technical means.

Most existing score representations for computer aided composition are of symbolic nature. A relatively simple example is the csound score data representation [Boulanger, 2000; csound:com; Vercoe, 1985].

4.2.2. Hierarchic Structure of the Score

When we talk about music, we rarely talk about single notes. Instead, we usually talk about groups of notes and other score elements. We talk about motives, voices, rhythmic patterns, or chords. In the literature on score representation for music composition and analysis, many authors introduce data types which contain other data and that way represent groups of score elements. Often, data can be recursively nested to form a score hierarchy (e.g. to express a note in a motive in a melody, or a note in a chord in a staff).

Most authors specify container data types which determine the start time of the contained data (e.g. elements are specified to be sequentially or simultaneously in time) [Dannenberg, 1989; Desain, 1990; Hudak; Taube, 1993]

Some authors generalise this approach to structure a score and to represent additional information. They propose further or more general container data types, which specify

various musical aspects of the contained data beside onset time (e.g. to specify grouping, harmonic information, bar structure etc.). CHARM [Harris et al., 1991; Smaill et al., 1993; Wiggins et al., 1989, 1993] offers a very general approach in this sense. A general score representation is also discussed by Balaban [1996].

4.2.3. Abstract Data Types

The score representation for the score description language subsumes the actual score data as well as means to generate, access and manipulate this data. This means, the score representation consists of abstract data types. The definition of an abstract data type defines an internal representation of the data – which can not be accessed directly – and means to generate, access and process the data. An example for a score representation consisting of abstract data types is CHARM [Harris et al., 1991; Smaill et al., 1993; Wiggins et al., 1989, 1993].

4.2.4. Class Hierarchy

A musical score contains many different symbols (e.g. in conventional music notation, notes marking pitch and timing information, articulation signs, comments etc.). Different musical styles may use different symbol sets. During the compositional process the composer may even use further symbols (e.g. roman numbers to sketch a conventional harmonic progression). The score representation specified here attempts to generalise the broad width of possible score information. Instead of implementing an enormous set of different symbols as unrelated data types, the score representation defines the score classes as a class hierarchy.

The representation employs the object-oriented programming paradigm (section 2.2.1). Subclasses inherit the features and interface functions of their super classes. This approach is already very common in the field of music representation [Agon et al., 1998; Desain and Honing, 1997; Garnett et al., 1992; Pachet, 1993; Pope, 1991].

A class hierarchy defined by object-oriented means is a representation consisting of abstract classes (section 4.2.3).

The actual class hierarchy defined for the score description language is outlined in section A.

4.2.5. Explicit vs. Procedural Representation

A computer program which generates music may itself serve as a music representation. Such a program is an example for a procedural representation. An explicit representation, on the other hand, shows not the generating program but the result of that program. An examples for a composition environment including an explicit score representation is OpenMusic 2.1.3, an example for an environment with mainly a procedural representation is Max 2.1.3.

The score representation discussed here represents music explicitly. An explicit representation has some advantages. In most cases, the explicit representation is more easy to read for humans. Editing of an explicit representation is much more easy to do than of a procedural representation. In the first case one edits the result of a computer aided composition program, while the the second case one edits the computer aided composition program itself. Finally, arbitrary information on the music can be extracted more freely from an explicit representation.

However, a purely explicit representation also has its limitations. Therefore, even conventional music notation uses quasi procedural means for the representation. For example, a repetition sign can only be read as a 'procedure', its 'argument' is the music to repeat. Also ornaments (e.g. trills) can be seen as a procedural representation: arguments to the ornament are the trill duration (i.e. the duration of the trilled note) and the pitches involved. Articulations signs change the meaning of other signs like procedures: a staccato dot shortens the sound but not the duration of the related note.

In all cases the 'procedures' can be replaced by an explicit representation. However, often the 'procedural' representation is more easy to read. For example, a note marked by a staccato sign is more easy to read than an explicit representation of the note with reduced duration and followed by a rest.

Comparable to a procedure in a computer program, a 'procedure' in conventional music notation serves as an abstractions which help to make the score more easy to read by presenting the same information in a more concise way.

To offer an equally concise representation, the score data representation described here combines explicit and procedural means. However, readability is always the main goal. Procedural descriptions are therefore only introduced as modifiers in the data structure – similar to the traditional score.

The most general way to represent procedural information within the score expresses this information as functions or procedures, in other words lambda constructs in the functional programming sense (section 2.2.1). Such an approach allows the representation of arbitrary score modifications.

Dannenberg et al. [1997] discuss the representation of procedural musical aspects (e.g. the representation of vibrato or a drum roll) by procedures.

4.2.6. Multi-Paradigm Programming

The score representation employs a multiple-paradigm programming approach (section 2.2) which makes a more concise representation possible. The main programming paradigms and their purpose in the score representation are listed in the next paragraphs.

Logic Programming: In the score representation, any entity can be represented by a logic variable (i.e. a variable bound by unification). That way, score entities can

reference other entities freely. Additionally, the representation can express scores which are only partially determined.

Functional Programming: Higher-order functional programming forms the backbone for higher-level accessors of the application programming interface to the score representation. Such accessors, for instance, collect all objects in a score which fulfil a certain test function. Functions as data are also used in the representation of certain score entities.

Stateful Programming: To make score editing possible, stateful programming is employed. However, the constraint based score generation process is completely stateless.

Object-Oriented Programming: Principles of object-oriented programming help to structure the diversity of different classes which are needed to represent a score.

Constraint Programming: Constraint programming plays the major role in the score generation process.

Some aspects of this multiple-paradigm programming approach can be found in the literature on score representation and where discussed in proceeding paragraphs. However, this compilation of paradigms for a score representation is highly unconventional, because the programming languages usually applied for computer aided composition do not allow such an approach. The programming language Oz (section 4.1.3) makes this multi-paradigm approach possible.

Some details of the actual score representation of the score description language are discussed in section A.

4.3. The Abstraction Layers of the Score Description Language

The score description language supports expressive programming constructs to describe various aspects of a musical score. The current section sketches general approaches of the score description language ordered in an increasing degree of expressiveness and abstraction.

4.3.1. Deterministic Score Description

The most basic way to describe a score is to describe it deterministically. The score data for a deterministic description is either given explicitly, is generated by a deterministic program, or the data is given by a mix of both (i.e. some score data is generated by a deterministic program, other data is given explicitly). A deterministic program can not

express a mutual dependency between values. Deterministic computer aided composition is discussed in section 2.1.2.

In the score description language, a piece can be partly or fully described deterministically. However, a pure deterministic description ignores the true expressive power of the score description language.

4.3.2. Direct Score Constraints

The score description language inherits a broad width of predefined constraints from the underlying Oz programming language (section 4.1.4). The score description language allows to specify constraints on score objects directly by using the predefined constraints. For instance, a simple voice-leading rule in counterpoint states that the interval between successive pitches shall not exceed a fifth. This constraint could be specified directly and literally between two pitches of some voice (the example was introduced in the sections 2.3.1 and 2.3.2.2):

$$\text{let } interval = pitch_{predecessor} - pitch_{successor}$$
$$(-5 < interval) \vee (interval < 5)$$

Nevertheless, to state constraints directly may lead to a bad programming style, because such style easily results in a program which is hard to read. For instance, if the users wants to state a general voice-leading rule between successive notes in a voices by direct constraint, the user must write this direct constraint explicitly for all successive notes in all voices! Instead, the user may take advantage of the more abstract approaches described in the next section.

4.3.3. Compositional Rules and their Application to the Score

Musicians and composition textbooks often speak about a compositional rule – possibly consisting of a complex set of statements or restrictions – as a single entity.

A computer science means to encapsulate multiple computations which conceptually belong together into a single entity is a *procedure*. A procedure has zero or more arguments and does some processing on these arguments. A procedure can be given an unique and self-explaining name.

In the score description language, a *compositional rule* is implemented as a procedure. That way, a (possibly very complex) rule becomes a single entity which encapsulates multiple statements. These statements impose constraints on the arguments of the rule.

For instance, the voice-leading rule example introduced above ("successive pitches shall not exceed a fifth interval") may be encapsulated in a procedure called `RestrictMelodicInterval` with two notes (or two pitches) as arguments. (By the way, a compositional

rule procedure can execute arbitrary computations including deterministic ones, it does not only need to state constraints.)

In the score description language, a compositional rule (implemented as a procedure) could be applied directly on single score objects. This approach would be comparable to the direct application of single constraints. For instance, the rule `RestrictMelodicInterval` could be applied to two single successive notes.

However, a compositional rule is usually more general. A rule shall not hold only once but for a certain subset of score objects. In the score description language, a compositional rule can be applied to multiple objects of a score (or multiple object sets, as the argument pair for the rule `RestrictMelodicInterval`) at once by *mapping* the procedure which imposes the rule on, for instance, a list of all score objects in question. Mapping is a technique of functional programming (section 2.2.1) which applies a procedure (or function) to several data entities.

Such an approach allows to freely control which subset of objects in the score shall be constrained by a certain compositional rule. For instance, the rule `RestrictMelodicInterval` may be applied to all note pairs following each other in each voice. The mechanism which selects the score objects to constrain is fully programmable. The user can define arbitrary selection mechanisms and that way define the scope of a compositional rule freely. These selection mechanisms themselves can be encapsulated into a procedure and become reusable that way.

Existing constraint based computer aided composition environments (section 2.3) allow to encapsulate compositional rules. Arno already proposes the use of procedures, respectively functions (section 2.3.2). Existing environments also offer ways to apply encapsulated compositional rules to a score. However, the expressive power of the rule application mechanisms of these environments is fixed and therefore limited. No environment offers a programmable application mechanism.

4.4. The Search Strategy

The current section sketches how a search for an arbitrary nested score can proceed in score time, even if the rhythmic structure of the score is not known beforehand [Anders, 2002]. The approach uses the finite domain constraint programming facilities of Oz (section 4.1.4.1). It defines a distribution strategy which always distributes the domain of an undetermined score element parameter with the smallest start time.

The search thus results in a non-predetermined search order. The most serious performance draw back of Arno is its naïve predetermined search order (section 2.3.2). The predetermined search order is also the reason why PWConstraints invariably uses precomputed rhythmic scores (section 2.3.1).

4.4.1. A Score Distribution Strategy

To ease the definition of new distribution techniques Oz offers special procedures to specify which constraint shall be added to which variable for distribution. Such procedure expects a specification of a distribution strategy and an Oz vector containing all data to be distributed. The distribution can be specified by functional arguments.

For our purpose the variables to be distributed are the parameter values of score elements (durations, pitches etc.). The score data representation of the score description language (section 4.2) therefore defines a special class for event parameters (as duration, pitch etc.). The parameter data structure contains both its value and a link to the element the parameter belongs to.[6] That way we can estimate which undetermined parameter belongs to a score element with smallest start-time.

The distribution strategy is now described in terms of the functions given to the arguments *filter*, *order* etc. of the Oz procedure `FD.distribute`.

Filter: Keep only parameters with an undetermined parameter value (once filtered out parameters will never be considered for distribution again).

Order: Find a parameter of which the start-time of the related element can be or is determined (e.g. the duration of the predecessor element is known) and whose start-time is minimal. Prefer durations over all other element parameters.

Select: Access the value variable of the parameter.

Value: Select a domain value (e.g. the minimum value of the domain).

4.4.2. Discussion of the Score Distribution Strategy

The proposed approach constrains only the parameters of score elements (duration, pitch etc.). The formal structure of the score needs fully be determined, for instance, the number of notes in a voice needs to be known before search.

Naturally, constraints can only be propagated between parameter values which are already know to be related. If, for example, a harmonic relation constrains pitches of notes which are simultaneous in score-time, propagation can hardly happen before these notes are known to be simultaneous.Therefore the start times and durations of score events need to be determined first.

The choice of the distribution technique is independent of other search issues in Oz. Therefore, the proposed distribution does not limit the expressiveness of an Oz constraint

[6] Of course the element data structure (e.g. a note) also needs access to their parameters. Therefore, bidirectional links are used.

script. Particularly a best solution search can still be performed and over-constraint problems can be handled using reified constraints.[7]

However, the search strategy proposed above is only an example to show how the general Oz search facilities may be adjusted to the special needs involved in searching for a hierarchically nested score. Instead of proceeding always in score-time, another approach is to first determine all timing related parameters (i.e. start times, durations). The choice of the appropriate search strategy is dependent to the search problem and the search strategy can easily be changed independent of the problem definition.

[7] Reified constraints constrain the truth value of other constraints and thus make, for instance, disjunction, implication, and negation possible. A truth value is expressed as a variable of the finite domain {0,1}, which allows to, for instance, maximise the number of fulfilled constraints.

A. The Score Data Representation of the Score Description Language

The section 4.2 introduced general guidelines of the score data representation. This section sketches the actual score representation design. The section introduces the core class hierarchy and provides a few score examples to illustrate the representation.

However, the examples only demonstrate the score representation itself. This section does not cover how to describe a score by compositional rules implemented by constraints.

For brevity, the section also omits vital aspects of the score data representation. For instances, the interface to the score representation (i.e. all functionality defined to, e.g., access score data) is left out to concentrate on the score class hierarchy.

A.1. The Class Hierarchy

The class hierarchy of the representation is implemented as a hierarchy of classes in the object-oriented programming sense. Consequently, this section uses object-oriented programming terminology. The section uses the terms *class*, *super class* and *subclass* in the meaning they have in object-oriented programming. The section uses the term *instance* for an instance of a class (this is often also called an *object*). Different programming languages differ in the terminology for the data contained in an object. The section uses the term *feature* for what is called elsewhere *instance variable*, *member variable* or data *slot*.

The core definition consists of 19 classes. The figure A.1 shows the hierarchic relations of these types as a graph (an arrow points from a super class to a subclass). Each box shows a class and its newly introduced features. A subclass also inherits all features of all its super classes.

The next paragraphs outline the purpose of each score class and its features.

Score Object (no super class, features: label, info) The most general class for score data is a score object. The feature `label` binds a symbol naming the class. The feature `info` can be used for arbitrary user information.

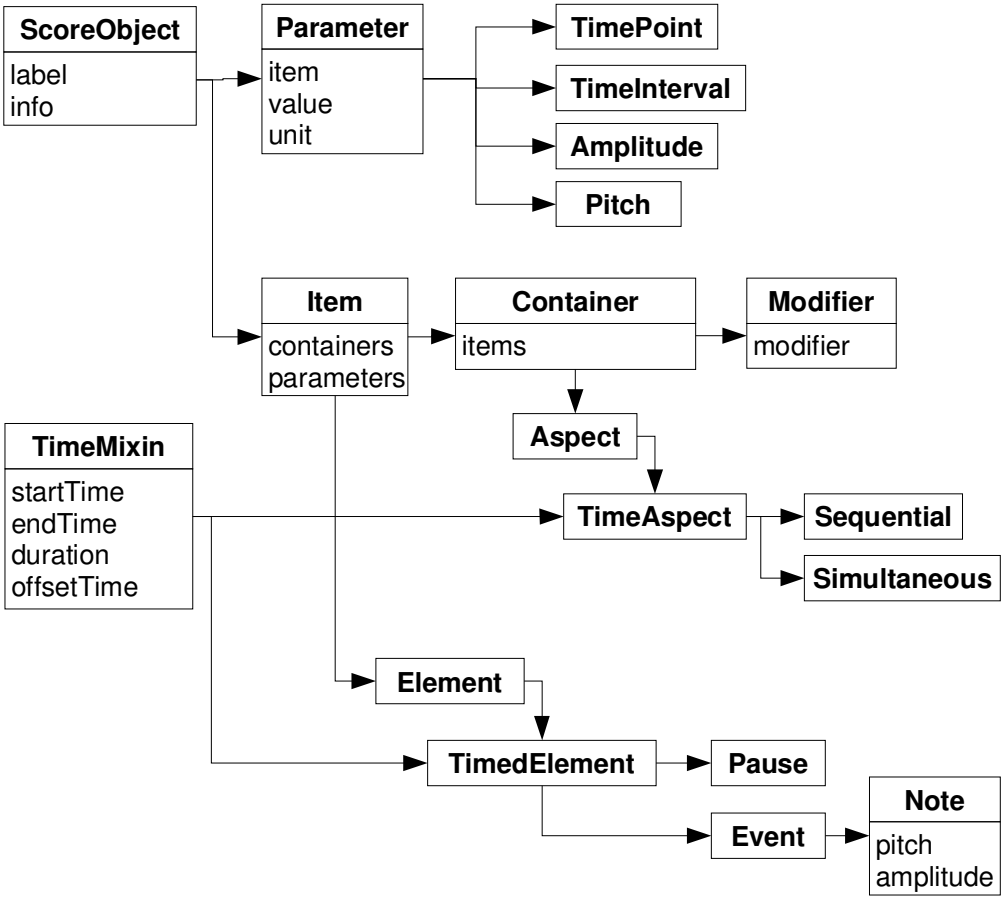


Figure A.1.: The score class hierarchy (simplified)

Parameter (super class: Score Object, additional features: item, value, unit) A parameter represents musical parameters of a single score item (explained below), for instance, the item duration. A parameter is represented by an own type (i.e. not just as a feature of a score item) to allow the expression of additional information on the parameter besides the actual parameter value. For instance, a single numeric value for a pitch is ambitious: the number could express a frequency, a MIDI-keynumber, MIDI-cents, a scale degree etc. Therefore, a parameter allows to specify the unit of measurement explicitly.

The parameter features `value` and `unit` specify the parameter setting and the unit of measurement. The feature `item` points to the score item the parameter belongs to.

Time Point (super class: Parameter)

Time Interval (super class: Parameter)

Amplitude (super class: Parameter)

Pitch (super class: Parameter)

Item (super class: Score Object, additional features: parameters, containers) An item is a generalisation of the class score container and element. An item can be contained in one or more containers, the feature `containers` points to them. The feature `parameters` binds a list with all parameters of the item.

Container (super class: Item, additional features: items) A container contains one or more score items. The feature `items` points to the items contained in a container. Because containers themselves are items as well, a container can contain other containers to form a score hierarchy of containers and elements. However, a container must not contain itself.

Aspect (super class: Container) An aspect contains one or more score items to group them and to provide additional information to them.

An aspect may be applied to express, for instance, information on grouping structure, information on the metric structure, information on the harmonic structure etc. The user may wish to define subclasses of the aspect class which provide additional features.

Time Mixin (no super class, features: startTime, endTime, duration, offsetTime) The time mixin is a super class of the classes time aspect and timed element. The time mixin class adds several timing features to these subclasses. All timing features are bound to the item feature `parameters`.

The features `startTime` and `endTime` are absolute time points. The feature `offsetTime` is a relative time interval to the `startTime` of the time aspect an item is contained in. The feature `duration` is the time interval difference of `startTime` and

`endTime` (of course, one of the three parameters `startTime`, `endTime` and `duration` is redundant – all three are represented for convenience and for performance reasons).

Time Aspect (super classes: Aspect, Time Mixin) A time aspect is an aspect which contains timing related features. A TimeAspect is a generalisation of a Sequential and a Simultaneous, which both impose timing information to the items they contain.

Sequential (super class: Time Aspect) A sequential expresses that the items contained in it follow each other in a sequential manner in time: the parameter `endTime` of a proceeding item equals the parameter `startTime` of the following item. However, setting the parameter `offsetTime` of an item to a value greater zero causes a gap (i.e. a pause) before the item and a negative `offsetTime` causes an overlap with the proceeding item.

Simultaneous (super class: Time Aspect) A simultaneous expresses that the items contained in it have all the same start time. However, setting the parameter `offsetTime` of an item to a value greater zero causes this item to delay its `startTime` the amount of `offsetTime`.

Modifier (super class: Container, additional features: modifier) A modifier contains one or more items and modifies them in some way. Conventional examples for modifiers in conventional music notation are the repetition sign, staccato sign, trill sign etc. These signs modify the music they belong to.

The feature `modifier` binds the modification itself, which is defined as a unary function. When the score is output, the modifier function is called with the value bound to the feature `items`. Arbitrary modifications can be defined by defining an appropriate modification function.

Element (super class: Item) An element is a score item which does not contain other items. For instance, a note and a pause are both elements. Also any score entity without timing information can be represented as an element. For instance, an initialisation statement for a sound synthesis language (as a f statement for csound) is such an element without timing information.

Timed Element (super classes: Element, TimeMixin) A timed element is an element with timing information. For instance, any action of a performer is a timed element – whether the action is heard or not. Some change of the instrument can be an unheard action (e.g. the action to mute an instrument), the performance of a note is a heard action.

Pause (super class: Timed Element) A pause is a score element to produce silence of a given duration. It can, for instance, be used within a sequential to produce an offset between two items in the sequential. However, in such situation a pause could also be replaced by the use of the parameter `offsetTime` of the item after the pause.

Event (super class: Timed Element) An event is a score element which produces sound when the score is played. An event is a very general representation for something producing sound. For instance, a note played on a piano (with a specific pitch, loudness etc.), a hand clapping (no pitch, but maybe a specific loudness), or an arbitrary sound synthesis language event (possibly with dozens of parameters) are all representable be an event.

To provide such generality, an event inherits the feature `parameters` which binds a list of all parameters of the event. The parameters themselves contain information about their purpose (e.g. parameters are of a certain type).

Note (super class: Event, additional features: pitch, amplitude) A note is an score event with the additional parameters pitch and amplitude.

A.2. Score Examples

The next paragraphs show score instance examples to illustrate the purpose of some score classes. For the examples, the score data – actually consisting of instances of abstract classes – is transformed into a textual representation.

The first example (figure A.2) represents a single pitch parameter. In the textual representation, each score class instance is represented by the class label, followed by a pair of round parenthesis which embrace the other features of the instance. In the example, the label is `pitch`. Each feature is shown with its respective name and value.

```
pitch('unit':keynumber value:60 item:MyNote info:_)
```

Figure A.2.: A single pitch

The example represents a MIDI pitch: its feature `unit` contains the symbol `keynumber` to indicate this fact.[1]. The pitch `value` is `60` (the MIDI keynumber for middle c). The pitch instance does belong to the note represented by the variable `MyNote`. This is indicated by the feature `item`. There is no additional information specified for the pitch, the feature `info` is bound to an anonymous variable (indicated by the character `_`).

The next example (figure A.3) shows a single note.

The note feature `parameters` contains a list of 6 different parameters. These parameters are specified by their respective class (expressed by their label). Besides, some parameters are further specified by a symbol bound to the feature `info` (another way to more precisely specify a class is to define a subclass).

The first three parameters specify the start time, the end time and the duration of the note.

[1]The textual representation uses Oz syntax. The feature name unit is enclosed in quotes to distinguish it from the Oz keyword `unit`

```
MyNote = note(info:_
              parameters:[timePoint(info:startTime item:MyNote
                                    value:0 'unit':secs)
                          timePoint(info:endTime item:MyNote
                                    value:1 'unit':secs)
                          timeInterval(info:duration item:MyNote
                                       value:1 'unit':secs)
                          timeInterval(info:offsetTime item:MyNote
                                       value:0 'unit':secs)
                          amplitude(info:_ item:MyNote
                                    value:64 'unit':midiAmplitude)
                          pitch(info:_ item:MyNote
                                value:60 'unit':keynumber)]
              containers:nil)
```

Figure A.3.: A single note

The note is not contained in any container: the feature `container` is bound to `nil`.

The example in figure A.4 shows three notes contained in a sequential container. The feature `items` of the sequential binds a list with the notes and the feature `containers` of each note binds the sequential. To simplify the example, various features and parameters are omitted.

```
Seq=sequential(
       items:[note(parameters:[timePoint(info:startTime value:0)
                               timeInterval(info:duration value:1)
                               pitch(value:60)]
                   containers:[Seq])
              note(parameters:[timePoint(info:startTime value:1)
                               timeInterval(info:duration value:1)
                               pitch(value:62)]
                   containers:[Seq])
              note(parameters:[timePoint(info:startTime value:2)
                               timeInterval(info:duration value:2)
                               pitch(value:64)]
                   containers:[Seq])]
       parameters:[timePoint(info:startTime value:0)
                   timeInterval(info:duration value:4)]
       containers:nil)
```

Figure A.4.: A single sequential container

The sequential constraints its items (which can be other containers as well) in a sequential time order. A simultaneous container constrains its items to start at the same time. Rests between items are expressed either by the parameter `offsetTime` or by explicit pause

class instances.

The example in figure A.5 expresses a sequence of chords. The example motivates the extendability of the score class hierarchy. The example first expresses the timing structure of four four-voiced chords: a sequence containing four containers of the type simultaneous, which in turn contain four notes (various features and parameters are omitted).

```
Timing=
sequential(items: [simultaneous(items:[N1=note(pitch:48 duration:1)
                                       N2=note(pitch:64 duration:1)
                                       N3=note(pitch:67 duration:1)
                                       N4=note(pitch:72 duration:1)])
                   simultaneous(items:[N5=note(pitch:53 duration:1)
                                       N6=note(pitch:62 duration:1)
                                       N7=note(pitch:69 duration:1)
                                       N8=note(pitch:72 duration:1)])
                   simultaneous(items:[N9=note(pitch:55 duration:1)
                                       N10=note(pitch:62 duration:1)
                                       N11=note(pitch:67 duration:1)
                                       N12=note(pitch:71 duration:1)])
                   simultaneous(items:[N13=note(pitch:48 duration:1)
                                       N14=note(pitch:64 duration:1)
                                       N15=note(pitch:67 duration:1)
                                       N16=note(pitch:72 duration:1)])])

ChordProgression=
region(items:
       [chord(degree:1 intervals:[1 3 5] bass:1 items:[N1 N2 N3 N4])
        chord(degree:2 intervals:[1 3 5 7] bass:3 items:[N5 N6 N7 N8])
        chord(degree:5 intervals:[1 3 5] bass:1 items:[N9 N10 N11 N12])
        chord(degree:1 intervals:[1 3 5] bass:1 items:[N13 N14 N15 N16])]
       scale:CMajor)
CMajor=scale(fundamental:1 intervals:[0 2 4 5 7 9 11])
```

Figure A.5.: A chord sequence, expressed by user-defined classes

The second part of the example expresses the harmonic structure of the chords. This part uses the two new classes chord and region – two user-defined subclasses of the aspect class. Both classes show the feature `items` of their super class container. Each new class defines also a few additional features which shall not be further explained. Another class, scale, is not based on any class of the core score class set.

The last example (figure A.6) uses the modifier class to express an articulation. A modifier is a container with the additional feature `modifier`, a function to transform the modifier items. The modifier function for an accent would, for instance, increase the amplitude of the contained element by a certain amount. In the example, the modifier function is represented by the variable `MakeAccent` (again, various features and parameters are omitted in the example).

```
modifier(info:accent
         items:note(parameters:[timePoint(info:startTime value:0)
                                timePoint(info:endTime value:1)
                                timeInterval(info:duration value:1)
                                timeInterval(info:offsetTime value:0)
                                amplitude(value:64)
                                pitch(value:60)])
         modifier:MakeAccent)
```

Figure A.6.: An accent, expressed by a modifier

Bibliography

Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

Carlos Agon, Gérard Assayag, Olivier Delerue, and Camilo Rued. Objects, Time and Constraints in OpenMusic. In *Proc. ICMC 98*, Ann Arbor, Michigan, 1998.

Adam Alpern. Techniques for Algorithmic Composition of Music, 1995.

Gloria Alvarez, Juan Francisco Diaz, Luis O. Quesada, Frank D. Valencia, Gerard Assayag, and Camilo Rueda. PiCO: A Calculus of Concurrent Constraint Objects for Musical Applications. In *ECAI 98 Workshop on Constraints for Artistic Applications*, Brighton, 1998.

Torsten Anders. Arno, über den Einsatz eines Suchalgorithmus für die musikalische Komposition. Master's thesis, Hochschule für Musik Franz Liszt, Weimar, 2000a.

Tosten Anders. Arno: Constraints Programming in Common Music. In *Proceedings of the 2000 International Computer Music Conference*, 2000b.

Tosten Anders. A wizard's aid: efficient music constraint programming with Oz. In *Proceedings of the 2002 International Computer Music Conference*, 2002.

Joe Armstrong, Mike Williams, Claes Wikström, and Robert Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, N.J., 1996.

Gérard Assayag. Computer Assisted Composition Today. In *1st Symposium on Music and Computers. Applications on Contemporary Music Creation, Esthetic and Technical aspects*, 23–25 October 1998. available at `http://www.ircam.fr/equipes/repmus/RMPapers/Corfou98/`.

Gerard Assayag, Camilo Rueda, Mikael Laurson, Carlos Agon, and Olivier Delerue. Computer Assisted Composition at IRCAM: From PatchWork to Open Music. *Computer Music Journal*, 23(3), 1999.

Mira Balaban. The Music Structures Approach to Knowledge Representation for Music Processing. *Computer Music Journal*, 1996.

Roman Bartak. on-line guide to constraints programming. `http://kti.ms.mff.cuni.cz/~bart`, 1998.

Anthony Beurivé and Myriam Desainte-Catherine. Representing Musical Hierarchies with Constraints. In *Seventh International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*, Paphos, Cyprus, 2001.

Antoine Bonnet and Camilo Rueda. *OpenMusic. Situation. version 3.* IRCAM, Paris, 3rd edition, 1999. in French.

Richard Boulanger, editor. *The Csound Book. Perspectives in Software Synthesis, Sound Desing, Signal Processing, and Programming.* The MIT Press, 2000.

Ivan Bratko. *Prolog, Programmierung für die Künstliche Intelligenz.* Addison-Wesley, Bonn, 1987.

Marc Chemillier and Charlotte Truchet. Two Musical CSPs. In *Seventh International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*, Paphos, Cyprus, 2001.

Marshall Cline. C++ FAQ lite. `http://www.parashift.com/c++-faq-lite/`.

Philippe Codognet and Daniel Diaz. Yet Another Local Search Method for Constraint Solving. In T. Walsh and C. Gomes, editors, *Proceedings of the AAAI Symposium "Using Uncertainty in Computation"*. AAAI Press, 2001.

Philippe Codognet, Daniel Diaz, and Charlotte Truchet. The Adaptive Search Method for Constraint Solving and its Application to musical CSPs. In *IWH02, International Workshop on Heuristics*, Beijing, China, 2002.

Common Music. Common Music. `http://ccrma-www.stanford.edu/software/cm/cm.html`.

csound:com. cSounds.com ... almost everything Csound. `http://www.csounds.com`.

Ben Curry, Geraint A. Wiggins, and Gillian Hayes. Representing trees with constraints. *Lecture Notes in Computer Science*, 1861:315–??, 2000. URL `citeseer.nj.nec.com/503430.html`.

Cycling74. Cycling '74 Software Products. `http://www.cycling74.com/products/`.

R. B. Dannenberg, P. Desain, and H. Honing. Programming language design for music. In G. De Poli, A. Picialli, S. T. Pope, and C. Roads, editors, *Musical Signal Processing.* Lisse: Swets & Zeitlinger, 1997. ISBN ISBN: 90-265-1483-2. available at `http://www.nici.kun.nl/mmm/abstracts/ddh-97-a.html`.

Roger B. Dannenberg. The Canon Score Language. *Computer Music Journal*, pages 47–56, 1989.

Roger B. Dannenberg. Music Representation Issues, Techniques, and Systems. *Computer Music Journal*, 1993.

Peter Desain. Lisp as a second language: functional aspects. *Perspectives of New Music*, 28(1):192–222, 1990.

Peter Desain and Henkjan Honing. CLOSe to the edge? Advanced object oriented techniques in the representation of musical knowledge. *Journal of New Music Research*, 2, 1997. available at `http://www.nici.kun.nl/mmm/abstracts/dh-97-e.html`.

Peg Eaton and David Joslin. Constraints Archive. `http://www.cs.unh.edu/ccc/archive/`.

Kemal Ebcioglu. An Expert System for Harmonizing Chorales in the Style of J.S. Bach. In Mira Balaban, Kemal Ebcioglu, and Otto Laske, editors, *Understanding Music with AI: Perspectives on Music Cognition*, chapter 12, pages 295–332. MIT Press, 1992.

ECLiPSe Prolog Home. The ECLiPSe Constraint Logic Programming System. `http://www.icparc.ic.ac.uk/eclipse/`.

Elliotte Rusty Harold. The comp.lang.java FAQ List. `http://sunsite.unc.edu/javafaq/javafaq.html`.

M. Ferrand, J. A. Leite, and A. Cardoso. Improving Optical Music Recognition by means of Abductive Constraint Logic Programming. In *Portuguese Conference on Artificial Intelligence*, pages 342–356, 1999. URL `citeseer.nj.nec.com/ferrand99improving.html`.

Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, 2dn edition, 2001.

Thom Frühwirth and Slim Abdennadher. *Constraint-Programmierung. Grundlagen und Anwendungen*. Springer, 1997.

Thom Frühwirth, Alexander Herold, Volker Küchenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, and Mark Wallance. Constraint Logic Programming. An Informal Introduction. Technical Report ECRC-93-5, European Computer-Industry Research Center (ECRC), 1993.

G. E. Garnett, J. Gomsi, C. Latta, D. Oppenheim, and S. T. Pope et al. The Smallmusic Object Kernel: A Music Representation, Description Language, and Interchange Format. Smallmusic discussion group notes, 1992.

M. Harris, A. Smaill, and G. Wiggins. Representing music symbolically, 1991. URL `citeseer.nj.nec.com/harris96representing.html`.

Martin Henz, Stefan Lauer, and Detlev Zimmermann. COMPOzE — intention-based music composition through constraint programming. In *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence*, pages 118–121, Toulouse, France, November16–19 1996. IEEE Computer Society Press.

Walter B. Hewlett and Eleanor Selfridge-Field, editors. *The Virtual Score. Representation, Retrieval, Restoration*. MIT press, 2001.

Lejaren Hiller and Leonard Isaacson. Musical Composition with a High-Speed Digital Computer. In Stephan M. Schwanauer and David A. Lewitt, editors, *Machine Models of Music*. MIT pres, 1993. reprint of original articel in Journal of Audio Engineering Society, 1958.

Henkjan Honing. From time to time: The representation of timing and tempo. *Computer Music Journal*, 35(3), Fall 2001, 2001. available at `http://www.nici.kun.nl/mmm/abstracts/mmm-1.html`.

Paul Hudak. The Haskore Computer Music System. `http://haskell.org/haskore/`.

Prolog. *Summary of Draft ISO Prolog*. ISO, 1993. Appendix to ISO/IEC JTCI SC22 WG17 N110 ("Prolog: Part I, General core").

John Peterson and Olaf Chitil. Haskell. A Purely Functional Language. `http://www.haskell.org/`.

Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, 1989.

Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

Gottfried Michael Koenig. *Ästhetische Praxis. Texte zur Musik*, volume Volume 1: 1954-1961 (1991), Volume 2: 1962-1967 (1992), Volume 3: 1968-1991 (1993), Volume 4: Supplement I (1999), Volume 5: Supplement II (2002) of *Quellentexte zur Musik des 20. Jahrhunderts*. Pfau Verlag, 1991–2002. W. Frobenius, S. Fricke, S. Konrad, R. Pfau, editors.

O. Laske. Composition theory in koenig's project one and project two. *Computer Music Journal*, 5(4), 1981.

M. Laurson. *PATCHWORK: A Visual Programming Language and some Musical Applications*. PhD thesis, Sibelius Academy, Helsinki, 1996.

Mikael Laurson and Mika Kuuskankare. PWGL: A Novel Visual Language based on Common Lisp, CLOS and OpenGL. In *Proc. ICMC 2002*, Göteborg, Sweden, 2002.

Vikas Malik. Smalltalk Frequently Asked Questions. `http://www.infi.net/~vmalik`.

Mark Kantrowitz and Barry Margolin. Answers to Frequently Asked Questions about Scheme. `http://www.cs.cmu.edu/Web/Groups/AI/html/faqs/lang/scheme/top.html`.

Kim Marriott and Peter Stuckey. *Programming with Constraints: An Introduction*. MIT Press,, 1998.

James McCartney. SuperCollider, a New Real Time Synthesis Language. In *Proceedings of the 1996 International Computer Music Conference*, San Francisco, 1996.

James McCartney. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26:4 (Winter 2002), 2002.

Eduardo Reck Miranda. *Composing Music with Computers*. Focal Press, 2001.

Mozart. The Mozart Programming System. `http://www.mozart-oz.org/`.

OpenMusic. OpenMusic. A Visual Programming Language. `http://www.ircam.fr/equipes/repmus/OpenMusic/`.

François Pachet and Olivier Delerue. MidiSpace: a Temporal Constraint-Based Music Spatializer. In *ECAI 98 Workshop on Constraints for Artistic Applications*, Brighton, 1998.

François Pachet and Pierre Roy. Musical harmonization with constraints: A survey. *Constraints Journal*, 2000.

François Pachet. An object-oriented representation of pitch-classes, intervals, scales and chords: The basic muses, 1993. URL `citeseer.nj.nec.com/pachet93objectoriented.html`.

Andreas Paepcke, editor. *Object-Oriented Programming: the CLOS Perspective*. MIT Press, 1993.

George Papadopoulos and Geraint Wiggins. Ai methods for algorithmic composition: A survey, a critical view and future prospects. In *Proceedings of the AISB'99 Symposium on Musical Creativity*, 1999.

Peter van Roy, Per Brand, Denys Duchier, Seif Haridi, Martin Henz and Christian Schulte. Logic programming in the context of multiparadigm programming: the Oz experience. *Theory and Practice of Logic Programming*, 2003. To appear,`http://www.mozart-oz.org/papers/abstracts/LogicProgrammingOzExperience.html`.

Kent Pitman. *Common Lisp HyperSpec*. The Harlequin Group, http://www.harlequin.com/.

Stephen Travis Pope. The Siren Music and Sound Package for Squeak Smalltalk. `http://www.create.ucsb.edu/Siren/index.html`.

Stephen Travis Pope, editor. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology*. MIT Press, 1991.

Stephen Travis Pope. Fifteen Years of Computer-Assisted Composition, 1995.

Miller Puckette. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*, 15/3, 1991.

Curtis Roads. *The Computer Music Tutorial*, chapter Chapter 18 "Algorithmic Composition Systems" and Chapter 19 "Representation and Strategies for Algorithmic Composition". MIT press, 1996.

R. Rowe. *Machine Musicianship*. MIT Press, Cambridge, 2001.

Robert Rowe. *Interactive Music Systems*. MIT Press, 1994.

Camilo Rueda, Magnus Lindberg, Mikael Laurson, Georges Block, and Gerard Assayag. Integrating Constraint Programming in Visual Musical Composition Languages. In *ECAI 98 Workshop on Constraints for Artistic Applications*, Brighton, 1998.

Camilo Rueda and Frank D. Valencia. Formalizing Timed Musical Processes with a Temporal Concurrent Programming Calculus. In *Seventh International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*, Paphos, Cyprus, 2001.

Örjan Sandred. *OpenMusic. RC library Tutorial. version 1.1*. IRCAM, Paris, 2nd edition, 2000a.

Örjan Sandred. *OpenMusic. RC library. version 1.1*. IRCAM, Paris, 2nd edition, 2000b.

Christian Schulte. Oz Explorer – Visual Constraint Programming Support. Technical report, Programming Systems Lab, Saarland University, Germany, 2003. available at `www.mozart-oz.org`.

Christian Schulte and Gert Smolka. Finite Domain Constraint Programming in Oz. A Tutorial. Technical report, Programming Systems Lab, Saarland University, Germany, 2003. available at `www.mozart-oz.org`.

Screamer Resposity. Screamer resposity. `www.cis.upenn.edu/~screamer-tools/home.html`.

Eleanor Selfridge-Field, editor. *Beyond MIDI. The Handbook of Musical Codes*. MIT press, 1997.

SICStus Prolog Home. SICStus Prolog. `http://www.sics.se/isl/sicstuswww/site/index.html`.

Jeffrey Mark Siskind. *Screaming Yellow Zonkers*. MIT Artificial Intelligence Laboratory, 1991.

Jeffrey Mark Siskind and David Allen McAllester. Nondeterministic lisp as a substrate for constraints logic programming. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*.

Jeffrey Mark Siskind and David Allen McAllester. Screamer: A portable efficient implementation of nondeterministic common lisp. Technical report, University of Pennsylvania Insitute for Research in Cognitive Science, 1993.

A. Smaill, G. Wiggins, and M. Harris. Hierarchical Music Representation for Composition and Analysis. *Computing and the Humanities Journal*, 1993.

Gerd Smolka. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000. Springer-Verlag, Berlin, 1995.

Standard ML. Standard ML. `http://cm.bell-labs.com/cm/cs/what/smlnj/sml.html`.

Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 2000.

Steve Summit. comp.lang.c Frequently Asked Questions. `http://www.eskimo.com/~scs/C-faq/top.html`.

SuperCollider. SuperCollider. A real time audio synthesis programming language. `http://www.audiosynth.com`.

Martin Supper. A Few Remarks on Algorithmic Composition. *Computer Music Journal*, 25(1), 2001. available at `http://muse.jhu.edu/demo/cmj/25.1supper.html`.

Heinrich Taube. Stella: Persistent Score Representation and Score Editing in Common Music. *Computer Music Journal*, 1993.

Heinrich Taube. An Introduction to Common Music. *Computer Music Journal*, 21:1: 29–34, 1997.

Heinrich Taube. *Notes from the Metalevel*. Swets & Zeitlinger Publishing, 2003. to appear.

Charlotte Truchet. [OM]Backtrack Modules. `http://www.ircam.fr/equipes/repmus/OpenMusic/Documentation/OMUserDocumentation/DocFiles/Reference/backtrack/`, a.

Charlotte Truchet. OMBacktrack Tutorial. `http://www.ircam.fr/equipes/repmus/OpenMusic/Documentation/OMUserDocumentation/DocFiles/Reference/backtracktutorial/`, b.

Charlotte Truchet, Carlos Agon, and Philippe Codognet. A Constraint Programming System for Music Composition, Preliminary Results. In *Seventh International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*, Paphos, Cyprus, 2001a.

Charlotte Truchet, Gérard Assayag, and Philippe Codognet. Visual and Adaptive Constraint Programming in Music. In *Proc. ICMC 2001*, La Habana, Cuba, 2001b.

Peter van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, To appear, `http://www.info.ucl.ac.be/~pvr/`, 2003.

B. Vercoe. The Csound Music Synthesis Language. `ftp://sound.media.mit.edu/pub/Csound`, 1985. Cambridge, MA: Media Lab, MIT.

G. Wiggins, M. Harris, and A. Smaill. Representing Music for Analysis and Composition. In *EWAIM89*, Genova, 1989.

G. Wiggins, E. Miranda, A. Smaill, and M. Harris. Surveying Musical Representation Systems: A Framework for Evaluation. *Computer Music Journal*, 1993.

M. Wright and A. Freed. OpenSound Control: A New Protocol for Communicating with Sound Synthesizers. In *Proceedings of the 1997 International Computer Music Conference*, Thessaloniki, Hellas, 1997.

Iannis Xenakis. *Formalized Music: Thought and Mathematics in Composition*. Pendragon Press, 1992. (Revised edition).

D. Zicarelli. An Extensible Real-Time Signal Processing Environment for Max. In *Proceedings of the 1998 International Computer Music Conference*, Ann Arbor, MI, 1998.

Detlev Zimmermann. Modelling Musical Structures. Aims, Limitations, and the Artist's Involvement. In *ECAI 98 Workshop on Constraints for Artistic Applications*, Brighton, 1998.

Detlev Zimmermann. Modelling Musical Structures. *Constraints*, 2001.