School of Music & Sonic Arts
Faculty of Arts, Humanities and Social Sciences
Queen's University Belfast

Thesis submission for the degree PhD in Music (Music Technology)

Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System

Torsten Anders

13th February 2007

Abstract

This research presents the design, usage, and evaluation of a highly generic music constraint system called Strasheela. Strasheela simplifies the definition of musical constraint satisfaction problems (CSP) by predefining building blocks required for such problems. At the same time, Strasheela preserves a high degree of generality and is reasonably efficient.

Strasheela is highly generic, because it is highly programmable. In particular, three fundamental components are more programmable in Strasheela compared with existing music constraint systems: the music representation, the rule application mechanism, and the search process.

Strasheela features an expressive symbolic music representation. This representation supports explicitly storing score information by sets of score objects, their attributes, and their hierarchic nesting. Any information available in the score is accessible from any object in the score and can be used to obtain derived information. The representation is complemented by the notion of variables: score information can be unknown and such information can be constrained.

This research proposes a rule formalism which combines convenience and full user control to express which score variable sets are constrained by a given rule. A rule is a first-class function. A rule application mechanism is a higher-order function. A rule application mechanism traverses the score in order to apply a given rule to variable sets. This text presents rule application mechanisms suitable for a large set of musical CSPs and reproduces important mechanisms of existing systems.

Strasheela is founded on a constraint programming model which makes the search process programmable at a high-level. The Strasheela user can optimise the search for a particular constraint satisfaction problem by programming a distribution strategy (a dynamic variable and value ordering) independent of the problem definition. Special distribution strategies for efficiently solving various musical CSPs – including complex polyphonic problems – are presented.

Acknowledgements

First and foremost I want to thank my supervisors, Michael Alcorn and Christina Anagnostopoulou, for their guidance and support throughout this research study. I am grateful to Michael for giving me the opportunity to work at the Sonic Arts Research Centre. It was a privilege and fun to work in a place with such a stimulating atmosphere. I am indebted to Christina who helped me to clarify the views presented in this thesis.

I am grateful to many people with whom I have discussed by PhD work: Stefan Bilbao, Ludger Brümmer, Darrel Conklin, Mikael Laurson, Camilo Rueda, Örjan Sandred and Alan Smaill. I have greatly benefited from their experience and constructive criticism. Kilian Sprotte was the first who dared to actually use Strasheela.

I would like to thank the Oz community. Denys Duchier answered countless questions I had when I began with this research. Christian Schulte and Raphael Collet helped me to better understand Oz' constraint programming model. Many more members of the Oz mailing list were always willing to help.

I was very happy being surrounded by so many friendly and helpful fellow PhD students: Tom Davis, David Fee, Jason Geistweidt, Katarzyna Glowicka, Banu Günel, Huss Hacıhabiboğlu, Rachel Holstead, Ravi Kuber, Christopher McClelland, Erdem Motuk, Emma Murphy, Damian Ryan, Philip Strain, Evren Tekin, and Henry Vega. I thank the Queen's University for funding my PhD research.

For detailed feedback on drafts of this work I am grateful to Christina Anagnostopoulou, Eoin Brazil, Cornelius Pöpel, and Chris Share. Chris Share also answered many questions I had on English language usage.

Finally, I would like to thank my wife Mechthild for all her support throughout these years.

Contents

1. Introduction

1.1. Computer-Aided Composition

Computers nowadays play an important role in many areas of music composition and its production. In particular, sound synthesis and effects processing plays an important role. By contrast, computer-aided composition utilises the computer in order to create symbolic scores. It focuses on higher-level musical concepts such as notes, chords, motifs, or the musical texture.

In the field of computer-aided composition (CAC, also known as algorithmic composition[1]) composers formalise their musical intentions and implement these formal specifications as computer programs. These programs output music which the composer then uses in the composition process. For writing such programs, composers use either general-purpose programming languages or special composition systems with programming support. Particularly widely-used composition systems include PatchWork [Laurson and Duthen, 1989; Laurson, 1996], the two PatchWork successors OpenMusic [Assayag and Agon, 1996; Assayag et al., 1999] and PWGL [Laurson and Kuuskankare, 2002], and Common Music [Taube, 1991, 2004]. Also the sound synthesis systems SuperCollider [McCartney, 1996, 2002] and Max [Puckette, 1991, 2002] (including its relatives jMax and Pure Data) support CAC very well. Ariza [2006] provides an extensive list of composition systems.

A rich set of composition techniques have been proposed, including mathematical models, models based on transforming existing data, and models which implement already-existing compositional strategies. Examples of mathematical models are stochastic models [Xenakis, 1992] or self-similar systems [Supper, 2001]. The compositional work of Gérard Grisey or Tristan Murail exemplify transformations of spectral analysis data into instrumental music. Koenig's software Project One and Project Two implemented concepts which stem from serial and chance composition [Laske, 1981].

Practitioners of computer-aided composition use various ways of working. In a pragmatic approach which is often applied, the composer delegates certain subtasks of the composition process to some software. The software takes over tedious subtasks, but the composer never surrenders control over the composition process itself. For example, the composer will often manually edit the output of the software and integrate it into the growing piece. Nevertheless, the delegation of composition subtasks to the computer

[1] Ariza [2005] proposes the term *computer-aided algorithmic composition* (CAAC) for this field, but his term has not yet become widely accepted.

often results in a different musical output than purely manually composed music. For example, self-similar systems often lead to highly complex but musically convincing results which are difficult to compose 'by hand'.

Systematic surveys of techniques in CAC – with historical annotations – are provided by Roads [1996], Miranda [2001], and Taube [2004]. Assayag [1998] outlines the history of computer-aided composition. Papadopoulos and Wiggins [1999] offer a systematic overview with a focus on systems based on techniques from Artificial Intelligence.

1.2. Constraint-Based Computer-Aided Composition

Rule-based approaches for CAC emulate the way in which music theories are typically expressed.[2] For centuries, compositional rules have been an established device for expressing explicit compositional knowledge. For instance, rules on how to create an organum (an early polyphonic form) are already described in the treatise Musica enchiriadis, written about 900 (usually attributed to Hucbald of St. Amand).

Because compositional rules are such a long-established concept, rule-based programming approaches have attracted much attention among composers and scholars. This approach allows a user to implement explicit compositional knowledge expressed by compositional rules in a high-level way. A computer program can then use this knowledge when generating music. The usage of rules for computational models of composition will be motivated further in Sec. 3.2.1.

Constraint programming has proven to be a particularly successful programming paradigm for realising ruled-based systems. Many compositional tasks have been addressed by constraint programming. In addition to tasks inspired by traditional music theory such as the generation of harmonic progressions [Pachet and Roy, 2001] or counterpoint [Schottstaedt, 1989; Laurson, 1996], examples include purely rhythmical tasks [Sandred, 2003], Ligeti-like textures [Laurson and Kuuskankare, 2001], or instrumentation [Laurson and Kuuskankare, 2001].

The attraction of constraint programming is easily explained. Constraint programming allows users to model complex problems in a simple way. A problem is stated by a set of *variables* (unknowns) and *constraints* (relations) between these variables. A compositional task is stated by (i) a music representation in which some musical aspects are unknown – and therefore represented by variables – and (ii) compositional rules which impose constraints on these variables. For instance, a chord can be expressed by an event list and the chord pitches can be variables. Some harmonic rules may specify how the chord pitches are related to each other. In the terminology of constraint programming, the modelled problem or task is referred to as a *constraint satisfaction problem* (CSP).

[2] In this text, the term 'rule' (and 'rule-based') primarily denotes the musical concept of a compositional rule. In particular, this term does not implicitly refer to any specific programming technique (e.g. the term does not implicitly refer to a Prolog rule [Bratko, 2001] or a condition-action rule [Russell and Norvig, 2002]). Instead, the text explains how the musical concept is realised as a programming concept in different systems.

In a solution to a CSP, every variable has a value which is consistent with all its constraints. For example, the solution of a musical CSP is a fully-determined score (or score excerpt such as a pure chord progression) consistent with all constraints expressed by the rules. Existing constraint programming systems (abridged: constraint systems) can efficiently solve a CSP – a fact which has greatly contributed to the popularity of constraint programming.

A musical CSP implements a music theory model as a computer program: when the program is executed, it generates music which complies with the modelled theory. The present text defines a music theory model as a model of musical concepts and how these concepts are related to each other. A music theory model implemented by a musical CSP must be fully formalised (e.g. fully expressible in mathematical notation). However, in the context of the present text a music theory model does not necessarily need to be consistent with any existing music. For instance, a composer may develop some musical CSP (and implicitly define a theory model) ad-hoc to generate some subpart of a new composition.

A musical CSP can always be defined 'from scratch' in a general constraint system. For instance, such a CSP can be defined in a regular programming language with support for constraint programming. However, subject-specific CSPs share a considerable amount of subject-specific knowledge: all musical CSPs require modelling of musical knowledge. For instance, concepts such as note, pitch, or voice are required in a large number of musical CSPs. Whenever a musical CSP is defined 'from scratch', all this knowledge must be modelled anew. What's more, any subject-specific optimisation of the search process must also be carried out again (if the chosen constraint system supports such optimisations at all).

Several generic music constraint systems have been proposed. A *generic music constraint system* predefines general musical knowledge and building-blocks shared by many musical CSPs and so highly simplifies the definition of such problems. For example, such a system may provide a specific music representation, templates to simplify the definition of compositional rules, or mechanisms to conveniently control how a rule is applied to the score. Examples of such systems are PWConstraints [Laurson, 1996], and Situation [Rueda et al., 1998; Bonnet and Rueda, 1999]. A pioneering system is Carla [Courtot, 1990]. Further examples include the aggregation of the music representation MusES with the constraint system BackTalk [Pachet and Roy, 1995], OMRC [Sandred, 2000a, 2003][3], Arno [Anders, 2000], OMBacktrack [Truchet, b], and OMClouds [Truchet et al., 2001, 2003]. The number of existing systems underlines the high level of interest in music constraint programming.

The constraint programming paradigm is well-suited to the needs of computer-aided composition. Composers often prefer a way of working which is situated somewhere between composing 'by hand' and formalising the composition process such that it can be

[3]OMRC is defined on top of OMCS, which in turn is a port of the PWConstraints subsystem PMC from its host composition system PatchWork [Laurson, 1996] to the descendant OpenMusic [Assayag et al., 1999].

delegated to the computer (cf. [Laske, 1981]). Constraint programming supports this way of working very well. For example, the composer can determine some aspects of the music (e.g. certain pitches) by hand and constrain other aspects by rules. Alternatively, the composer may specify the high-level structure (e.g. the formal structure) manually and let the computer fill in the details. Furthermore, composers usually do not fully formalise certain aspects of the composition process before they start composing. Instead, the formalisation is often an integral aspect of the composition process itself. A compositional task defined by means of constraint programming can be shaped in a highly flexible way during the composition process by the adding, removing and changing of individual rules.

Established composers have already made extensive use of constraint programming. These include Antoine Bonnet (e.g. for *Épitaphe* for 8 brass instruments, 2 pianos, orchestra and electro-acoustics, 1992–1994, using Situation [Bresson et al., 2005]), Magnus Lindberg (*Engine* for chamber orchestra, 1996, using PWConstraints [Rueda et al., 1998]), Georges Bloch (*Palm Sax* for seven saxophones, using Situation [Rueda et al., 1998]), Örjan Sandred (*Kalejdoskop* for clarinet, viola and piano, 1999, using OMRC, [Sandred, 2003]), Jacopo Baboni Schilingi (*Concubia nocte, in memoria di Luciano Berio* for soprano and live computer, 2003, using OMCS)[4], Johannes Kretz (*second horizon* for piano and orchestra, 2002, using both OMRC and OMCS [Kretz, 2003]), and Hans Tutschku (*Die Süsse unserer traurigen Kindheit*, music theater, 2005, using OMRC)[5].

1.3. Requirements for a Generic Music Constraint System

Generic music constraint systems differ in their degree of generality. A system is more generic if more musical CSPs can be implemented in this system. A *most generic music constraint system* would allow its user to implement any music theory model conceivable. From the perspective of the present research, such a system is an ideal system (as long as efficiency concerns are ignored). For example, composers usually prefer to make compositional decisions themselves; only with reservations do they accept a composition system which restricts their compositional freedom. A most generic music constraint system would not restrict its users to any set of specific CSP classes.

The following paragraphs state a few conditions a most generic music constraint system would have to meet. The present research does not propose such a most generic system for efficiency reasons. Nevertheless, these conditions are important as a guideline for the design of a new system.

A most generic system allows the user to leave arbitrary information about the music unknown – such unknown information is represented by variables which can be freely constrained. For example, the rhythmical structure of the music can be unknown and freely constrained in such a system (e.g. the metric structure or the duration of notes).

[4]Personal communication at PRISMA (Pedagogia e Ricerca sui Sistemi Musicali Assistiti) meeting, January 2004 at Centro Tempo Reale in Florence.

[5]Personal communication at PRISMA neeting, June 2006 at IRCAM, Paris.

Similarly, the musical texture can be constrained (e.g. whether the music is homophonic, polyphonic, or some piano-like texture where the number of voices is constantly changing, etc.). The pitch structure of the music can be unknown and constrained (e.g. the CSP implements a conventional theory of harmony, a dodecaphonic theory, or a microtonal theory of harmony). Also, the instrumentation, or sound synthesis details (e.g. envelopes for various parameters) can be unknown and constrained. A most generic system allows for an extreme case (which is only theoretical), where all information about the music is unknown in the CSP definition. The set of solutions for this extreme CSP contains any conceivable score. The user can freely constrain any unknown information in order to reduce the set of solutions as desired.

A most generic system provides access to arbitrary musical information required for the definition and application of compositional rules in order to support arbitrary musical CSPs. For example, traditional counterpoint rules require detailed information which is deduced from the information explicitly represented in common music notation. For instance, a common contrapuntal rule permits dissonant note pitches in situations where several conditions are met which involve various musical aspects: a note may be dissonant in cases where it is a passing note on an weak beat and below a certain duration. This rule requires harmonic information which is deduced from the pitches of simultaneous notes (whether a certain note is dissonant), melodic information which is deduced from the pitches of notes in the same voice (whether this note is a passing note), metric information which is deduced from the position of the note in a measure (whether this note is on an weak beat), and rhythmical information (the note's duration).

Existing generic music constraint systems, however, are designed to cover specific ranges of musical CSPs. These systems support the formalisation of certain music theory models very well, but other theory models are hard or even impossible to define. For example, OMRC is designed solely for solving rhythmical CSPs, and Situation is best suited to harmonic CSPs. PWConstraints' subsystem score-PMC is designed to solve polyphonic CSPs, but score-PMC requires a fully determined rhythmical structure in the problem definition (i.e. only note pitches can be constrained).

A most generic system not only supports rhythmical, harmonic, as well as polyphonic CSPs etc. Such a system also supports complex CSPs which constrain all these aspects (and more) at the same time. Existing systems are not well-suited for addressing such complex CSPs.

Nevertheless, to a certain extent existing systems are already generic. That is, they support a considerable number of musical CSPs. These systems are generic because the allow the user to program a CSP. In particular, existing systems support the free definition of compositional rules: compositional rules can make use of arbitrarily complex expressions.

Still, only certain aspects of a musical CSP can be programmed in these systems. Other aspects cannot be changed or these systems only offer a limited set of selectable options. For example, the music representations of many existing systems predefine common musical concepts such as notes, pitches and durations which greatly simplifies the definition

of many musical CSPs. However, the user has only limited influence on the form of these representations. Therefore, the representations are only well suited for a limited set of problems. In particular, the representations of existing systems limit the explicit score information that can be stored and what derived information can be accessed. For example, many systems provide a sequential music representation and primarily support deriving information from sets of score object which are positionally related (e.g. neighbouring notes or chords in a sequence). Access to other derived information (e.g. whether a note is on an weak beat, or whether a note is dissonant with respect to the chord expressed by its surrounding notes in a polyphonic texture) is restricted – which clearly affects the set of CSPs which can be defined in these systems.

In addition, the search strategy of existing systems (i.e. how these systems find a solution) is usually optimised for specific classes of musical CSPs. In effect, systems sometimes even purposefully restrict their users to CSPs which they can solve efficiently. For instance, score-PMC does not allow the user to constrain the temporal structure of music, because a determined temporal structure is required by the polyphonic music search approach of score-PMC to compute an efficient static search order [Laurson, 1996]. Similarly, the search strategy of Situation (which performs a consistency enforcing technique to reduce the search space) is optimised for its specific music representation format [Rueda et al., 1998].

The present research proposes a highly generic music constraint system. This system allows the user to define and solve a large set of musical CSPs and is well suited for complex musical CSPs (i.e. CSPs which depend on much musical information). At the same time, this system performs reasonably efficiently – even for problems which were hard or even impossible to solve in previous systems due to their computational complexity (e.g. polyphonic CSPs in which both the rhythmical structure and the pitch structure is constrained). The design of the system is outlined in Sec. 1.4.

This research originates from the field of computer-aided composition. Nevertheless, its results are applicable to areas outside this field. A composer can apply a generic music constraint system as an assistant in the composition process, a music theorist can use it as a testbed to evaluate a music theory, an analyst can conduct a rule-based analysis, and a teacher can demonstrate the effect of different compositional rules to students.

1.4. The Approach Taken

The present research proposes to make music constraint systems more generic by making them more programmable. This proposal is exemplified in the design of the generic music constraint system Strasheela.[6] When comparing Strasheela with previous systems,

[6]Strasheela is also the name of an amicable and stubby scarecrow in the children's novel *The Wizard of the Emerald City* by Alexander Volkov [Wolkow, 1939] in which the Russian author retells *The Wonderful Wizard of Oz* by Baum [1900]. The latter inspired the name for the programming language Oz [van Roy and Haridi, 2004], which forms the foundation for the prototype of the Strasheela

three important aspects are (more) programmable: the music representation, the rule application to the score, and the search strategy.

Strasheela's music representation aims to conveniently provide any information required to express musical CSPs. To this end, the representation is highly extendable. Representation building blocks required for many CSPs are ready-made, but Strasheela additionally predefines building blocks which assist the user in extending the representation according to their needs.

Strasheela defines a novel music representation in the spirit of CHARM (Common Hierarchical Abstract Representation for Music, [Harris et al., 1991]). Two principles have been adopted from CHARM. Like CHARM, Strasheela's representation is based on the notion of data abstraction [Abelson et al., 1985] and it supports user-controlled hierarchic nesting of score objects.

Strasheela's representation complements these principles by other principles drawn from the music representation literature, for example, selectable score parameter (music magnitude) representations [Pope, 1992] (e.g. a pitch can be represented by a key-number, cent or frequency value), bidirectional links between score objects to facilitate free traversing in the score hierarchy [Laurson, 1996], temporal containers which organise their elements sequentially or simultaneously in time [Dannenberg, 1989], organisation of musical data types in a user-extendable class hierarchy [Pope, 1991; Desain and Honing, 1997], and a highly generic data abstraction interface realised by higher-order functions [Desain, 1990].

The Strasheela user freely controls which variables in the music representation are constrained by which compositional rule. To this effect, Strasheela fully decouples the definition and application of a rule in order to make the rule application freely programmable.

Strasheela proposes the encapsulation of compositional rules in functions (actually procedures) as first-class values [Abelson et al., 1985]. This approach allows the user to define rule application mechanisms as higher-order functions expecting rules (i.e. functions) as arguments. Several rule application functions suited to many CSPs have been defined, which either reproduce rule application mechanisms of existing systems or constitute convenient novel application mechanisms. The user can easily define further rule application functions according to their needs (as shown in Sec. 6.2.4).

To be useful in practice, a constraint system must be reasonably efficient. It makes a big difference whether a CSP takes seconds or hours to solve. Strasheela is founded on a constraint programming model based on the notion of computation spaces [Schulte, 2002]. This model makes the search process itself programmable at a high-level. The programmable constraint model allows the user, for example, to optimise the search process for CSPs with a particular structure (e.g. harmonic CSPs or polyphonic CSPs)

composition system.

The scarecrow's brain consists only of bran, pins and needles. Nevertheless, he is a brilliant logician and loves to multiply four figure numbers at night. Little is yet known about his interest in music, but Strasheela is reported to sometimes dance and sing with joy.

by defining what decisions are made during search (the distribution strategy, i.e., the branching heuristics). For instance, the user can control the order in which variables are visited in the search process – depending on the information available at the time of the decision (dynamic variable ordering). This decision order influences immensely the size of the search space. Most previous systems do not allow the user to customise this. In Strasheela, such optimisations are independent of the actual problem definition, which allows the user to easily test a CSP with different search strategies or to reuse proven strategies.

Several novel score distribution strategies have been defined for Strasheela which are suitable for a large range of musical CSPs. In particular, Strasheela provides a score distribution strategy which allows the user to efficiently solve polyphonic CSPs in which both the rhythmical structure as well as other parameters (e.g. pitches) are unknown and constrained in the problem definition [Anders, 2002]. Previous systems discouraged or even disallowed the definition of such problems for efficiency reasons.

1.5. Outline

This thesis consists of four parts: a literature review, a motivation of a generic music constraint system including a description of its requirements, a presentation of a system which fulfills these requirements, and an evaluation of this system.

Literature Review: The related literature is surveyed in two separate chapters, because the presented two research fields have been relatively independent so far. Chapter 2 reviews how music can be represented in computer programs. The chapter discusses both the actual information to express, and fundamental computer science concepts used to represent this information adequately. Chapter 3 surveys the research into constraint-based music composition. After introducing the field in general, the chapter reviews generic music constraint systems which support a considerable number of musical CSPs (the present text continues the research which led to these systems).

Motivation: Chapter 4 analyses the shortcomings of existing generic music constraint systems in terms of generality (i.e. which musical CSPs cannot be solved by these systems and why), formulates the research goal based on this analysis (the development of a more generic system) and outlines how this research goal is addressed by the design of Strasheela. The analysis singles out three problematic aspects in the design of existing systems: their respective music representation, rule formalism and search strategy. Later parts of this text are often organised in accordance with these three aspects.

Strasheela: Chapters 5 to 9 introduce Strasheela in detail. The first three chapters explain Strasheela's design, that is, its music representation (Chap. 5), rule formalism

(Chap. 6) and search approach (Chap. 7). Chapter 8 demonstrates the use of Strasheela by examples and shows how the different parts of Strasheela work together. Chapter 9 introduces a Strasheela prototype and relates it to the model proposed in this text.

Discussion: The last part evaluates and discusses the results of this thesis. Chapter 10 compares Strasheela's approach with existing systems. The chapter investigates by means of general criteria the respects in which Strasheela is more generic than previous systems. Finally, Chap. 11 summarises the main contributions and suggests further developments.

Because this thesis presents a highly interdisciplinary subject, the text explains many terms, even terminology familiar in one of the related areas. In particular, many terms and concepts common in computer science are briefly defined (notational conventions are specified in App. A).

2. Survey I: Issues in Music Representation for Computer-Aided Composition

For any music system, the format and the features of its underlying music representation prove to be highly influential. For instance, a composition system can only output scores which can be expressed in terms of its music representation.

Various music representations have been proposed, as many systems come with their own novel representation. Nonetheless, different representations also share many concepts and techniques which have been identified as important and suitable. Dannenberg [1993] and Wiggins et al. [1993] both survey this research field.

This chapter reviews how music can be represented in computer programs. It discusses various issues concerning how information contained in a score is represented in an appropriate way for computer-aided composition systems. Although this chapter presents several concepts and techniques proposed by the literature, it covers only a subset of this research. For example, representations of continuous data as well as representations for real-time systems are important research subfields, but this text does not even touch on these subjects. Instead, this chapter focusses on concepts and techniques influential for the present research.

Chapter Overview

Section 2.1 introduces what a music representation is. To make the discussion more specific, Sec. 2.2 briefly presents the event-list concept as a simple but fully-fledged representation scheme. Section 2.3 comments on the complexities of representing score parameters (musical magnitudes). Various hierarchic score topologies are surveyed in Sec. 2.4.

The remaining sections review the application of fundamental programming concepts to the definition of a music representation in order to make this definition easy to extend by making extensions modular and concise. These programming concepts are first motivated by underlining the importance of an user-extensible representation (Sec. 2.5). Section 2.6 compares the idea of a textual representation with a representation based on data abstraction and lists their respective benefits and drawbacks. The concept of object-oriented programming to incrementally define a data abstraction is introduced in Sec. 2.7. Section 2.8 presents the idea of higher-order programming which can complement a data abstraction interface by highly generic functions. A brief conclusion summarises this chapter (Sec. 2.9).

2.1. Introduction

2.1.1. What is a Music Representation?

The present section explains in general terms what a music representation is. The problem of such an explanation lies in the fact that different research areas require very different music representations which are hardly compatible.

This introductory explanation therefore explains primarily what a music representation is in the context of the present research. It will relate this view to slightly different views from the literature. Section 2.1.2 further widens this perspective and positions the view on music representation of this research in the overall picture.

A *music representation* exhibits musical information. For instance, the representation expresses the information contained a printed score or knowledge how to perform such a score.

The present research mainly deals with score information. Therefore, the term *score representation* (e.g. used by Taube [1993]) could have been used instead in this text. Nevertheless, the more general term music representation is used throughout this thesis because this term is more commonly used for denoting a representation of score information.

Often, a music representation uses a representation format which is designed for this purpose. For example, a widespread representation format is common music notation (sheet music). This format uses specially devised symbols to represent information on the music such as the pitch and the duration of notes. The present survey studies music representations for computer programs.

In the context of the present research, the term music representation denotes some symbolic representation. Other authors also regard a raw waveform or sound analysis data as a music representation [Wiggins et al., 1993].

More specifically, the term music representation denotes some data structure in this text. Other authors also subsume composition systems under this term as procedural music representations [Dannenberg, 1993]. The present text would instead speak of a system which creates or processes a music representation. Still, a representation does not only consist of explicitly represented information (i.e. static information). Also derived information (i.e. information deduced from other information) plays a very important role. For example, common music notation explicitly expresses note pitches. From this information, the intervals between the pitches can be derived. Derived information is often obtained procedurally, for example, by a function call.

2.1.2. The Broader Perspective

There exists a substantial amount of literature on music representation and its related fields. Music representations have been proposed for a wide range of application areas and

the music representation literature is often influenced by specific application areas. The following paragraphs briefly outline these different application areas in order to position the present research in the overall picture using categories proposed in the literature.

Honing [1993] identifies several different research fields with specific requirements for an appropriate music representation. Honing distinguishes between two larger research fields: music analysis and production on the one hand (where representations are of a technical nature) and cognitive sciences on the other hand (with conceptual or mental representations). To the first field belong research areas such as musicology, computer music, and music publishing and to the second field areas such as artificial intelligence, computational psychology, as well as music perception and cognition. The present research clearly belongs into the field addressing music analysis and production. Therefore also this survey focusses on the literature from this area.

Wiggins et al. [1993] discuss three general tasks (in music analysis and production) for which a music representation is used and which influence its design: recording, analysis, and composition. Recording requires the representation primarily to be accurate. For analysis it is important to exploit information which is often only implicit. Finally, composition requires a particular flexible representation. All three tasks are important for the present research, but the importance increases from the first to the third task.

A music representation stores information of different musical aspects. The Standard Music Description Language (SMDL, [Sloan, 1997]) differentiates between four basic domains: the logical domain (abstract information), gestural domain (performance information), visual domain (graphical information), and analytical domain (bibliographic and interpretative information). For the present research, the logical domain and the analytical domain are of primary importance.

2.1.3. Terminology

The music representation literature introduces a large number of terms. Unfortunately, so far there exists no generally accepted terminology. Different representations introduce different terms for similar concepts. For example, a score data object which contains and groups other data objects is either called a 'constituent' (in CHARM, [Harris et al., 1991]), a 'structured' (in Expresso, [Desain and Honing, 1997]), a 'collection', a 'holder' or a 'container' (in different versions of Common Music, e.g., [Taube, 1993]).

To allow comparison of different representations in a uniform way, this text aims to use a consistent terminology. For example, this text uses the term *container* for the above-mentioned concept. Of course, the constructs of the different systems modelling this concept are not truly equivalent and therefore the text mentions the original terminology of each representation as well.

2.2. A Basic Representation: Event List

A fairly basic music representation is the *event list*, where the score consists of a flat list of events. An *event* is a score object[1] which produces sound. An event features a set of *parameters* – the basic magnitudes in a music representation – such as start time, duration or pitch.[2]

Event lists have been highly successful, especially for sound synthesis. This can be seen in languages in the Music-N tradition such as Csound [Boulanger, 2000]. Furthermore, also more advanced systems such as Siren/SmOKe [Pope, 1992] or SuperCollider [McCartney, 2002] still support an event list.

2.3. Parameter Representation

The seemingly simple parameter concept has in fact many facets. Often, parameters can be specified in several ways, for instance as an acoustic property (e.g. amplitude), a perceptual property (e.g. loudness), a performance property (MIDI velocity), or a music notation score symbol (e.g. a dynamic symbol such as *f* to denote *forte*). In practice, however, these clearly distinct properties are often blurred. For example, in a sound synthesis context the term amplitude is often used to denote loudness. Parameters are represented symbolically or numerically (e.g. symbolic pitch name or numeric key number) and are measured in an absolute or relative way (e.g. an absolute duration in seconds or a relative duration in beats complemented by a tempo specification).

Some systems define only a single representation form for each score parameter. In the internal music representation of the visual-programming composition system PatchWork [Laurson, 1996] and its successor OpenMusic [Assayag et al., 1999], for instance, note parameters are always specified as integers in the following way: the absolute note duration is measured in milliseconds, the pitch in midicent (a combination of the MIDI pitch number and cent values, i.e. middle *c* is represented by 6000) and the loudness is given as a MIDI velocity [Agon et al., 2001]. This uniform representation simplifies the implementation of the graphical music editors which are bundled with the representations of these systems.

Other systems allow the user to choose from a predefined set of supported representation forms. In the composition system Common Music, parameter values can be specified in several ways. A note pitch, for instance, is either given as a symbolic note name, a MIDI key-number, or a frequency in Hertz [Taube, 2004].

Still, none of these representation forms does represent the enharmonic spelling (i.e. the representation does not differ between $c\sharp$ and $d\flat$). To address this issue, some systems

[1] The term *score object* is used to denote anything in a score (e.g. a pitch, a note, or whole score) which is represented by a single – but possibly compound – value. Later, in the context of object-oriented programming (Sec. 2.7) this term is used in a more specific meaning.

[2] Following the terminology in composition and sound synthesis, this text uses the term *parameter* instead of the terms *feature* or *attribute* used elsewhere.

represent the pitch in a composite way. For example, CHARM (Common Hierarchical Abstract Representation for Music, [Harris et al., 1991]) represents a pitch by three components: its note name, accidental and octave. The important advantage of such a representation is the ability to correctly spell accidentals for western tonal harmony. Unfortunately, such a representation is also limited to western music (i.e. non-western or microtonal music is excluded).

Other systems express the rhythmic structure in a more elaborated manner. A highly expressive example is the RTM-notation [Laurson, 1996], featured by PatchWork and by OpenMusic (in slightly incompatible versions). RTM-notation represents relative durations in a tree, where each subtree consists of a proportional duration value and optional children subtrees. The sum of the proportional durations of all children specifies how the proportional duration of their parent tree is subdivided and thus what the proportional duration of each child means. For instance, a triplet is represented by the proportional duration of the whole triplet and three children of equal duration – which optionally are further subdivided in smaller rhythmic values by further subtrees. RTM-notation expresses pauses and ties by special data types. Usually, all durations are expressed by positive integers. However, pauses are expressed by negative integers and a note tied to its predecessor is expressed by a float. The RTM-notation reflects clearly the hierarchic structure of rhythmic subdivisions, especially for highly complex 'embedded' rhythms. On the other hand, this notation is relatively hard to read. ENP-score-notation (Expressive Notation Package, [Laurson and Kuuskankare, 2003]) – supported by PWGL (PatchWork with OpenGL, another successor of PatchWork [Laurson and Kuuskankare, 2002]) – extends the purely rhythmical RTM representation to a full music notation representation which also contains information such as note pitch, instrumentation and expressions.

2.4. Topologies of Hierarchic Representations

When musicians talk about music, they rarely talk about single notes. Instead, they talk about groups of notes like motives, voices, rhythmic patterns, or chords. Furthermore, it is widely agreed that such structural objects can be hierarchically nested, for instance, in a hierarchic harmonic structure as in Schenkerian analysis [Forte and Gilbert, 1982] or in an hierarchic grouping structure [Lerdahl and Jackendoff, 1983].

The benefits of a structured representation are obvious for the purpose of music composition and analysis. Even for sound synthesis – where event-list-like music representations are successfully applied since decades (Sec. 2.2) – a structured representation can be beneficial. For example, in a Csound score it is tricky to apply an amplitude envelope to a group of notes [Dannenberg, 1993]. Such an envelope is more easily expressed in the MIDI protocol using MIDI controller messages. Still, the group of notes and the envelope are not linked in the MIDI format: when a MIDI file is edited (e.g. in a MIDI sequencer) the user has to maintain this relation manually (e.g. by first moving the notes along the time axis and then the controller data).

Consequently, many researchers studied music representations in which a set of score objects (e.g. notes) is somehow grouped and this grouping is explicitly represented.

Many music representations allow the user to explicitly group a set of score objects by collecting all objects of this set in some higher-level object. Several different terms have been proposed for such an higher-level object grouping other score objects (Sec. 2.1.3 already listed several examples when discussing the differences of music representation terminology in various systems). In general, this text uses the term score *container*, but when discussing a particular system also its respective terminology will be mentioned.

The present section discusses the different approaches to hierarchical music representations found in the literature by sorting them into different score topologies. Some representations define two-dimensional lattice-like representations (Sec. 2.4.1), others trees which either mirror the hierarchic layers of traditional music notation (Sec. 2.4.2) or define a set of generic containers which can be arbitrarily nested in a tree (Sec. 2.4.3). Finally, some systems represent music in an acyclic graph (Sec. 2.4.4).

The Score Context Concept

The present research introduces the term *score context* to denote an arbitrary set of score objects which are inter-related.[3] A context is defined by specifying how its elements are inter-related. For example, all objects (e.g. notes and pauses) which belong to a voice denote a score context. Figure 2.1 depicts this context for the voice *myVoice* in a formal way using the common set-builder notation. A container can be used to express such a score context explicitly.

$$\{x : x \text{ is part of } myVoice\}$$

Figure 2.1.: First score context example: the set of all objects belonging to the voice *myVoice*

However, in this thesis the term score context is more general. It does not only denote a set of score objects which are all contained in the same object (e.g. a voice, a measure etc). Instead, a score context is an arbitrary set of score objects. Thus, another score context example is the set of notes which all sound at a certain time point t (see Fig. 2.2).

$$\{myNote : isNote(myNote) \wedge start_{myNote} \leq t \leq end_{myNote}\}$$

Figure 2.2.: Second score context example: the set of all notes which sound at time t

[3] The term score context is inspired by Lilypond [Nienhuys and Nieuwenhuizen, 2003] and PWConstraints [Laurson, 1996], however the term is redefined it here with a much more general meaning.

2.4.1. Two-Dimensional Representation

In `**kern` – the representation of Humdrum [Selfridge-Field, 1997; Huron, 2002] – music is organised such that the two 'musical dimensions' time and instrument part are represented in a two-dimensional way. The figure 2.4 shows the topology of the musical example in Fig. 2.3 using such a two-dimensional representation.

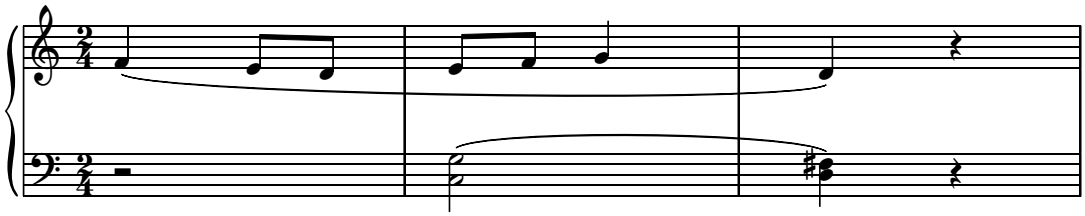


Figure 2.3.: Béla Bartók. Mikrokosmos, No. 87, beginning

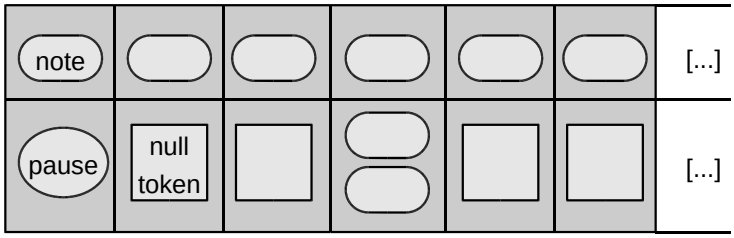


Figure 2.4.: Representation of the Mikrokosmos example (Fig. 2.3) by a two-dimensional lattice

The `**kern` representation uses a purely textual (ASCII) format. Notes and other score objects simultaneous in time are represented in horizontal lines (called *records*) whereas time is represented in vertical columns (called *spines*). Thus, compared with common music notation or the graphical representation of the `**kern` topology above, the horizontal and the vertical dimensions are swapped. By this practise, `**kern` avoids the line breaks of common music notation.

The two-dimensional representation requires some transformation of the musical data. Although the duration of records may differ (or have no length at all such as clef declarations), each field in the two-dimensional representation must be specified. Therefore, in case a few shorter notes run in parallel with a longer note, it is important that the persistence of the longer note is explicitly denoted by a null token (see Fig. 2.4).

It can also be seen in in Fig. 2.4, that the data in a single spine is not necessarily a flat sequence: the two notes of the chord in the second measure are placed in a single field. Furthermore, the representation may also split the actual note and specific note parameters (e.g. fingering information or lyrics) in different spines.

It should be noted that `**kern` explicitly represents information necessary to access two important score contexts: the set of score objects (e.g. notes or pauses) in a part

(including their order in the part) and the set of simultaneous score objects at any time point in the score (including which object belongs to which part). In case some simultaneous object is a null-token, then its preceding object is simultaneous.

2.4.2. Representation with Specialised Containers in a Tree

Several representations explicitly embody hierarchic components inspired by music notation. For example, these representations explicitly express notation concepts such as the whole score, a system, a staff, a voice, a measure, a chord, a pause, or a note by their own score object. These score objects are often organised in a tree as shown in Fig. 2.5.

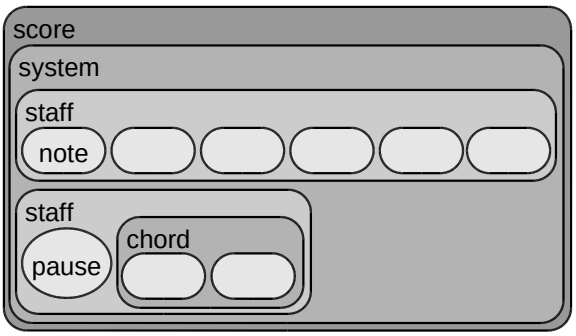


Figure 2.5.: Representation of the Mikrokosmos example (Fig. 2.3) by a tree of specialised containers (the types of this example are inspired by Common Music Notation)

For self-evident reasons, music notation systems in particular favour this representation form. Examples include textual music notation languages (e.g. Common Music Notation [Schottstaedt, 1997], Lilypond [Nienhuys and Nieuwenhuizen, 2003], and MusicXML [Recordare LLC, 2006]), as well as systems which closely integrate music notation (e.g. PWGL [Laurson and Kuuskankare, 2002] which makes use of ENP-score-notation [Laurson and Kuuskankare, 2003]).

The actual structure supported depends on the respective system. For example, Fig. 2.5 depicts the structure of a Common Music Notation score. A score in ENP, on the other hands, is structured differently and consists of parts, which in turn consist of voices, which consists of measures, which consists of beats in RTM-notation (Sec. 2.3) which finally rhythmically position score events.

Music notation systems often predefine a large number of different containers, to have a corresponding container for every concept of nesting in common music notation. Often, the hierarchic nesting is fixed (e.g. all staffs are contained in the top-level score object or in system objects). In other cases, the hierarchic nesting is fixed, but levels are optional – if not explicitly specified, the system will automatically create the respective level with default settings.

The representation of specialised containers in a tree explicitly represents information required to access a number score contexts. In particular, hierarchic relations are stored

explicitly (e.g. which note belongs to which part). Also the order of score objects (e.g. the order of notes in a part), and their type (e.g. object x is a note) is explicitly represented. Some systems allow the user to explicitly encode further groupings, for example, by marking the start and end of an analysis bracket, a slur, or a crescendo hairpin. Yet, the context of simultaneous score objects is not explicitly represented – in contrast to the two-dimensional score topology (Sec. 2.4.1).

2.4.3. Representation with Generic Containers in a Tree

The representation of each hierarchical musical concept (e.g. chord, voice, part or system) by a special score object type results in a large number of types. For applications beyond music notation, a large number of types is often not desirable and a more generic representation scheme is preferred.

Nested Event-Lists

A radically simplified hierarchic representation scheme (when compared with the topology of specialised score objects) extends the event list representation (Sec. 2.2) by the notion of a single container type. This container contains either events or other containers. Each contained score object features both an explicit start time and a duration to define the temporal structure within the container. Examples for such temporal containers – which allow for nested event lists – are the `EventList` in SmOKe [Pope, 1992], the `seq`-object in Common Music since version 2 [Taube, 2004], and the Maquette in OpenMusic [Agon et al., 1998].

Nested Temporal Containers

A related representation scheme complements the notion of events by *two* different generic temporal container types. The first container organises its contained score objects sequentially in time and the second organises them simultaneously in time. Contained objects can be either events or other temporal containers – the resulting structure is thus again a tree (Fig. 2.6).

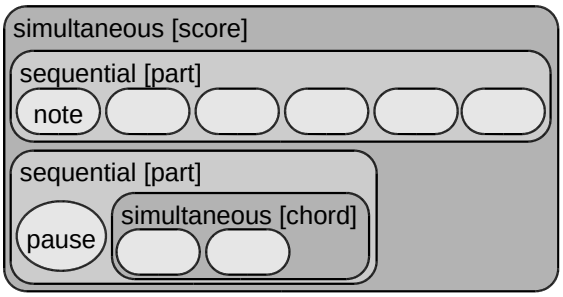


Figure 2.6.: Representation of the Mikrokosmos example (Fig. 2.3) by tree of generic temporal containers

This second representation with generic containers can be more concise than the nested event-list representation, because only the duration of each score object must be specified explicitly – the start time of each object is implicitly specified by its position in a container (e.g. by the start time and duration of its predecessor in a sequential container).

Several systems apply a representation scheme in which sequential and simultaneous arrangements of score objects can be nested freely. Examples include the functional playback-time control language Arctic [Dannenberg, 1984] (and the continuative work such as the score language Canon [Dannenberg, 1989] and the synthesis language Nyquist [Dannenberg, 1997]), the visual composition system LOCO [Desain and Honing, 1988], the representation framework Music Structures [Balaban, 1996], as well as the CAC systems Haskore [Hudak], and Common Music (the now obsolete version 1) [Taube, 1993]. Also some music notation systems offer this generic representation form as a complement for specialised containers (e.g. Lilypond).[4]

Desain and Honing [1997] point out that using this generic representation, the same music can be represented in different ways to explicitly represent different musical information. For example, the score topology in Fig. 2.6 expresses the sequence of notes which constitute a part. The topology of Fig. 2.7, on the other hand, expresses which section of a part belongs to a certain measure. A user-defined nesting of generic temporal containers thus allows the user to define which information is represented.

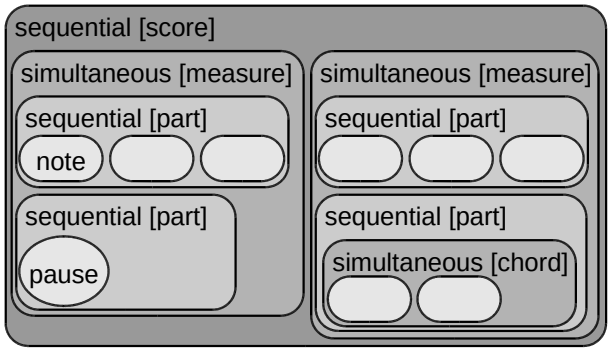


Figure 2.7.: Representation of the Mikrokosmos example (Fig. 2.3) by an alternative tree of generic temporal containers (cf. Fig. 2.6 using the same temporal containers to express different hierarchic structures and thus different score contexts)

2.4.4. Representation with Containers in an Acyclic Graph

The nesting of generic temporal containers in different ways in a tree allows for the representation of different information. Some researchers generalised this approach and allow the user to subsume all this information in a single representation (Fig. 2.8).

[4]Some representations (e.g. Arctic, Music Structures and Haskore) introduce operations instead of special containers to arrange score objects sequentially or simultaneously in time.

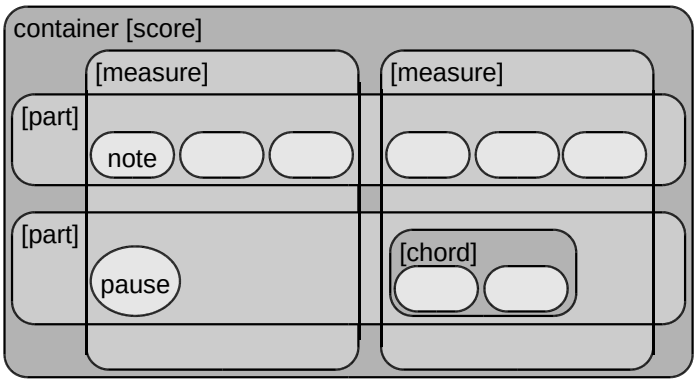


Figure 2.8.: Representation of the Mikrokosmos example (Fig. 2.3) by a music representation consisting of multiple orthogonal containers which form an acyclic graph. This representation subsumes the score contexts expressed in Fig. 2.6 and in Fig. 2.7

For example, CHARM [Wiggins et al., 1993, 1989; Harris et al., 1991; Smaill et al., 1993] supports such a score topology. As previous systems, CHARM defines events and containers (called constituents in CHARM). However, each score object in CHARM features a unique identifier. The objects grouped in a container (the particles in CHARM terminology) are referenced by their identifier. This design decision makes it possible to incorporate an object in multiple containers at the same time. However, no constituent may be contained in itself (neither directly nor indirectly). The resulting topology thus forms a directed acyclic graph.

In such a score topology, the multiple 'dimensions' of nested events and containers can express various information. In the example in Fig. 2.8, one 'dimension' expresses the sequence of objects in a part, the other the set of objects in a measure. Other examples include grouping, harmonic information, or motif boundaries.

Each of these 'dimensions' could even be hierarchically nested. For example, the metric 'dimension' could nest notes recursively in beats, measures, and higher-level accent groups. The harmonic 'dimension', on the other hand, could represent prolongational trees created by Schenkerian analysis.

2.4.5. Summary

This survey introduced several score topologies. The following topologies have been reviewed.

- Event list (Sec. 2.2)
- Two-dimensional representation (Sec. 2.4.1)
- Tree with specialised containers (Sec. 2.4.2)

- Nested event-lists (Sec. 2.4.3)
- Tree with two generic temporal containers (Sec. 2.4.3)
- Acyclic graph (Sec. 2.4.4)

Some of these topologies generalise other topologies, some topologies stand on their own. For example, the pure event list is generalised by the nested event list which in turn is generalised in the acyclic graph representation. The two-dimensional representation, however, is not related with any other representation.

The topology of a hierarchic music representation influences which musical information is represented explicitly. For example, in the two-dimensional representation shown in Fig. 2.4 the contexts 'sequence of notes in a system' as well as 'simultaneous notes' are explicitly represented. Much further information can be represented in this topology (as demonstrated by the `**kern` representation), but some information can only be expressed implicitly. In particular, hierarchic nesting cannot be expressed explicitly.[5]

Different topologies express different information explicitly. The tree of temporal containers depicted in Fig. 2.6 expresses hierarchic nesting (e.g. some note is contained a chord, which is contained in a part). This information is not expressed explicitly in the two-dimensional representation. However, the hierarchic representation cannot explicitly represent the two contexts 'sequence of notes in a system' as well as 'simultaneous notes' at the same time. Only accidentally, in the second measure of the example every note of the upper part is simultaneous with the lower part notes. In case of different note start times in the lower part, only a procedural comparison of the start and end times of all notes would reveal which notes are actually simultaneous and which are not. The two-dimensional representation, on the other hand, does represent both these contexts explicitly.

The acyclic graph representation can potentially express any set of related score objects explicitly. However, for a large number of score contexts to represent, such an explicit representation can become rather laborious.

Musical information which is only implicitly expressed can still be accessed conveniently when the representation is designed accordingly. For instance, the data abstraction concept (presented in the following Sec. 2.6) makes this possible. Also, higher-order programming (Sec. 2.8) can greatly facilitate accessing various derived information in a concise way.

2.5. Extensibility

It is impossible for the designer of a music representation to foresee all future uses of the representation. For instance, composers are constantly developing novel composition and

[5] The `**kern` representation allows for special markings to denote, for example, nested slurs and phrases. Still, this information is expressed by special symbols and not by explicit nesting.

instrument playing techniques. Furthermore, scores for novel sound synthesis techniques have to reflect the synthesis parameters, and research is developing new insights on music (e.g. new music theory concepts). For these reasons, a music representation cannot be fixed but requires that it can grow with its use.

Computer music has a tradition of highly extensible music representations. For instance, important for the success of the musicology research toolkit Humdrum was the fact that the Humdrum user can easily define new Humdrum representations according to her/his very specific needs (e.g. for Bugandan xylophone music, Schenkerian graphs, or dance steps) [Huron, 2002]. Furthermore, Music-N inspired languages such as Csound allow the user to extend the number of parameters per event and to interpret the meaning of the additional parameters.

A counter-example is the MIDI protocol which strictly limits the format and number of parameters of note events to start time, key-number (integers in $[0, 127]$), velocity ($[0, 127]$), channel ($[0, 15]$) and end time. In case the user wants to apply MIDI events for a purpose not pre-conceived in the representation design (e.g. to represent microtonal music with more than 12 pitches per octave) the user cannot adjust or extend the representation but has to invent some kludge which works around to problem (e.g. adjusting the tuning of different MIDI channels with the same sound program and distribute note events on different channels accordingly instead of simply extending the pitch resolution).

Particularly flexible representations are often integrated in a programming environment where the composer can extend and adapt the representation by programming. Examples are the representations of composition systems such as Common Music, OpenMusic, PWGL, or Haskore.

In such a a programming environment, a representation is more easily extended in case extensions can be modular and concise. An extension is modular if the user can develop such an extension without changing the original program which implements the representation. For example, some user may need to derive information which is not supported so far (e.g. deduce a symbolic chord description from a set of notes). A modular extension for such derived information does not need to change the original program itself but only adds a definition.

An extension is concise if it does not require duplicating functionality or repeating code which is already part of the original program. For example, some user may need to define a new score object type. If such an extension defines the new type as a variant of an already exiting type where only the differences to the exiting type require definition, then this extension is defined concisely.

The following sections discuss how the application of fundamental programming concepts can make a representation highly extendable by making extensions modular and concise. Concision and modularity is facilitated by the following concepts: data abstraction (Sec. 2.6), object-oriented programming (Sec. 2.7), and higher-order programming (Sec. 2.8).

2.6. Textual Representation vs. Data Abstraction

Textual Representations

Many music representations proposed in the literature are textual representations (i.e. use ASCII code). For example, the Csound score is represented textually. Many more textual representations are discussed by Selfridge-Field [1997].

Raymond [2003] explains the benefits of textual representations in general compared with binary representations: "Text streams are a valuable universal format because they're easy for human beings to read, write, and edit without specialized tools." Moreover, text-files are easily read and written by other programs and thus facilitate cooperation between independent programs. Binary data formats are only suitable for large data sets which require compression (and even in this case it is important to keep in mind that textual data compress very well with standard compression tools such as gzip). Huron [2002] advocates textual music representation formats (e.g. used by Humdrum) for similar reasons.

Raymond and Huron recommend textual formats primarily for exchanging data between applications. One could conclude that it would also be a good idea to base the internal representation of an application on a textual format, that is a format using data which have a literal representation in a programming language. For instance, numbers, symbols and lists are literally represented in Lisp.

However, using a textual representation directly inside an application also poses limitations: such a representation only shows its explicitly encoded information. For instance, it may be necessary to represent note parameters in different formats: pitches may be represented either as MIDI key-numbers or as frequency values (cf. Sec. 2.3). Yet, an application part (e.g. some user-function) which uses a textual representation directly and expects pitches encoded as key-numbers would require some changes whenever the format of the representation changes to frequency values. If multiple parts of an application depend on the format of the textual representation, every part would require changes – which can prove extremely hard and error-prone.

Moreover, it is often necessary to represent the same musical information in different ways for different purposes. Some application part using the representation may require note pitches represented by absolute pitches, whereas another requires the intervals between these pitches, and still another some symbolic specification of the chord formed by these pitches. It would be convenient, if the music representation could provide such diverse viewpoints for its musical information consistently even if the internal representation changes (e.g. regardless whether pitches are represented by key-numbers or frequencies).

Data Abstraction

Computer science suggests the concept of *data abstraction* [Abelson et al., 1985, Chap. 2] to address such requirements. The main idea here is to define an *abstract data type* (ADT) which encapsulates a set of values into a single datum and defines a set of operations

on this datum. The set of all operations is called the ADT *interface*. The values are encapsulated such that only the operations of the interface can process them.

Typically, the interface consists of operations such as a constructor to create a datum, a destructor[6], accessors (also known as selectors) and modifiers (also known as mutators or setters) to access and change the encapsulated data, operators for type-checking, equality checking and arbitrary special purpose operators. For instance, if a system defines a note as an ADT, it defines a note constructor, accessors and setters to access and change note parameters (such as duration or pitch), an operator to check whether some datum is a note et cetera.

When a system uses the data only via its interface, the actual implementation of the data can change as long as the interface remains the same. For instance, some special note pitch accessors *getKeynumber* and *getFrequency* may implicitly transform the pitch representation and consistently return either MIDI key-numbers or frequencies – regardless of the internal format.

Moreover, the interface can abstract whatever information can be derived from the explicitly stored information. When procedurally derived information is wrapped in an interface function, it can be accessed as conveniently as explicitly stored information. For instance, Sec. 2.4.5 showed how different score topologies express different information explicitly whereas other information can only be derived. The representation based on generic temporal containers (Sec. 2.4.3) can only explicitly represent one of the two contexts 'sequence of notes in a system' and 'simultaneous notes' at a time. Yet, the missing context can be derived, for instance, by traversing the whole score hierarchy to search for the score objects which belong to the missing context (this idea is discussed further in Sec. 2.8). In most cases, it can even be neglected that such derived information takes longer to retrieve.

Many systems design their music representations according to data abstraction principles. For example, CHARM [Harris et al., 1991] defines ADTs for parameters, events and constituents (i.e. containers). Moreover, CHARM defines special arithmetic operations for parameters such as time (i.e. a time point) and duration (i.e. an interval between time points): adding a time and a duration results in a time, whereas adding two durations results in a duration. MusES [Pachet, 1993] introduces a rich algebra of pitch classes where pitch classes – with up to two accidentals – are represented as data and accidentals serve as operations on the pitch data.

Some systems combine the benefits of data abstraction with the benefits of a textual representation. For instance, the ENP-score-notation [Laurson and Kuuskankare, 2003] is a textual input and output format for the internal representation of the graphical score editor of PWGL.

[6] Many computer-aided composition and analysis systems are based on a high-level programming language which supports garbage collection and therefore makes a deconstructor superfluous.

2.7. Class Hierarchy

A musical score contains data entities of many different data types. For instance, a music representation modelling conventional music notation may require the following types: notes marking pitch and timing information, articulation signs, and staves to organise notes in voices. Different musical styles may use different data type sets. During the compositional process the composer may even introduce further types (e.g. roman numbers to sketch a harmonic progression).

Existing music representations often generalise this broad width of possible score information. Instead of implementing an enormous set of different types in an unrelated way, many representations define musical data types as instances of classes organised in a class hierarchy in the object-oriented programming sense.

In the *object-oriented programming* (OOP) paradigm [Booch, 1991; Rumbaugh et al., 1991; Jacobson, 1992], the user defines an abstract data type (Sec. 2.6) by defining a *class* where a class *instance* constitutes the actual data entity (more precisely, a class instance has a type which the user defined by defining the class). A class instance is often also called an *object*. Typically, a class definition defines class *attributes* (data encapsulated in a class instance) and class *methods* (functions or procedures defining the data abstraction interface).[7]

OOP languages differ in their terminology: alternative terms for a class attribute are, for example, instance variable, member variable, or slot. Another term for methods are *messages* to stress the fact that objects communicate with each other.

OOP extends the notion of ADTs by the concept of inheritance. Multiple ADTs have often much in common. For instance, note classes for different output formats may share attributes and methods. OOP allows incrementally defined ADTs [van Roy and Haridi, 2004]: the shared attributes and methods are defined only once in a more general superclass from which a subclass *inherits*. Inheritance is often used to model an *is-a* relation between instances of a subclass and a superclass, whereas attributes often model an *has-a* relation between an object and the data at its attributes.

Polymorphism is a general data abstraction concept which is important to structure programs in a maintainable way. *Polymorphic* data types are different data types which (partly) define the same data abstraction interface but 'respond' in different ways. For example, a MIDI note and a Csound note may both understand the interface function `play` but cause different output formats. In OOP, a subclass inherits all attributes and methods of its superclass. However, the superclass definitions can be extended or even overwritten for a subclass to define polymorphic classes. For example, a subclass MIDI-note may inherit from a more general class note and add or overwrite the `play` method.

A well-designed object-oriented music representation is easily extended by the user. For example, to add new features/attributes or to extend the data abstraction interface the

[7] OOP often encapsulates both the actual data and the methods processing these data in a single datum – in contrast to ADTs. Yet, this difference is irrelevant for the present research.

user defines a new subclass which inherits the full interface and all features of an existing class without changing its original definition.

Figure 2.9 shows an example of a class hierarchy for a music representation. The diagram depicts the class hierarchy of Common Music presented by Taube [1993], which was slightly simplified and translated into the UML [Booch et al., 1998; Fowler, 2003].[8] In the diagram, class attributes and methods are omitted for brevity.

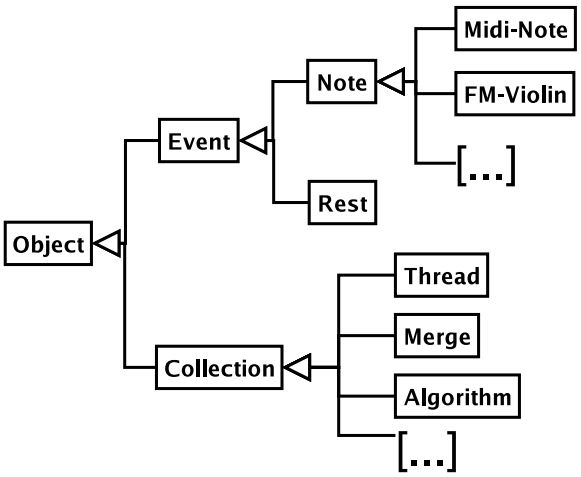


Figure 2.9.: The class hierarchy of Common Music (cf. Taube [1993])

It can be seen how inheritance is used to derive highly specific concepts (implemented by classes) from more general concepts. Every Common Music class inherits from the most general class `object`. On the next level, Common Music distinguishes between an `event` (Sec. 2.2) and a `collection` (a container, Sec. 2.4). These concepts are further sub-divided into specific event classes (e.g. `note` or even more specific `midi-note`) and container classes (e.g. the sequential container called `thread` and the simultaneous container called `merge`, see Sec. 2.4.3). A thus organised class hierarchy allows for user-extensions (e.g. allows the user to add a new note subclass such as a `csound-note` with a specific set of Csound p-fields).

For more than 20 years, OOP has proven a successful programming paradigm to implement a music representation (cf. [Pope, 1991]). This fact is also reflected by the implementation programming languages often used in the field (e.g. Smalltalk, Common Lisp with CLOS, Java). The next paragraphs briefly introduce further representation examples.

SmOKe [Pope, 1992], originally designed to provide an exchange format for music programming languages, extends the event-list concept by higher level concepts (e.g. nesting of event-lists in a tree or user-definable 'middle-level' structures such as chord or trill).

[8] Taube [1993] refers to Stella, the representation of the now obsolete Common Music version `1.*`. When compare with recent Common Music versions, the class hierarchy of Stella clearly differs in the set of containers implemented as well as in its terminology.

Desain and Honing [1997] discuss some more advanced OOP techniques in the context of modelling musical expression knowledge. The authors point out limitations of *single inheritance* where each class has at most a single superclass (as in Fig. 2.9). *Multiple inheritance* on the other hand, allows for more then a single superclass. Using this approach, a programmer can model orthogonal representational aspects in in their own superclass which results in a more modular design.

Figure 2.10 depicts the Expresso class hierarchy of an early design phase (cf. Desain and Honing [1997]). The definition of the classes *note* and *pause* shows similarities to the Common Music class hierarchy shown above. The definition of containers (called *structured* in Expresso), however, is different. To model musical expression knowledge, Expresso complements the common temporal container classes (see Sec. 2.4.3, these containers are called *sequential* and *parallel* in Expresso) by explicit representations of the ornaments *acciaccatura* and *appogiatura*.

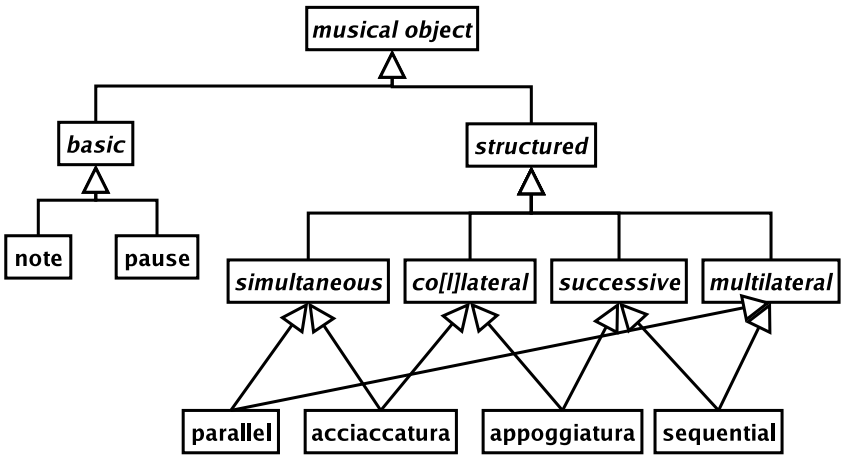


Figure 2.10.: Expresso class hierarchy with multiple inheritance resulting in the diamond problem (cf. Desain and Honing [1997])

Expresso represents two orthogonal representational aspects with four *abstract classes* (i.e. classes which will never be instantiated, marked by italic fonts in the UML) which all inherit from the class *structured*. Here, the classes *simultaneous* and *successive* model a temporal aspect of the representation whereas the classes *multilateral* and *colateral* (sic!) model a distinction between a container with peer objects and a container representing an ornament. However, in this class hierarchy some classes have a set of superclasses which in turn has a common superclass. This situation – sometimes referred to as the *diamond problem* in the OOP literature [Wikipedia contributors, 2005] – causes complications in the inheritance scheme. For example, the class *sequential* inherits from the class *structured* twice: once via the class *successive* and also via the class *multilateral*.[9]

An alternative multiple inheritance technique is the definition of a mixin class, which

[9] In CLOS – the implementation language of Expresso – such a class hierarchy is legal with a unambiguous semantics, well-defined by the precedence list of CLOS.

avoids the diamond problem but still allows the user to define a set of attributes and methods in a modular way. A *mixin class* extends its subclasses without causing the diamond problem. It can be useful to think of a mixin as an associated class instead of 'is-a' relation. In Fig. 2.11, the class hierarchy presented before is altered such that the use of mixin classes avoids the diamond problem.

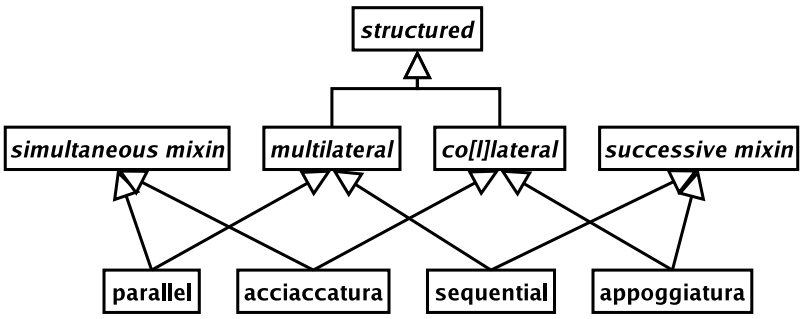


Figure 2.11.: Expresso class hierarchy excerpt with mixin classes (altered version of a class hierarchy presented in Desain and Honing [1997])

MusES [Pachet et al., 1996] defines an exhaustive and theoretically founded model to represent time and temporal relations between time intervals which are defined by a start time and a duration. The system represents temporal objects (e.g. actual notes) and non-temporal objects (e.g. pitch classes or intervals between pitches) as classes organised in a class hierarchy. The particular rich pitch representation of MusES, also modelled by OOP [Pachet, 1993], has been mentioned above (Sec. 2.3). The definition of MusES avoids multiple inheritance by using *delegation* instead: an object references another object and delegates specific method calls to it. MusES is designed as a generic and reusable representation framework to define other systems. For instance, the application of MusES in a music constraint system will be discussed later in the text (Sec. 3.3.4).

The visual-programming composition system OpenMusic [Assayag et al., 1999] features also OOP in a visual way. The OpenMusic user visually defines classes, their attributes and methods and organise them in a class hierarchy.

OOP languages disagree not only in the OOP terminology (see above), but also in many details of the paradigm itself. For instance, languages differ in the semantics of multiple inheritance: languages differ in which method to call in case a method is defined by multiple superclasses. Other languages disallow multiple inheritance altogether. Only few languages support OOP generalisations such as a meta-object protocol [Kiczales et al., 1991] (by which, e.g., the semantics of multiple inheritance can be defined by the user). These incompatibilities are likely due to the fact that OOP represents a particular complex programming paradigm which misses a solid mathematical foundation – in contrast to other programming paradigms such as functional or logic programming. Nevertheless, *design patterns* [Gamma et al., 1995] – a catalogue of higher-order abstractions for program organisation used by experienced programmers – are an example of a unifying tendency in actual OOP practice.

2.8. Higher-Order Programming for Score Processing

Section 2.6 motivated a data abstraction interface for single score objects, but it is also important that the user can query and process complex score hierarchies. For instance, the user may want to transform a hierarchic representation into a flat event list (e.g. to transform the hierarchic representation into a Csound score for output). Often, the user wants to include only specific events in the output score and exclude others (e.g. to include only events of a specific instrument or events starting after a specific time point).

Such a score transformation can be provided by an data abstraction interface function which recursively traverses the score hierarchy: by visiting all events, the program collects only the events which meet some specific condition (e.g. collects only events after a specific start time).

The designer of a music representation does not know which condition the user will need. The designer may therefore provide many highly similar transformation functions, but such an approach would violate the concision and modularity design principles (Sec. 2.5).

A more generic interface function allows the user to freely define this condition by a test function. Whereas the control structure of the program (i.e. recursively visiting all events) remains constant, not only some value (i.e. the minimal start time of events) but some sub-program (i.e. the test function) is given by the user.

The concept of *higher-order programming* addresses such situations [Abelson et al., 1985; van Roy and Haridi, 2004]. In functional programming languages – which primarily introduced this concept – a function is a first-class value. A *first-class value* can be freely stored in data structures, can be passed to other functions as arguments, or returned as results – like integers or strings in other programming paradigms. In effect, the concept higher-order programming introduces subprograms (i.e. functions) as data like anything else (i.e. a subprogram is a 'normal' data structure like an integer or a string). A function which expects a function as an argument (or returns a function as result) is called a *higher-order function*.

Functional languages usually define a highly generic and rich interface for the list data type including several higher-order functions. A typical example is the function *filter* which collects only the values of a list for which a predicate – a function given as an argument – returns *true*. For example, calling *filter* with a list of integers and a function which checks whether its only argument is even results in a list containing only the even integers (2.1).

$$\mathit{filter}([1,2,3,4,5], \mathit{isEven}) = [2,4] \tag{2.1}$$

Further common examples for a generic list interface include the functions *map* (collecting the results of a unary function applied to each list element), *sort* (sorting a list according to a binary predicate used to compare two list elements), *find* (returning the

first list element for which a predicate returns true), and *count* (counting for how many elements a predicate returns true).

In the example at the beginning of the present section, an hierarchic music representation is transformed into a flat event list such that only specific events are included in the output. The function *filter* provides the required functionality. Missing is only a function *collectAllObjects* which traverses a hierarchically structured score and returns a list with all score objects contained in the score. Figure 2.12 demonstrates how to extract an event list from an hierarchic music representation which only contains the events starting after time point 60 seconds. Whereas the previous example (2.1) used a predefined function (i.e. *isEven*), this example uses a user-defined function: the function *checkEvent* checks whether its argument is an event and whether the start time of this event exceeds a certain value.

$$\begin{array}{ll}\textbf{let} & \mathit{checkEvent}(\mathit{myEvent}) \stackrel{def}{=} \mathit{isEvent}(\mathit{myEvent}) \wedge \mathit{getStartTime}(\mathit{myEvent}) > 60 \\ \textbf{in} & \mathit{filter}(\ \mathit{collectAllObjects}(\mathit{myHierarchicScore}), \\ & \qquad \mathit{checkEvent})\end{array}$$

Figure 2.12.: Extracting an event-list with only specific events (events starting after 60 seconds) from an hierarchically structured score

Section 2.4 introduced the term score context to denote a set of related score objects and used set-builder notation as a generic device to express such a context. Filtering score objects is very similar to set-builder notation and a highly suitable means to extract complex score contexts. For example, the expression in Fig. 2.12 can be transformed into set-builder notation as shown in Fig. 2.13 (an important difference between both expressions is, that a list returned by a higher-order filtering function also expresses the order of its elements).

$$\begin{array}{l}\{x : x \text{ belongs to } \mathit{myHierarchicScore} \\ \qquad \wedge\ \mathit{isEvent}(x) \\ \qquad \wedge\ \mathit{getStartTime}(x) > 60\}\end{array}$$

Figure 2.13.: Figure 2.12 translated into set-builder notation

The function *checkEvent* is needed only once in Fig. 2.12. In such a situation, the function can be defined 'in place'. In the example variant in Fig. 2.14, the function f is defined 'in place' and given to *filter*. The colon (:) in the example reads 'where' or 'such that'.

Landin [1966] proposed the where-notation as a substitute for Church's λ-notation [Barendregt and Barendsen, 1991]. When compared with the λ-notation, the where-notation has the advantage that it uses the conventional function notation. Landin spells out 'where' as in

$$\begin{array}{l}\textbf{let}\ \ minTime \stackrel{def}{=} 60\\ \textbf{in}\ \ \ filter(\ collectAllObjects(myHierarchicScore),\\ \qquad\qquad f : f(x) \stackrel{def}{=} isEvent(x) \wedge getStartTime(x) > minTime)\end{array}$$

Figure 2.14.: Extracting an event-list with only specific events (cf. Fig. 2.12) where the filtering test function is defined 'in place'

$$g(f\ \textbf{where}\ f(x) \stackrel{def}{=} \langle function\ body \rangle)$$

In this thesis, each function is essentially a λ-expression, that is a first-class value bound to a variable. The function f in Fig. 2.14 is even a *closure* [van Roy and Haridi, 2004] which consists of the function itself plus the enclosing lexical environment binding the variable *minTime*. Such a function definition plays the same role as a `lambda` expression in languages such as Lisp and Python or a block in Smalltalk and Ruby.

A function which expects other functions as arguments (i.e. a higher-order function) is defined like any other function. As a demonstration, Fig. 2.15 defines the function *filter* as a recursive function. After an abort condition test (which finishes the recursion at the end of the list *xs*), the predicate *test* is applied to the first element of *xs*. In case this test returns *true*, this element is added to the accumulated result and *filter* calls itself recursively with the rest of *xs*. Otherwise, this first element of *xs* is ignored when *filter* calls itself recursively.

$$\begin{array}{l}filter(xs, test) \stackrel{def}{=} \textbf{if}\ isNil(xs)\ \textbf{then}\ nil \quad \textit{/* abort condition */}\\ \qquad \textit{/* Apply test to first element of } xs\textit{. Collect only matching elements but recur in any case with rest of } xs \textit{ */}\\ \qquad \textbf{else if}\ test(head(xs))\\ \qquad\quad \textbf{then}\ cons\,(head(xs),\\ \qquad\qquad\qquad filter(tail(xs), test))\\ \qquad\quad \textbf{else}\ filter(tail(xs), test)\end{array}$$

Figure 2.15.: Definition of the higher-order function *filter*

Some researchers apply higher-order programming for processing an hierarchic music representation. For example, Desain [1990] explains this concept in detail in a Lisp programming tutorial. Also, some composition systems integrate higher-order programming. For example, Common Music defines the functions `map-subobjects` (and its cousin `map-subcontainers`) which maps a function f to subobjects (respective subcontainers) of a given container [Taube, 2005]. Several optional arguments allows the user to control the behaviour of `map-subobjects`. For example, with an optional predicate the user can filter to which subobjects the function f is applied (i.e. all subobjects for which

the predicate returns true). In addition, the user can specify whether `map-subobjects` works recursively or processes only the direct subobjects of the given container.

2.9. Conclusion

The representation of music is a complex task. A music representation expresses a large amount of explicit information which constitutes a score. A representation often provides means to conveniently access explicitly stored information as well as information which can be derived from it. The information to represent is influenced by several factors, for example, the style of the music or music theory concepts. This can be seen, for instance, in the complex subfield of representing musical parameters such as pitch and also in the different schemes of hierarchic representations proposed.

This survey proposed the term score context to denote a set of related score objects and discussed the different ways shown in the literature to explicitly store and derive information required to access such contexts (e.g. by grouping the elements of a context in a container or by filtering a score for objects which satisfy a test).

A generic music representation requires that it can represent a wide range of different information (e.g. music of different styles). Such a representation can be conveniently extended by the user to express information not supported by predefined constructs.

To fulfil these demanding requirements, the field of music representation calls upon a range of different computer science concepts. The concept of data abstraction is highly important for any non-trivial music representation. Programming paradigms such as object-oriented programming and higher-order functional programming allow the expression of complex musical knowledge in a concise way.

3. Survey II: Composing with Constraint Programming

This chapter introduces the notion of rule-based composition with constraints by reviewing musical constraint satisfaction problems and systems which either implement a specific problem or allow the user to define such problems.

Chapter Overview

Section 3.1 explains the main idea of constraint programming in an informal way. The application of constraint programming for music composition is motivated in Sec. 3.2, first in general terms and then illustrated by examples from the literature of this research field. Section 3.3 surveys generic music constraint systems in which the user defines and solves own rule-based composition systems. A short conclusion closes this chapter (Sec. 3.4).

3.1. What is Constraint Programming?

Constraint programming is a programming paradigm which introduces techniques to solve constraint satisfaction problems. A *constraint satisfaction problem* (CSP) consists of a set of *variables* and mathematical relations between these variables which are called *constraints*. Usually, a CSP presents a combinatorial problem. A *constraint solver* finds one or more solutions for the problem. A *solution* of a CSP shows for each variable of the problem a determined value which is consistent with all constraints. A *constraint system* (e.g. a programming language supporting constraint programming) allows its user to define and solve CSPs.

A simple numeric example may illustrate these concepts. The example introduces the two variables X and Y and restricts their value by two basic arithmetical operations (connected by a conjunction).[1] One possible solution for this problem is $X = 3$, $Y = 4$, another solution is $X = 1$, $Y = 6$.

$$X + Y = 7 \ \wedge \ X < Y$$

In mainstream programming paradigms or with mainstream programming languages (e.g. C or Java) this problem is even not expressible at all in the concise way it is given

[1] As a convention, constrained variables are written with upper-case letters, see App. A.

here, let alone solve it. To solve a CSP automatically, a special software is needed: a constraint solver.

The reference to programming languages such a C or Java reveals conflicting terminology: in constraint programming, the term *variable* has a clearly different meaning compared to its meaning in mainstream programming languages. In mainstream programming languages, a variable denotes a *stateful* computational entity: such a variable has always a specific value and a program can alter the value of a variable with an assignment statement at any time.

In constraint programming, the notion of a variable is more similar to the notion of a variable or *unknown* in mathematics. More specifically, a variable in constraint programming is very similar to a logically quantified variable in first-order logic [Kelly, 1997]. In constraint programming, a variable never changes its value. However, the value of a variable may only be partially known.

The remainder of this text will always use the term variable in this latter meaning (as long as not explicitly noted otherwise). Sometimes, the term *constrained variable* will be used to explicitly denote a variable in the context of constraint programming.

A constrained variable is a variable which has a domain, that is a set of values it may take in a solution. Some constraint systems support variables with infinite domain (e.g. the domain of all real numbers in some interval). However, in the remainder of this text, the domain of a variable is usually a finite set of possible variable values. The values in the domain are often all of the same type (e.g. boolean domain, integer domain or domain of finite sets of integers), other systems support domains with mixed types.

As already mentioned, a variable in constraint programming never changes its value (much like a variable in first-order logic). The domain of a variable will be reduced during the search process, but the domain will not change in any other way. That is to say, a variable is *stateless*, the information available about a variable will only be amplified.[2]

In principle, constraints can be arbitrary mathematical relations. Examples include numerical relations, set relations, logic relations, and tree or graph relations. Constraint systems predefine a set of constraints and often allow the user to extend this set.

Solving a CSP requires searching. Because the search space – that is the set of (partial) solution candidates – of a CSP is often huge, an efficient constraint solver has great impact on the usability of a constraint system.

There exists a whole library of literature on constraint programming. Only important introductions are cited here for further references. Apt [2003] provides a general overview of the field with many CSP examples. Frühwirth and Abdennadher [2003] survey different constraint programming approaches and systems. Dechter [2003] primarily explains how

[2] In the actual implementation of some systems discussed below, however, the concept of the variable and its domain is somewhat decoupled. In such systems, during search the variable is indeed statefully bound to different values of the domain before constraints are applied.

constraint solvers find solutions. A less formal introduction into the field is given in the web-tutorial by Bartak [1998] which also links to further resources such as related journals, conferences, web-links etc.

3.2. Constraint-Based Composition

3.2.1. Motivation

A rich set of strategies for computer-aided composition have been proposed. Section 1.1 listed several examples.

Often, a specific composition strategy can be formalised in different ways. For example, a transformation of existing data can be formalised in the *procedural* way by imperative program statements which change this data. Alternatively, this transformation can be formalised *declaratively* by a function which expects some data and returns the transformed data – without changing the original data. If a strategy can be formalised both procedurally and declaratively, then the declarative formalisation is often preferable because declarative programs are compositional and more easy to reason about (cf. [van Roy and Haridi, 2004]).

A declarative approach is also often used in the field of music theory. Prominent examples are the explanations in composition textbooks (e.g. discussing counterpoint, harmony, musical form or instrumentation). These textbooks primarily specify the properties of a suitable result without detailing how to procedurally achieve this outcome.

An important device to declaratively express compositional knowledge in a music theory is the *compositional rule*. A rule restricts score objects (e.g. sets of notes) and their parameters (e.g. durations or pitches). Frequently, a rule controls mutual dependencies between multiple score objects and parameters.

Compositional rules are particularly well suited to describe music, because rules describe the multi-dimensional nature of music in a modular way. When listening to music, we perceive various aspects such as rhythm, harmony, voice leading or instrumentation simultaneously. During the compositional process as well, the composer builds up a complex network of relationships between all the musical elements, and observes these elements from various viewpoints more or less simultaneously. Similarly, the formalisation of a task as complex as composition is greatly simplified when the task is stated in a modular way. When the task description is broken down into rhythmic rules, melodic rules, rules on the harmony etc. then the various musical dimensions are formalised one by one.

It is very difficult to procedurally build up such a network of relations between the various musical elements of a piece. In addition, changing or adding a single rule can require redesigning the entire program.

Because rules are well-suited to describe music, rule-based approaches have been proposed since the early days of computer-aided composition. For example, Hiller and Isaacson [1958] already formalised counterpoint rules when composing movements (called 'experiments') for their famous Illiac suite.

Constraint programming (see Sec. 3.1) has proven a particularly successful programming paradigm to realise ruled-based systems. Many rule-based composition systems have been proposed which make use of constraint programming techniques. Several of these systems will be introduced shortly.

This thesis uses the term *music constraint system* to denote a system which applies constraint programming for musical purposes (e.g. assisting a composer in the composition process or a scholar in analysing music). A *musical constraint satisfaction problem* (musical CSP) implements a formal music theory model (Sec. 1.2) by a set of constraints on a set of variables.

In contrast to most CAC systems, a music constraint system user creates a musical score by formally describing a desired result only. Thus, constraints programming frees the composer to concentrate on what he wants to do in a musical sense; he does not need to define how to achieve this outcome.

A musical CSP does not necessarily result in only a single solution. Instead, the restrictions and dependencies expressed by a set of rules reduce the set of solution candidates.

A particular advantage of a music constraint system lies in the fact that it allows the composer to define compositional rules in a modular fashion. Multiple rules can even affect the same parameter value. For instance, a musical CSP may restrict the note pitches of a score by melodic rules on the one hand and by harmonic rules on the other hand. Each of these separate rules affect the same parameter values, namely the pitches. However, no rule necessarily determines the parameter values fully. Search finds one or more solution which fulfils all rules.

Many of the examples presented in this and the following chapters mainly constrain note pitches in some way. In other algorithmic composition strategies, it is often difficult to adequately address the complexity required to model pitch structures. A rule-based approach (e.g. constraint programming) is highly suited for this specific task. On the other hand, other algorithmic composition strategies are well applicable for other compositional tasks, for example, to generate complex trajectories for parameters which control arbitrary sound synthesis details or the spatialisation of sounds. It therefore makes sense to complement these different strategies and to create different aspects of the music by different strategies. Yet, the present text focusses on pure constraint-based composition for brevity.

3.2.2. A First Example

In the following, a simple (but non-trivial) musical constraint satisfaction example is described. The example was chosen for its brevity, so it can be discussed in full detail.

The example stems from dodecaphonic music composition. In this technique, the composer organises the pitches in a composition with the help of a tone row (also known as twelve-tone series) and its transformations. Křenek [1952] provides a practical introduction to dodecaphonic composition (also known as twelve-tone technique).

A *tone row* is a sequence of twelve tone names of the chromatic scale or twelve pitch classes, in which each pitch class occurs exactly once. The tone row is used as a device to achieve music which is coherent in itself even if the music abandons conventional harmony.

Aiming for a music which is most coherent in itself, several composers also shaped the tone rows themselves with great care. A prominent example for such special tone-rows is the set of *all-interval series*. In an all-interval series, also the eleven intervals between the twelve pitches are all pairwise distinct (i.e. each interval occurs only once). Therefore, such a series is sometimes also called an *eleven-interval series* [Křenek, 1952]. Other examples for special tone-rows are symmetric rows, in which some part of the row is a strict transformation of another part.

Figure 3.1 shows an all-interval series example (cited from [Gervink, 1995], but this example can also be found in [Křenek, 1952]). In this particular case, the series is even both an all-interval series and a symmetric series. It can be seen that each pitch class occurs only once in the series. Similarly, also every interval between the pitches (reported by integers above the staff) is unique. These intervals are computed in such a way that they are inversional equivalent: complementary intervals such a fifth upwards and a fourth downwards count as the same interval (namely 7).

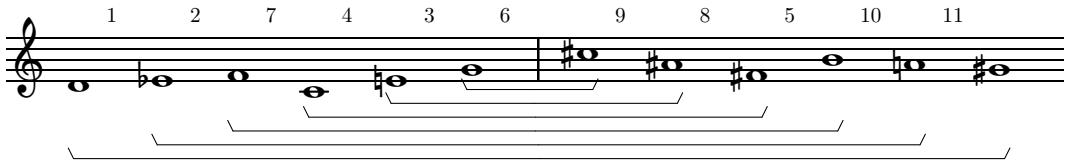


Figure 3.1.: All-interval series, whose two halves also form transposed retrogrades of each other (cited from [Gervink, 1995])

The rest of this subsection provides a full all-interval series formalisation as demonstration how a musical CSP may look like. An all-interval series obeys only very few constraints. Therefore, it is often discussed as a CSP example. For instance, [Laurson, 1996, p. 209 f] exhibits an all-interval series implementation.

Figure 3.2 provides the all-interval series CSP definition in mathematical notation. The example first creates a music representation (consisting of two lists) and binds it to the variables *pitchClasses* and *intervals*. The constrained variables in this problem are the elements in these lists. The domains for these constrained variables consist of integers. In the following, the example applies several constraints to these variables. All these constraints are connected by conjunctions. This definition has the 3856 solutions known from the literature [Morris and Starr, 1974].

$pitchClasses \stackrel{def}{=}$ a list of 12 undetermined integers, each with the domain $\{0, \ldots, 11\}$

$intervals \stackrel{def}{=}$ a list of 11 undetermined integers, each with the domain $\{1, \ldots, 11\}$

/* Constrain the relation between the $pitchClasses$ and the $intervals$. */

$$\bigwedge_{i=1}^{11} inversionalEquivalentInterval(pitchClasses_i, pitchClasses_{i+1}, intervals_i)$$

/* All elements in the $pitchClasses$ as well as the $intervals$ between them are pairwise distinct. */

$$\wedge\ distinct(pitchClasses)$$

$$\wedge\ distinct(intervals)$$

/* The first series pitch is determined to 0 (i.e. note name c) to prohibit series transpositions. */

$$\wedge\ pitchClasses_1 = 0$$

/* The last pitch is set to 6, because the interval between the first and the last interval in an all-interval series must be a tritone [Křenek, 1952, p. 50]. This additional constraint is redundant, but dependent on the constraint solver it may reduce the search space considerably. */

$$\wedge\ pitchClasses_{12} = 6$$

Figure 3.2.: All-interval series definition

Compared with the music representations discussed before (see Chap. 2), the music representation of this example – consisting of two lists of parameter values – is rather simple. The list *pitchClasses* represents the sequence of pitch classes, that is the all-interval series solution. The list *intervals* represents the sequence of the intervals between these pitch classes.

Each pitch class is represented by an integer between 0 and 11 (i.e. between *c* and *b*). Each interval is represented by an integer between 1 and 11 (0 would be unison or an octave: these intervals do not occur in a twelve-tone row). In the CSP definition, the values of the pitch classes and the intervals are undetermined – they will only be found during the search by the constraint solver.

The rest of the example consists of a conjunction of constraints on this music representation. First, the relation between all pairs of consecutive pitch classes and the intervals between them is restricted by the constraint *inversionalEquivalentInterval*. This constraint is defined in Fig. 3.3. Similar to the wellknown computation of a pitch class from a key-number, the computation of an inversional equivalent interval between two pitches makes use of the modulo function.

After the application of *inversionalEquivalentInterval* in Fig. 3.2, the constraint *distinct* demands that all variables in the list *pitchClasses* are unique. The same constraint is applied to the list *intervals*. Finally, two additional constraints determine the first and the last pitch class in the series to prohibit series transpositions and to amplify the

/* Constraints *Interval* to be an inversional equivalent interval between the two pitch classes *PC1* and *PC2* (i.e. a fifth upwards and a fourth downwards count as the same interval). */

$$inversionalEquivalentInterval(PC1, PC2, Interval) \stackrel{def}{=}$$
$$Interval = (PC2 - PC1) \bmod 12$$

Figure 3.3.: Definition of the constraint *inversionalEquivalentInterval*

information available to the constraint solver.

Morris and Starr [1974] present a specially developed algorithm to compute all-interval series. Nevertheless, music constraint programming has the advantage that the user only needs to declaratively state the problem without developing a special algorithm, whereas the latter is often a very time-consuming activity. For example, the constraint system user may easily add additional constraints which requires the solution to be both an all-interval series and a symmetrical series (as the example row in Fig. 3.1). Using an algorithmic approach often requires a laborious algorithm redesign when the problem specification changes. On the other hand, a specially designed algorithm is often more efficiently executed.

3.2.3. Musical Constraint Satisfaction Problems

Many music theories have been modelled and implemented by constraint programming. The present subsection lists research projects conducted to demonstrate the variety of musical CSPs tackled so far.

Counterpoint

There exists a considerable body of literature on counterpoint. Different textbooks often cover different musical styles. Today, two style families are taught most frequently. One approach is oriented at Renaissance music and Palestrina in particular. Fux [1725] wrote the seminal treatise on this approach; the standard textbook on Palestrina counterpoint today is [Jeppesen, 1930]. Whereas in the first approach harmonic considerations are secondary, the other approach teaches how to compose polyphonic music which expresses a harmonic progression. Baroque music, for example, usually follows this approach. Important textbooks on this approach are [Schoenberg, 1964] and [Piston, 1947]. Finally, some authors teach several different contrapuntal styles [Motte, 1981].

The 20th century saw further developments of polyphonic music; important examples include the dodecaphonic technique of Schoenberg and others, its descendants like serialism or Stockhausen's formula composition, the micropolyphony developed by Ligeti, and Nancarrow's rhythmical counterpoint.

There exist several systems creating polyphonic music by means of constraint programming. Scholastic counterpoint (e.g. the strict application of [Fux, 1725]) features a par-

ticular strict set of rules (when compared with other music theory sub-disciplines, e.g., rhythm or melody). A strict rule set makes formal modelling more easy, which explains why counterpoint has been of great interest for designers of rule-based systems.

Ebcioglu [1980] proposes a system for creating two-part florid counterpoint: to a given cantus firmus the system composes a matching voice which is rhythmically independent. The author lists almost 50 implemented counterpoint rules which include complex high-level rules such as 'the pitches of different local maxima (i.e. melodic peaks) within three measures of the voice are unique'. Sources for compositional rules were Joseph Marx and Charles Koechlin. Because their rules were insufficient for automatic composition, the author added rules of his own. The search strategy embeds heuristics which prefer steps to skips and note pitches that have not occurred before.

A system for creating species counterpoint was introduced by Schottstaedt [1989], who aimed to follow the rule set of Fux [1725] as closely as possible. The system implements all five species for up to six voices. However, the author modified the original Fuxian rule set (more then 40 rules are quoted in article) to get closer to Fux' actual examples. In accordance with music theorists (including Fux) which state that rules are merely guidelines and no absolutes, the system assigns each rule a numeric penalty value to denote its relative importance: the system searches for a solution with a small accumulated penalty.

Polyphonic music in the style of Josquin Desprez is addressed by Laurson [1996], who implemented several rules from the Josquin chapter of Motte [1981]. The goal of this research is not so much to simulate a specific historical style, but to study the problem of polyphonic CSPs in general. The resulting system PWConstraints and its subsystem score-PMC is further discussed in Sec. 3.3.1.

Also non-European counterpoint-like music theory models have been discussed in the literature. Chemillier and Truchet [2001] introduce the Nzakara harp problem. This CSP models a two-voice canon following rules which are typical for the Nzakara harp repertoire (from the Central African Republic).

Dense musical textures play an important role in the music of the 20th and 21st century. A musical CSP which resembles Ligeti-like micropolyphony was proposed by Laurson and Kuuskankare [2001] and another by Chemillier and Truchet [2001].

Harmony

There also exists a huge amount of literature on the subject harmony. Authors differ less in their focus on a certain style – the history of harmony shows a more continuous development when compared with counterpoint (where different styles are more distinct in their rule set). However, there exist different approaches to explain harmonic phenomena. Most authors describe harmonic progressions as progressions of chord roots and analyse all chords in terms of their relationship with the tonic (this idea draws on Rameau [1722]), but different conclusions are drawn. One approach is based on the assumption that chord roots indicate one of the seven (major or minor) scale degrees (notated by

roman numerals). Schoenberg [1911] wrote a particular accomplished textbook using this approach. Another approach (often called functional harmony) only accepts three different main harmonic functions (tonic, dominant and subdominant; notated usually with their initials) and explains all chords as variants of one of these main functions. This approach was founded by Hugo Riemann and is used, for example, by Motte [1976]. Finally, some authors – including Schenker [1935] and Schoenberg [1969] – put particular emphasis on larger-scale structures in harmonic progressions.

The development of harmony and its study is still ongoing. For example, the influential work by Forte [1973] describes the harmonic language of 'atonal music'[3] (i.e. music in 12-tone equal temperament without a tonal centre and often consisting of highly complex chords) in terms of pitch class sets.

Microtonal music, and in particular music in just intonation is another important example for the ongoing development. Music in *just intonation* is music based on a tuning of intervals whose frequency ratios can be represented by relatively small integers (e.g. $\frac{3}{2}$ constitutes the fifth and $\frac{5}{4}$ the major third). Whereas common practice uses only intervals consisting of integers whose highest prime factor is 5 (often called 5-limit), 20th century music greatly extended this set of intervals by new and consonant intervals (e.g. by the harmonic seventh $\frac{7}{4}$ or the harmonic augmented fourth $\frac{11}{8}$), as well as intervals more dissonant than available in common practice. The term *extended just intonation* is often used to label tuning systems which contain intervals beyond the 5-limit. The seminal work on just intonation is by Partch [1974], Doty [2002] wrote a tutorial on the subject, and an extensive encyclopedia on just intonation theory is presented by Monzo [2005].

Much research has been carried out on constraint based harmonisation. Pachet and Roy [2001] provide a survey on this subject.

Chorale, designed by Ebcioglu [1984, 1992], is a system which creates four-part harmonisations of given choral melodies in the style of Johann Sebastian Bach. Chorale received much attention for the musical quality of its output: according to its author, Chorale accomplished the competence of a talented music student. Indeed, the music created by this system can be much more complex than the output of most other harmonisation systems discussed here (e.g. most authors do without modulations). Ebcioglu's detailed analysis of compositions by Bach resulted in the impressive amount of about 350 rules implemented by the system. These address two subtasks: the harmonisation (creating of chord skeleton, style-appropriate modulation and cadencing) and melody generation (with special care of the outer voices). To meet his purpose in an efficient way, Ebcioglu first designed and implemented the constraint programming language BSL (Backtracking Specification Language).

The often-cited article by Tsang and Aitken [1991] proposes a lucid system with a small set of 20 rules which creates four-part harmonisations of a choral melody.

[3] Schoenberg – who was the first to radically break out of traditional tonic-related harmony – detested the term 'atonal music'; he preferred 'pantonal music' instead.

The system designed by Ramirez and Peralta [1998] also automatically harmonises a given melody with an appropriate chord sequence. The system finds a sequence of absolute chord names such as $[C, Dm, G, C]$,[4] whereby the system is limited to single key melodies and only considers diatonic triads in the solution. To increase the musical quality, the system further constrains solutions to follow standard chord patterns (e.g. $[I, II, V, I]$) stored in a database.

Coppelia [Zimmermann, 2001] creates homophonic chord progressions which also feature a rhythmical structure. The music theory model is split into two layered sub-models which are implemented by two independent applications. The subsystem Aaron creates a harmonic plan, represented by harmonic functions in the tradition of Hugo Riemann (such as $[T, S_3, D^7, T]$) and complements this plan by additional information (e.g. the duration of each chord and further restrictions on single voices such as "the soprano melody shall move downward"). COMPOzE [Henz et al., 1996], the second subsystem, creates the actual four-voice chord progression from this harmonic plan.

Phon-Amnuaisuk [2001, 2002] presents another system which creates choral harmonisations in the style of Johann Sebastian Bach. Phon-Amnuaisuk criticises Chorale [Ebcioglu, 1992], that this system is hard to modify. To realise a more adaptable constraint system design, he proposes a control language which controls the temporal order of decisions during the composition process. For the four-voice Bach choral example, the search process may first create the harmonic skeleton for the given melody, then outline the bass skeleton, create a draft of the other voices, and eventually create the final version of each voice by adding ornamentations such as passing notes.

A CSP based on dodecaphonic pitch class sets is described by Laurson [1996]. In this CSP, a solution consists of a sequence of (possibly overlapping) pitch class sets. These results could be used by a composer to organise the harmonic structure of music.

Melody

Melody-writing is highly style-dependent and is not a traditionally established subject in music education. Nevertheless, the subject is covered, for example, by some textbooks on composition in general. For instance, Schoenberg [Schoenberg, 1943, 1967, 1995] (the three textbooks are sorted according to their intended audience from entry-level to advanced) explains how in classical music a melody expresses the underlying harmony and how a melody is composed from motifs and their variations. Whereas Schoenberg teaches melody composition in a more systematic way, Motte [1993] studies various aspects of melodies from different musical styles (ranging from Gegorian chant to Ligeti, including children's and political battle songs).

The work of Löthe [1999] constitutes one of the few literature examples on rule-based melody composition. Löthes system creates minuet melodies (in early classical style) over a given harmonic progression. The author describes several example rules (based on

[4] The authors forbeared to formalise melodic and voice-leading rules as well: the system only creates a chord name sequence.

several sources including music theory literature from the classical period, for example, [Koch, 1793]) in detail and demonstrates the effect of different rule sets with musical examples.

Rhythm

Over centuries, western music focused on developing the pitch structure instead of rhythm, which may be the reason why the rhythmic aspect was also largely neglected by music theory. In one of the rare textbooks on the subject, Cooper and Meyer [1960] clearly define rhythmical terms (such as pulse, meter, rhythm, accent, stress, tie, syncopation, and suspension) and explain their relations in a theory of rhythm which studies the hierarchical nature of rhythmical organisation.

Yet, the 20th century saw considerable developments of the rhythmical aspect. Examples include the "additive" rhythmical permutations of motifs resulting in changes of the metric structure introduced by Stravinsky and later further developed by Messiaen [Messiaen, 1944], Nancarrow's already mentioned rhythmical counterpoint, the phasing technique in the music of Reich, and the astonishingly complex rhythmical structures in the work of Ferneyhough.

Some musical CSPs in the literature are of a purely rhythmical nature. Truchet et al. [2001] propose a poly-rhythmic problem in which each voice literally repeats a rhythmical pattern but common onsets between the voices are avoided.

A system completely devoted to rhythmical CSPs is OMRC [Sandred, 2000a,b, 2003]. For example, Sandred [2004] proposes a rule-based quantification of the rhythms of everyday gestures (e.g. extracted from the sound of a passing train) and forces these gestures into readable music notation. Rules may control what time signatures are allowed and how often the time signature may change. Additionally, the composer may apply further constraints. For example, the composer may demand that the quantified result will be build from pre-composed motifs.

Instrumentation

Berlioz and Strauss [1904] wrote a seminal book on orchestration (orchestration includes instrumentation and additionally addresses how to balance groups of instruments). Instrumentation and orchestration are also still developing today. An important example is the music of Lachenmann, which introduced many new sounds by novel playing techniques. One could even argue, that electro-acoustic music [Ungeheuer, 2002] or computer music [Roads, 1996] mainly provide means which resemble new musical instruments (e.g. hard disc recording – a software which allows its user to organise sound snippets in time – can be perceived as a new instrument for creating music) which leads to new instrumentation techniques.

To accomplish idiomatic instrumental writing, Laurson and Kuuskankare [2000, 2001] present musical CSPs in which melodic, harmonic and voice leading rules are complemented by instrumentation rules. The authors discuss guitar and brass instrument fingering in two case studies. For example, writing music for the guitar in a way which is

well playable requires instrument-typical considerations. Guitar music is performed on six strings – tuned in a particular way – on which only four left-hand fingers are placed. Furthermore, the fingers can only be stretched up to a certain amount and moving the fingers requires a certain amount of time.

Multi-Media

All musical CSPs discussed so far create scores (or score excerpts) as solutions. The following CSPs do not fall into this category, but still contribute to music.

Midispace [Pachet and Delerue, 1998] realises sound mixing and spatialisation with constraint programming. In real-time, a listener can control the position of sound sources by a graphical user interface. The system ensures a consistent mix which follows several constraints.

Ferrand et al. [1999] propose a system which assists an optical music recognition process (i.e. the automatic recognition of scanned sheet music) in dealing with uncertain and missing information. Here, constraint programming allows the declaration of musical knowledge on a high level of abstraction, which can enhance the recognition process to detect and eventually correct recognition errors of polyphonic music.

RecitalComposer [Pachet et al., 2000] is a system which does not create single pieces of music but instead creates sequences of pieces forming a music program. To this end, title attributes are stored in a database (e.g name and author, the duration, describing style, instrumentation, and tempo). Constraints on a solution sequence of titles aim to balance the amount of repetition and surprise for the listener. Example constraints are 'no slow music', '40 percent of the music is for brass', 'all authors all different', or 'succeeding titles are similar in style'.

Similarly to image mosaicing, 'musaicing' [Zils and Pachet, 2001] builds a musical sequence by specifying global properties of the solution sequence, and letting the system select and sequence automatically the sound samples. For example, a Beatles song could be recomposed out of many small fragments from famous rock titles of the sixties.

Zhong and Zheng [2004] propose a melody representation which finds a melody quickly in constraint-based database when a fraction of the melody is given.

3.3. Generic Music Constraint Systems

Much research has been conducted which employs constraint programming to implement music theory models – as the list of systems in Sec. 3.2.3 demonstrates. Most of these systems were designed exclusively to solve a single CSP or a small range of problems (e.g. the automatic harmonisation of a given melody which complies with a fixed set of rules).

Although some authors report that it is 'not particularly difficult to write such a program' [Schottstaedt, 1989], the design of a system which solves a complex CSP with reasonable

efficiency is demanding. For example, Ebcioglu [1992] deemed it necessary to first develop a new programming language for this task (namely BSL).

Moreover, music constraint systems share important requirements. Firstly, any music constraint system requires some constraint solver. Secondly, domain-specific CSPs share a considerable amount of domain-specific knowledge: all musical CSPs require modelling of musical knowledge. For instance, concepts such as note, pitch, or voice are required in a large number of musical CSPs. Consequently, several more generic systems were proposed since the early nineties.

This thesis introduces the term *generic music constraint system* to indicate a system which is designed to solve a considerable number of musical CSPs – in contrast to a system designed specifically to solve a single CSP or a small set of problems. A generic system is often developed for users – such as composers and music theorists – who would hardly consider the design of a new rule-based system from scratch, but can greatly contribute to the research in this field.

A generic music constraint system allows these users to define and solve their own musical CSPs. In order to be more generic, these system usually aim to be style-neutral (much like general computer-aided composition environments as OpenMusic or Common Music aim to be style-neutral).

A pioneering generic music constraint system is Carla [Courtot, 1990]. This system emphasises its music representation which features an extensible type system. Constraints complement this representation. The system features a visual logic programming language to make it accessible to non-programmers (such as composers) and is implemented in a Prolog dialect (Prolog II).

The remainder of this chapter introduces several generic music constraint systems. Three such systems are presented in detail (Sec. 3.3.1 to 3.3.3) to introduce important concepts of music constraint programming and to demonstrate how typical requirements on such a system have been answered in different ways. Furthermore, these systems gained particular importance as they had/have a relatively large number of users. Section 3.3.4 lists further systems.

3.3.1. PWConstraints

PWConstraints [Laurson, 1996; Rueda et al., 1998; Laurson, 1999] is a constraint programming language on top of PatchWork [Laurson, 1996; Assayag et al., 1999], a visual programming language for computer-aided composition.

This present section describes PWConstraints in much detail for two reasons. PWConstraints constitutes a fine example for introducing important concepts of constraint-based computer-aided composition in general. Moreover, PWConstraints served as a role model for the present research.

PWConstraints consists of two main music constraint systems which share a similar search strategy. A general constraint programming language offers means to constrain

common data-types, in particular lists of integers (Sec. 3.3.1.1). A special constraint programming language for polyphonic music offers means to constrain a score represented in a more complex music representation (Sec. 3.3.1.2). Section 3.3.1.3 discusses PWConstraints search strategy.

A Side Note: Visual Programming Languages

PatchWork programming constructs present themselves as graphical boxes, which is typical for many visual programming language (e.g. this is also the case for the widespread system Max [Puckette, 2002] and its relatives jmax and Pd). 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.

A main purpose of the visual programming paradigm [Boshernitsan and Downes, 2004] is to make the programming more easy for the user, in particular for programming beginners. However, not every program becomes more easy to write in a visual way. Visual programming is highly suited for control flow languages with concurrency (such as Max). Mathematical relations – which are the bread-and-butter of compositional rules – are often more concisely expressed by textual formulae. Similarly, the pattern matching mechanism provided by PWConstraints (see Sec. 3.3.1.1) can only awkwardly be expressed visually.

Interestingly, visual programming systems for music are usually not defined in themselves but in textual languages. This is in clear contrast to other high-level languages (e.g. Common Lisp, or Smalltalk) which define much functionality in themselves. This difference clearly hints that the visual programming paradigm does not scale up for larger projects in general.

On the other hand, visual programming is well suited for the top-level definition of an application. When compared with a textual interface, a graphical user interface (GUI) is much more convenient for daily use of an application. Yet, the programming logic of a GUI is usually fixed – in contrast to a user interface which consists of a textual programming language. Visual programming unites the convenience of a GUI with the flexibility of a programming language. PWConstraints follows this pattern: the user defines important components of a musical CSP (e.g. the rules) in a textual programming language, but combines them in a visual language. This approach can facilitate large scale changes of the CSP during the composition process (such as enabling and disabling of rules).

Further music constraint systems with support for visual programming are Situation and OMClouds which are both introduced below (Sec. 3.3.2 and Sec. 3.3.3). Also the visual language Cordial [Rueda et al., 1997] (supporting constraint programming and

object-oriented programming) has already been applied for music constraint programming [Rueda et al., 2001]. Burnett [2006] provides a bibliography of research on visual languages, including visual logic and constraint languages.

3.3.1.1. The General Constraint Programming Language: PMC

The Music Representation

The basic concepts of PWConstraints are explained here by sketching a simple example CSP which will be by and by extended later. In this example, a composer wants to create a choral melody. The composer decides that the melody consists of 9 notes and 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). 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. Each variable has thus the domain $\{57, \ldots, 69\}$.

The general constraint programming language of PWConstraints (i.e. `PMC`) is well suited for constraining a list of integers. Because the said composer is only interested in the note pitches, a music representation in which the solution consists of a plain sequence of MIDI-pitches (i.e. integers) is sufficient. The composer thus creates a list of 9 constrained variables with integer domain. In the PWConstraints documentation, the term *search variable* is used for constrained variable.

In `PMC`, all variables are declared together in a list of domain specifications (Fig. 3.4). According to PWConstraints terminology, the user 'defines the search space' when declaring the variables which serve as the music representation. When no further constraints are applied, each combination of nine pitches in the specified domain is a solution (9 variables – each with a domain of 12 values – have $12^9 = 5159780352$ solutions in total).

$$[\{57, \ldots, 69\}, \{57, \ldots, 69\}, \{57, \ldots, 69\}, \{57, \ldots, 69\}, \{57, \ldots, 69\},$$
$$\{57, \ldots, 69\}, \{57, \ldots, 69\}, \{57, \ldots, 69\}, \{57, \ldots, 69\}]$$

Figure 3.4.: Declaring the domains for the melody pitches

The user may already determine certain variables in the CSP definition. For example, the user may specify that any solution begins and ends with a certain pitch by specifying this pitch as the only domain value for these variables.

In general, the music representation format of PMC consists of a list containing constrained variables. PMC supports variables with universal domain. Whereas in the example the domain of these variables was a user-specified set of integers, a variable domain can in fact contain arbitrary values. For example, variable domains often consist of lists of integers.

The user can interpret variable values in various ways. In the example above, a sequence of integers were interpreted as the MIDI-pitches of a melody. Alternatively, such a sequence can be a sequence of duration values. A list of lists of integers can be interpreted, for example, as a chord sequence (where a list of integers represents the chord pitches by MIDI numbers without specifying the chord duration) or as a sequence of rhythmic motifs (where a list of integers represents a sequence of note durations).

The Rule Formalism

In PWConstraints, a CSP is defined by complementing the declaration of the search space by compositional rules which pose restrictions on a solution. Figure 3.5 reveals that the declaration of the variables of the CSP and the rules constraining these variables are given separately to the constraint solver. Handing multiple rules to the solver implicitly expresses a conjunction of all these rules. The actual variables are created by the system automatically. The PWConstraints user accesses variables only within a rule definition. This design has important consequences (e.g. it is not possible to directly apply a constraint to variables).

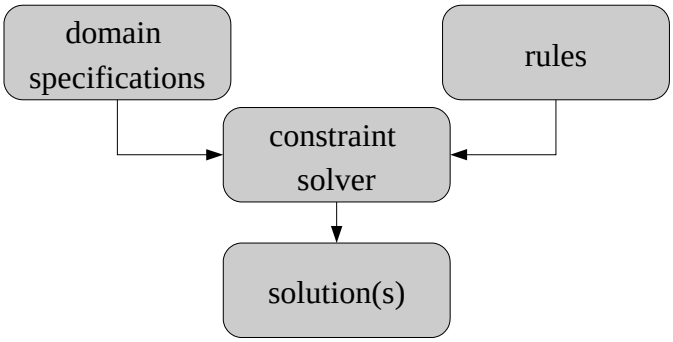


Figure 3.5.: In PWConstraints, the variable domain specifications and the rules of a CSP are handed independently to the constraint solver

In PWConstraints, the body of a rule is an expression which returns a boolean value. Figure 3.6 shows an example of such a rule body which restricts the interval between two successive pitches ($Pitch_{predecessor}$ and $Pitch_{successor}$) to a fifth as maximum.[5] Because the pitches are encoded numerically, the composer can define numeric relations between them. The example introduces a new constrained variable *Interval* and applies a conjunction of two constraints.

PWConstraints introduces a pattern matching mechanism to control to which variables in the CSP a rule is applied. This mechanism also binds the free variables in the rule body. For example, the constraints in Fig. 3.6 can be applied several times to the choral

[5]This rule is a simplified version of a Palestrina-style counterpoint rule which is reflected by many counterpoint treatises. In its strict form, the rule prohibits all diminished and augmented melodic intervals and permits downwards only intervals from minor second to fifth plus octave and upwards all these intervals plus the minor sixth [Jeppesen, 1930].

$$\begin{array}{ll}\textbf{let} & \mathit{Interval} \\ \textbf{in} & \mathit{Interval} = \mathit{Pitch}_{predecessor} - \mathit{Pitch}_{successor} \\ & \wedge\, \mathit{Interval} \in \{-7, \ldots, 7\}\end{array}$$

Figure 3.6.: The body of a rule which restricts melodic intervals

melody notes such that $\mathit{Pitch}_{predecessor}$ and $\mathit{Pitch}_{successor}$ are bound to every pair of successive melody pitches.

Pattern matching is well-known, 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` introduces a language to express the position of elements in the sequential music representation. This language consists of only three constructs.[6] There are two place-holders and a pattern-matching variable declaration. The place-holder symbol ? matches exactly one sequence element, and the symbol $*$ matches zero or more elements. The third construct specifies pattern-matching variables (which provide bindings for the free variables in a rule body definition). A pattern-matching variable is bound to the the element at its position in the sequence (all variable names in a pattern matching expression must be unique).

An example will illustrate this pattern-matching language. In Fig. 3.7, the place-holder ? matches the first element in the sequential music representation. The symbol $*$ matches either no element, or the second, or the second and the third and so fourth. Consequently, the two pattern-matching variables $\mathit{Pitch}_{predecessor}$ and $\mathit{Pitch}_{successor}$ match any pair of two successive elements – except the first pair. Note that no final $*$ is required which would match the rest of the sequence.

$$[?, *,\ \mathit{Pitch}_{predecessor},\ \mathit{Pitch}_{successor}]$$

Figure 3.7.: Pattern matching expression matching any pair of two successive elements – except the first pair

In PWConstraints, a rule definition consists of a pattern-matching part and the rule body. The pattern-matching part is expressed in the pattern-matching language just introduced. The rule body is an expression in which the pattern-matching variables of the pattern-matching part are free variables. The free variable in the rule body with the same name as some pattern-matching variable is bound to its value (i.e. the sequence element matching the pattern-matching variable in the pattern).

$$[\langle \textit{pattern matching expression} \rangle, \langle \textit{rule body} \rangle]$$

[6] PWConstraints also introduces a fourth construct – index variables – for convenience. However, all patterns using index variables can be reproduced with the three constructs introduced in this section.

Figure 3.8 defines a full PWConstraints rule definition consisting of a pattern matching expression and the rule body expression.[7] For every solution to a CSP, PWConstraints makes sure that the body of a rule returns true for every match of its corresponding pattern matching expression.

$$
\begin{array}{l}
[[*,\ Pitch_{predecessor},\ Pitch_{successor}], \\
\textbf{let}\ \ Interval \\
\textbf{in}\quad Interval = |Pitch_{predecessor} - Pitch_{successor}| \\
\qquad \wedge\, Interval \in \{0, \ldots, 7\}]
\end{array}
$$

Figure 3.8.: Full `PMC` rule definition example which states that the interval between two consecutive pitches must not exceed a fifth

The Rule Scope Concept

This text introduces the term *rule scope* to simplify the discussion of rules and their effect. In most cases, the constraints of a rule are applied to multiple variable sets. In case of the rule in Fig. 3.8, its constraints are applied to any set of variables in the CSP which consists of two consecutive pitches and the interval between them.

The phrase 'the scope of rule x' denotes 'the set of variable sets constrained by rule x'. The scope of the rule in Fig. 3.8 is the set of all variable sets which consists of two consecutive pitches and the interval between them. Figure 3.9 notates this example rule scope more formally in set-builder notation.

$$
\begin{array}{l}
\{\{Pitch_1, Pitch_2, Interval\} : \\
\quad Pitch_1 \text{ and } Pitch_2 \text{ are consecutive pitches, } Interval \text{ is the interval between them}\}
\end{array}
$$

Figure 3.9.: The rule scope of the rule in Fig. 3.8

It is important to note, that a PWConstraints rule always defines its constraints together with its rule scope. This scope is primarily expressed by the pattern matching expression of the rule – an integral part of a rule definition – which indicates to which sets of score objects the rule is applied. Other systems (e.g. `score-PMC`, Sec. 3.3.1.2) support further techniques to express which variable sets are constrained by a rule (e.g. by providing means to access further variables from within the rule).

Additional terminology further simplifies the discussion of the effect of a compositional rule. The term *rule application* denotes the enforcing of the constraints of a rule to

[7] In PWConstraints, rules are defined in Lisp – this text uses a more abstract mathematical notation instead. The notation has been further simplified. For example, PWConstraints follows the convention used by Norvig [1992] to mark pattern matching variables by a preceding ? (e.g. ?*Pitch*) and to mark the rule body by a preceding ?*if*. Also, every PWConstraints rule definition contains a documentation string.

a single set of constrained variables. Following this terminology, the pattern matching expression of a single PWConstraints rule often expresses multiple rule applications. This text refers to any set of variables constrained by a single rule application as a *score context* constrained by this rule (see Sec. 2.4). All score contexts of all applications of a single rule together form the scope of this rule.

Dependency Between the CSP Definition and the Search Strategy

In PWConstraints, the declarative side of a rule definition and the procedural side of the search process are somewhat interwoven. The search strategy visits and determines the variables in the CSP in a specific order.

The validity of any rule application is only checked after the full score context constrained by this rule is determined.[8] This fact has some consequences for the definition of a rule.

The first consequence is a benefit. Because any variable is already determined when it is used in the body of a rule, it can be processed by any function. This means that virtually any Lisp function can be used to define compositional rules in PWConstraints.

The second consequence restricts the manner how a compositional rule is defined. When the validity of a rule is checked during the search process, all the variables it constraints must already be determined.

This restriction is demonstrated by an example. Figure 3.10 defines a rule which constrains a melody such that the highest pitch occurs only once. Such a rule must be formulated by constraining the relation of already determined note pitches. For this purpose, PWConstraints provides the special variable *alreadyDeterminedVariables*.[9] This variable is bound to a list of all determined variables (in reverse order, including the currently visited variable as first element). Each step during the search process its value is updated and each rule application sees the current binding. Using this variable, the rule can be defined as follows: either the value of the currently visited variable is higher than any previously determined variable, or it is lower than the previous maximum value (Fig. 3.10).

$$[[*,\ Pitch],\\ Pitch \neq max(tail(alreadyDeterminedVariables))]$$

Figure 3.10.: A PWConstraints rule can only express constraints between already determined variables: the rule constrains that a melodic peak is unique in the melody

The most important disadvantage is this: checking the validity of a variable only after it becomes determined severely impairs efficiency. This aspect is briefly discussed in Sec. 3.3.1.3.

[8] Forward-checking rules form a special case (see Sec. 3.3.1.3). The validity of these rules are checked when a single variable of the constrained score context is still undetermined and the domain of this variable is possibly reduced.

[9] The actual PWConstraints name for this variable is `rl`.

3.3.1.2. Constraining Polyphonic Music: score-PMC

Because of its music representation – a sequence of variables – `PMC` is best suited for musical CSPs in which some sequence of score objects is constrained. Many musical CSPs can be expressed that way, particularly music theory models which belong to some preparatory stage for the actual composition process (e.g. the creation of a purely rhythmical figure, a dodecaphonic series, or a harmonic progression).

However, most western music is polyphonic by nature. Here, the term polyphony is used in a more general meaning than usual: most music is organised in multiple layers, for example, multiple voices, a melody plus accompaniment, or any other musical texture where multiple events are played simultaneously.

A constraint system must take the polyphonic nature of music into account for most CSPs outside the category of preparative CSPs. Yet, polyphonic CSPs are hard to express by only a sequential music representation.

A constraint system with only a sequentially representation can in principle represent complex polyphonic music – after all, the sequential event list representation (Sec. 2.2) can express highly complex scores – but it is still very hard for such a system to express polyphonic CSPs. Polyphonic CSPs often require constraining complex score contexts. For example, typical counterpoint rules allow dissonant notes only in very specific circumstances. Checking whether these circumstances apply for a certain note requires accessing complex score information such as the pitch of the note and its melodic predecessor or successor notes, the metric position of the note with respect to the measure it belongs to (e.g. either strong or weak beat) and whether or not the note pitch is consonant with respect the pitches of simultaneous notes (or the chord these pitches form). It is hard to express and constrain these score contexts in a sequential representation, because the information required to deduce which score elements belong to theses contexts is missing.

Additional information to denote score contexts can be added in a hierarchical representation. For example, the information which note belongs to which voice can be expressed adequately by grouping all voice notes in a voice container.

PMC supports variables with universal domain (Sec. 3.3.1.1) which can in principle be used to create hierarchically structured scores with PMC. In this approach, the domain of a variable does consist of composite values which represent more or less elaborated musical fragments (e.g. motifs) – which can even be created by sub-CSPs. This approach has been used, for example, for constraint systems build on top of the PMC[10] (e.g. OMRC [Sandred, 2003], and the system by Jacopo Schilingi[11]). Still, this approach resolves the restrictions concerning score contexts only partly (e.g. how to access at the same time the neighbouring notes in the melody, the simultaneous notes, and the metric position of a note?). Moreover, this approach leads to severe performance problems (which are discussed in more detail in Sec. 10.1.2).

[10] These systems are actually built on the port of PMC to OpenMusic called OMCS.
[11] Personal communication at PRISMA meeting, January 2004 at Centro Tempo Reale in Florence.

PWConstraints proposes an alternative approach to address polyphonic CSPs: PWConstraints defines the special subsystem `score-PMC`. When compared with `PMC`, the main change of `score-PMC` is its more elaborate music representation. Most notably, the music representation of `score-PMC` explicitly represents additional score contexts.[12] In contrast to the one-dimensional music representation of `PMC`, the music representation of `score-PMC` resembles more a two-dimensional 'musical map'. Here, the main 'dimensions' are notes in sequential order and simultaneous notes. More specifically, the music representation of `score-PMC` explicitly represents the following contexts:

Melodic Context: For any part of a polyphonic score, the music representation stores the horizontal order of notes.

Harmonic Context: The representation of simultaneous notes again shows a dependency between a CSP definition and the search strategy. The harmonic context of a note x is the set of all notes which are sounding at the attack time of the current note and are already determined when the note x is visited during the search process.

Harmonic Slice: In contrast to the harmonic context, a harmonic slice is the set of all notes which are simultaneous to a note x, regardless whether these notes are already determined or not.

Metric Context: The metric context of a note expresses the rhythmic pattern which this note belongs to (e.g. a succession of an eighth-triplet) and the position of this note in the pattern (e.g. second note of the triplet). The rhythmic pattern is represented by the expressive RTM-notation (Sec. 2.3).

The additionally available score contexts of `score-PMC` allow the user to complement melodic rules (e.g. the simple rule restricting melodic intervals which was discussed above) by rules which constrain other score contexts. The following harmonic rule example prevents voice-crossing in polyphonic music. Voice-crossing occurs when an upper voice becomes the lower voice and vice versa (see Fig. 3.11). Prohibition of voice-crossing constitutes a common counterpoint rule, often stated for pedagogical purposes (cf. [Schoenberg, 1964]).

Figure 3.12 shows an implementation of a rule which prohibits voice-crossing. The rule constrains that the order of simultaneous note pitches corresponds to the order of parts (i.e. the note pitch in a lower part is below the simultaneous note pitch in an upper part). Note that a pattern matching expression in `score-PMC` matches a note object and not just a pitch as in `PMC`. The note's pitch is an attribute of that object.[13]

[12] Compared with the score context definition given in Sec. 2.4, PWConstraints defines the term context with a more narrow meaning which only denotes the four contexts listed here.

[13] The example transforms an original Lisp implementation of the rule [Laurson, 1996, p. 238] into mathematical notation. Additionally, the names of the following `score-PMC` constructs are changed for better readability. The function *getHarmonicContext* is originally called *hc*, *getPartNumber* is called *partnum* and *getPitch* is called m in PWConstraints.

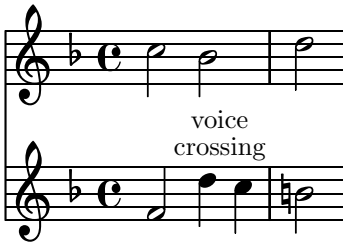


Figure 3.11.: Crossing of two voices

$[[*, note],$

let /* *sortedNotes* is a list containing *note* and all its already determined simultaneous notes (the harmonic context of the note). This list is sorted by their part number (the bass has the lowest part number). */

$sortedNotes \stackrel{def}{=} sort(cons(note, getHarmonicContext(note)),$
$f : f(x, y) \stackrel{def}{=} getPartNumber(x) < getPartNumber(y))$

in /* *isIncreasing* constraints the pitches of the *sortedNotes* to be sorted in increasing order. */

$isIncreasing(map(sortedNotes, getPitch))]$

Figure 3.12.: Definition of a harmonic rule which prohibits voice-crossing (cf. [Laurson, 1996, p. 238])

An important restriction of `score-PMC` lies in the fact that note *pitches* are the only variables in the music representation. All other aspects – in particular the rhythmic structure of the score – must be fully determined in the CSP definition. However, the rhythmic structure can be arbitrarily complex.

In `texture-PMC` [Laurson and Kuuskankare, 2001], an alternative extension of PWConstraints, other note parameters can also be constrained, 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.

Because `score-PMC`'s music representation shows a complex internal structure – in which aspects of the representation and the search are intertwined – it would be very hard for the user to adjust or extend this representation. For example, it would be hard to add support for additional score contexts which may be required to define complex CSPs.

3.3.1.3. The Search Strategy

PWConstraints' constraint solver performs a *chronological backtracking* (BT) search [Dechter, 2003]. This algorithm first selects a value for a variable from its domain. Then the algorithm checks whether all constraints applied to this variable (or this and any other already determined variable) are satisfied. In case all these constraints are satisfied, the algorithm proceeds to determine the next variable. If any constraint fails, the

algorithm tries sequentially the other domain values for this current variable – as soon as a variable value fulfils all constraints, the algorithm proceeds to the next variable. However, if no domain value satisfies all constraints, then the algorithm met a dead-end. In that case, *backtracking* occurs: the algorithm turns back to the previously visited variable and continues to try out its other domain values. When all variables are determined, the algorithm found a solution to the CSP.

BT performs a *complete* search (i.e. if there exists a solution, the algorithm finds it). The algorithm can find one, multiple, or all solutions.

BT has several well-known weaknesses:

- BT always *detects a conflict too late*: only after all variables related by a constraint are determined, the constraint may either succeed or fail. This problem is addressed by *constraint propagation* (or *consistency enforcing*) [Dechter, 2003].
- Failures with the same cause occur repeatedly (*thrashing*): BT does not analyse which variable causes a constraint to fail. This problem is addressed by techniques like *intelligent backtracking* (or *backjumping*) [Dechter, 2003].
- When after backtracking the algorithm checks variables which it already checked before, it it does not remember which value combinations failed before and will check them again (*redundant work*). This problem is addressed by techniques like *backmarking* [Dechter, 2003].
- Finally, BT performs decisions in an order that was fixed before the search started (*static variable order* and *static value order*, also known as static variable and value selection). In particular, the order in which variables are visited during the search process immensely affects the size of the search space. This problem is addressed by *dynamic variable orderings* [Dechter, 2003].

In case of `PMC`, the order in which variables are visited during the BT search process is the sequential order of the variables in the music representation.

`score-PMC` also performs BT. However, the system first computes an efficient order in which the notes in the score are visited during the search process. To this end, `score-PMC` evaluates the rhythmic structure of the solution. Using this rhythmic structure, `score-PMC` calculates a search order for its polyphonic music representation such that the search process always proceeds 'from left to right', that is notes with a smaller start time value are visited first. This search strategy is the reason why `score-PMC` requires that the temporal structure of any CSP is fully determined in the problem definition.

In addition to strict rules (which a solution must always obey), PWConstraints supports heuristic rules which allow the user to express mere preferences and to avoid over-constrained situations (where multiple constraints contradict each other). The body of a heuristic rule returns a number (the *heuristic value*) instead of a boolean value. For

'better' solutions the rule returns a higher number. In case of a conflict between heuristic rules, the more important heuristic rule (the one with a higher heuristic value) is preferred to hold.

Anyway, this technique does not necessarily find any best solution with respect to heuristic rules. Heuristic rules affect the *value order* [Bartak, 1998], that is in which order the values of a variable's domain are checked. During the search process, a domain value of the current variable with the highest heuristic value is checked first. Thus, heuristic rules guide the search to find 'better' solutions earlier, but with no guarantee to find an optimal solution.

For a better performance, *forward-checking* [Dechter, 2003] (a limited form of constraint propagation) can be applied together with BT by defining special forward-checking rules. Nevertheless, Laurson [1996, p. 209] reports his experience that pure BT performs quite often surprisingly efficient, and thus forward-checking is rarely used.

3.3.2. Situation

Situation [Rueda et al., 1998; Assayag et al., 1999] was originally conceived in collaboration between the composer Antoine Bonnet and the computer scientist Camilo Rueda as a constraint system for solving a range of harmonic CSPs. Situation was first developed as a PatchWork library, and was later extended and ported as OMSituation [Bonnet and Rueda, 1999] to the PatchWork successor OpenMusic [Assayag et al., 1999]. Nevertheless, this text will refer to any version of the system simply as Situation.

Early versions of the system supported quasi ready-made components to define CSPs. For example, instead of defining compositional rules from scratch, the user utilised predefined 'rule templates', that is compositional rules which expected arguments from the user to detail their effect. Although such an approach limits the CSPs the user can define, it also simplifies the definition process. Later, the system was extended to support a larger set of CSPs (e.g. rhythmical CSPs as well). Still, the design of the system is better understood with its history in mind.

3.3.2.1. Music Representation

The music representation of Situation consists of a sequence of *objects*. An object has an internal structure: it consists of one or more *points*. Furthermore, Situation defines *distances* between these points. In many CSPs, a point represents a note pitch, an object represents a chord containing multiple pitches, and a distance represents an interval between two note pitches.

Situation distinguishes between internal distances and external distances. An *internal distance* represents the distance between two points of a single object (also called vertical distance: interpreted as the distance between two neighbouring note pitches in a single chord) and an *external distance* represents the distance between two points from different

objects (also called horizontal distance: interpreted as the distance between matching note pitches of two neighbouring chords)

The points and the distances constitute the variables in a CSP. Their domain consists of numbers.[14]

The meaning of points and the distances between them is not specified by Situation. The user may interpret their meaning at will. For instance, points may represent pitches which can be interpreted as measured in MIDI key-numbers, or as frequency values measured in *hertz*. Points may also represent note start times, for example, measured in beats. In that case, distances represent temporal intervals.

The music representation of Situation is particularly suited for CSPs on chord sequences: an explicit representation of pitches and the intervals between them is suitable for many such problems. The implicit constraints between points and distances conveniently control the relation between these variables.

Situation is not only convenient to use but also generic. The Situation user can control which relation actually holds between points and distances. This allows for the range of interpretations already mentioned. For example, in case points represent MIDI key-numbers, then the relation between points and distances is governed by *addition* constrains. If, on the other hand, points represent frequencies, then *multiplication* constraints govern the relation between points and distances. Situation also offers an alternative representation mode which consists of a plain sequence of variables. This representation mode supports variables with universal domains (e.g. OpenMusic chord instances as domain values).

3.3.2.2. Rule Formalism

Situation provides pre-defined constraints which are not just constraint primitives like numeric operations (e.g. =, >, or +) but generalised compositional rules. For example, Situation defines a set of constraints which impose patterns on various viewpoints of the music (e.g. rules affecting a voice profile by constraining the number of consecutive upward and downward movements or the number of repeated melodic intervals).

Situation complements these pre-defined constraints by means for user-defined constraints. Comparable with the body of a PWConstraints rule, a user-constraint in Situation is an arbitrary expression (on already determined variables) which returns a boolean value. In Situation, this expression is encapsulated in a function.

Again similar to PWConstraints, Situation supports a convenient and powerful language to express to which variables in the music representation a constraint (predefined or user-defined) is applied. The user applies constraints always by the special constraint application language of the system – like in PWConstraints. However, Situation defines

[14] Internally, each point is represented by its own variable, whereas a sequence of distances is represented by a single variable whose domain consists of number sequences.

much more symbols than PWConstraints' pattern matching language and Situation's supported symbol set depends on the actual constraint applied. Nevertheless, for many constraints the application specification has the same structure: an alternating sequence of range specifications and further arguments.

$[\langle \textit{range specification} \rangle_1, \langle \textit{arguments} \rangle_1, \langle \textit{range specification} \rangle_2, \langle \textit{arguments} \rangle_2, \dots]$

Somewhat simplified, range specifications define the scope of a constraint (the term rule scope was introduced in Sec. 3.3.1.1). For example, with the following expression the user may apply a rule to the first, second and fifth object in the music representation with $\langle \textit{arguments} \rangle_1$ and to the seventh to tenth object with $\langle \textit{arguments} \rangle_2$.[15]

$[0, 1, 4, \langle \textit{arguments} \rangle_1, 6\_9, \langle \textit{arguments} \rangle_2]$

This chapter earlier introduced the notion of the set of variables constrained by a single constraint application (the score context constrained by a rule or constraint, see Sec. 3.3.1.1). In Situation, this set is specified by several factors including a single range specification (in the constraint application language), the associated arguments, the number of arguments expected by a function implementing a user-constraint, and the optional access of already determined objects (predecessors of current object) within a user-constraint definition. In summary, the score context constrained by a single constraint application can be any set of variables in the music representation. Also the set of all score contexts constrained by multiple applications of a constraint (i.e. the rule scope) is fully user controllable by range specifications of the constraint application language.

Nevertheless, because of the structure of its music representation Situation does only support score contexts defined by positional relations of objects (and their points plus distances). Consequently, it is difficult to express rules which constrain complex score contexts as required, for example, in polyphonic CSPs. Section 4.1.1 explains this limitation further.

3.3.2.3. Search Strategy

Compared with PWConstraints, Situation's constraint solver performs a more sophisticated search strategy. The Situation applies a variation of minimal forward checking. *Minimal forward checking* [Dent and Mercer, 1994] is a variant of forward checking. By means of lazy evaluation, minimal forward checking delays the reduction of the domain of variables not yet visited (i.e. the forward checking) until it is actually required during the search. That way, the algorithm provides the benefits of forward checking (to prune of domains, i.e. to reduce the search space) by reducing the work the algorithm has to perform (searching through possibly large domains for consistent values)

Situation's *first-found forward checking* algorithm [Rueda and Valencia, 1997] adapts minimal forward checking for hierarchical domains. A variable for distances (a variable whose domain consists of number sequences, see above) has often a large domain. Therefore, Situation optionally organises this domain in a hierarchical manner for efficiency.

[15] Indices are 0-based, index 0 points to the first object.

Such an hierarchically structured domain combines domain value subsets which share some property into a subtree of the domain. That way, a constraint on this common property can be propagated for all domain values in the subtree at once. Per default, this common property is the sum of the sequence elements (i.e. the sum of object distances), because this property is constrained in many harmonic CSPs. The user can specify a different common property. The data structure of Situation's music representation is optimised for first-found forward checking (i.e. all distances of an object are represented by a single variable with a sequence domain, as mentioned already in fn. 14).

In addition to strict CSPs, Situation also supports CSPs with weak constraints. In the latter, a numeric preference value is specified for each constraint.

3.3.3. OMClouds

The music constraint system OMClouds [Truchet et al., 2001, 2003] extends the composition system OpenMusic [Assayag et al., 1999].

3.3.3.1. Search Strategy

The search strategy applied by OMClouds differs clearly from the strategies of the systems discussed so far and therefore it is discussed first. The system is a based solely on a heuristic search strategy which iteratively improves an initial random solution to a CSP. More specifically, OMClouds' constraint solver performs a local improvement search strategy [Codognet and Diaz, 2001; Codognet et al., 2002]. This strategy quickly finds an approximated solution to a constraint problem which fulfils many or most of the constraints imposed.

Internally, a constraint is implemented by a cost function, which returns a numeric value indicating how much its constraint is violated. In a nutshell, the algorithm starts by determining all variables of the CSP to some random value from their domain. Then, the algorithm computes the accumulated cost of all cost functions for every variable to find the variable whose current value conforms least a solution to the CSP. This variable is updated for the next iteration. OMClouds extends this basic idea by a taboo search method to avoid local minima in the search space.

On the user-level, however, a CSP is expressed by strict constraints (e.g. $X = Y$). Only internally, these constraints are translated into cost functions (e.g. $X = Y$ is transformed into $|X - Y|$).

The heuristic approach of OMClouds makes the definition of complex CSPs easier because no over-constrained situation can occur. In a system which only supports strict constraints, the CSP must explicitly handle special cases in which some rule can be neglected. 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.

Adaptive search does not execute a complete search. If the system is asked multiple times for a solution to the same problem it usually comes up with different solutions, but the system cannot output all solutions. Moreover, the system does not guarantee to find an optimal solution for strict constraints. For examples, constraints like 'all elements of a list are pairwise distinct' are hard to fulfil and the algorithm may thus never find an exact solution to a CSP with this constraint (e.g. OMClouds may never find a true all-interval series).

3.3.3.2. Music Representation

In OMClouds, variable domains can consist of any values. However, in every CSP all variables share the same domain. This restriction limits what CSPs can be defined, but simplifies their definition.

Similar to PMC (see Sec. 3.3.1.1), OMClouds' music representation consists of a flat list of variables.[16] Truchet chose this straightforward representation format, because several composers she contacted were afraid they would be biased by a more expressive music representation.[17] A purely sequential representation can express several musical structures. However, such a structure is poorly suited for more complex musical CSPs, in particular polyphonic problems (see Sec. 3.3.1.2).

3.3.3.3. Rule Formalism

OMClouds supports easy-to-use means to apply constraints homogeneously to all variables of the CSP. However, when compared with PWConstraints and Situation, OMClouds offers little user-control on the scope of a rule (Sec. 3.3.1.1). Again, this design results in a system which is clearly limiting but also more easy to use. All constraints of a CSP are merged in quasi a single rule associated with the music representation. That is, the rule application of OMClouds applies all constraints together with a uniform rule scope. The score contexts of which this rule scope consists are based the positions of variables in the sequential music representation. For example, constraints can be applied to every single variable or to every variable set consisting of some variable and its next but one successor. In contrast to the rule application mechanisms of PWConstraints and Situation, OMClouds does not provide an easy-to-use means to apply a constraint only to a specific subset of variables (e.g. only on the first and third variable).

[16] In fact, OMClouds feature three representation cases: the solution constitutes either a sequence of variables, a cycle of variables, or a permutation of variables. In all cases the solution is represented by a flat list.

The second case, however, slightly changes the rule application mechanism while the third case enforces an additional constraint. In the second case – the cycle – all elements (including first and last) have a predecessor and successor. In the third case – the permutation – the number of variables in the music representation equals the number of domain values specified and every domain value appears only once in a solution. That way, OMClouds can avoid an `all-different` constraint on the solution which is hard to fulfil for its heuristic search.

[17] Email communication with C. Truchet (23 September 2003).

3.3.3.4. User Interface

Compared with other generic music constraint systems discussed here, the OMClouds user can only solve a limited set of musical CSPs. On the other hand, OMClouds is particularly easy to use. A CSP is fully defined in a visual programming language, which makes the system easy to learn for composers who have little experience in programming. Additionally, the user can supervise the search progress: OMClouds can display the current state of the solution in a graphical score editor.

3.3.4. Other Systems

This sections itemises several further systems. Each of these system allows the user to solve a considerable range of musical CSPs.

CHARM [Wiggins et al., 1989; Harris et al., 1991; Smaill et al., 1993] is a highly expressive music representation. Although CHARM was not designed as a music constraint system, the Prolog implementation of CHARM could be extended into such a system. Pure Prolog itself only supports the equality constraint (unification) and is thus far from being optimised to solve, for instance, numerical CSPs. Yet, there exist several Prolog implementations which extend pure Prolog by constraint programming capabilities (cf. also [Bratko, 2001]). BackTalk [Roy and Pachet, 1997] constitutes a system to solve CSPs over finite domains of Smalltalk objects. BackTalk implements a substantial number of constraint solving algorithms (e.g. several consistency enforcing algorithms to filter variable domains in the first stage of the search process complemented by solution enumeration algorithms including forward-checking, backjumping, and backmarking). The combination of BackTalk and the music representation MusES [Pachet, 1993; Pachet et al., 1996] forms a highly expressive and extendable music constraint system which has been applied to several musical problems (e.g. automatic harmonisation [Pachet and Roy, 1995, 1998]). However, MusES provides only a limited set of score contexts. MusES defines an exhaustive set of temporal relations between score objects and thus temporally related object sets are well accessible. Other relations (e.g. positional relations) are missing in the representation. For example, information such as the 'following note' in a melody is not easily accessible [Pachet, 1994]. Also, the pitch representation of MusES is only suited for tonal harmony and limited to the set of conventional western pitches. MusES explicitly represents enharmonic spelling (i.e. $c\sharp$ and $d\flat$ are distinct pitches). Because of this important capacity, nevertheless, MusES cannot be easily generalised, for example, to represent microtonal pitches.

Alvarez et al. [1998] and Rueda et al. [2001] contribute to theoretical computer science and music constraint programming at the same time. The authors propose *PiCO*, a calculus integrating concurrent processes, object-oriented programming and constraint programming as a foundation for composition systems. This research also developed Cordial [Rueda et al., 1997] – a visual programming language for composers in the spirit of OpenMusic [Assayag et al., 1999] – which shares semantics with *PiCO*.

Arno [Anders, 2000] supports CSPs on the music representation of Common Music [Taube, 1997]. The constraint solver Screamer [Siskind, 1991; Siskind and McAllester, 1993] is employed for this purpose. Arno can express a large number of musical CSPs. In particular, Arno supports polyphonic CSPs as well. However, the system performs an inefficient search except for a specific class of CSPs, namely canons.

OMBT [Truchet et al., 2001; Truchet, a,b] integrates important functionality of the constraint solver Screamer into the composition system OpenMusic [Assayag et al., 1999].

3.4. Conclusion

A rule-based approach is particularly adequate to state music theories (and has therefore been applied for centuries). Constraint programming has proven a well-suited programming paradigm to realise rule-based computer-aided composition systems. Many musical CSPs have already been addressed by means of constraint programming.

Generic music constraint systems pre-define important building blocks shared by many musical CSPs and that way greatly simplify their definition. Several generic music constraint systems has already been proposed. These systems differ considerably in the significant aspects of such a system: they differ in their music representation, the way they define and apply rules, as well as the search technique of their constraint solver.

4. Motivation and System Overview

The present research aims at creating a generic music constraint system, that is, at a system which allows its user to define and solve a large set of musical constraint satisfaction problems. Chapter 1 described the motivation for this research at large, but the present chapter describes this motivation more specifically. This chapter analyses how existing systems are limited in terms of generality and outlines how these limitations are addressed in a more generic system.

Chapter Overview

A number of generic music constraint systems have already been proposed (surveyed above in Sec. 3.3). Section 4.1 shows that these systems support only a restricted set of musical CSPs and analyses their limitations. Based on this analysis, Sec. 4.2 formulates which conditions a more generic system has to meet. Section 4.3 outlines Strasheela's design by explaining how Strasheela meets these conditions (succeeding chapters will discuss the design of Strasheela in detail).

4.1. Limited Generality of Existing Systems

A music constraint system can be very useful even if it is specialised for a narrow range of musical CSPs. For instance, Situation was originally conceived solely for solving a specific set of harmonic CSPs (Sec. 3.3.2, [Rueda et al., 1998; Bonnet and Rueda, 1999]). In this form, Situation was successfully employed by Antoine Bonnet for composing his piece *Épitaphe* (1992–1994, for 8 brass instruments, 2 pianos, orchestra and electro-acoustics).

Situation could be used by other composers at this stage of its development if they were content with Bonnet's aesthetic decisions embodied in the set of predefined rules Situation supported. Yet, composers usually prefer to make aesthetic decisions themselves. Situation was therefore later generalised to cover a larger range of musical CSPs (see Sec. 3.3.2).

A similar tendency from a more specialised music constraint system towards a more generic system can be seen in the redesign of the PWConstraints subsystem score-PMC (Sec. 3.3.1.2, [Laurson, 1996]) into texture-PMC (Sec. 3.3.1.2, [Laurson and Kuuskankare, 2001]).

The present thesis continues this research on how a music constraint system can be made highly generic. To this end, the present section analyses how existing systems are limited

in terms of generality. The key question is: how do existing systems restrict the set of musical CSPs they support?

Some examples of limitations are quite obvious. For instance, OMRC (Sec. 3.2.3, [Sandred, 2003]) is solely devoted to rhythmical CSPs, and score-PMC requires the rhythmic structure to be fully determined in the problem definition. However, there are also many more subtle limitations which will become manifest only in the course of the present analysis

The following sections systematically examine three fundamental aspects of any music constraint system and analyse how existing systems are limited by their design of these aspects. These aspects are the respective music representation (Sec. 4.1.1), their rule formalism (Sec. 4.1.2), and the search strategy underlying these systems (Sec. 4.1.3).

4.1.1. Music Representation

Any musical CSP applies some sort of music representation to represent score information (i.e. either a full score or some excerpt such as a purely rhythmical structure or a harmonic progression). The music representation of a music constraint system has great influence on its expressivity (i.e. how musical CSPs can be expressed).

Certain aspects in this music representation are *variables* (unknowns) in the CSP (e.g. score object parameters such as note pitches or durations). A number of constraints restrict these variables. A solution determines all variables without violating any constraint (Sec. 3.1).

Solution Representation

Section 1.3 introduced the notion of a most generic music constraint system as a system in which any music theory model conceivable can be implemented. The music representation of such a system represents in a convenient way any information its user wants to represent in a solution score. Many existing systems, however, support a music representation which can make it difficult to express complex musical information. For example, the following systems support a purely sequential music representation: the PWConstraints subsystem PMC (Sec. 3.3.1.1, [Laurson, 1996]), Situation, and OMClouds (Sec. 3.3.3, [Truchet et al., 2003]). A sequential music representation is less suited to representing music which features multiple layers that run parallel in time (e.g. parallel voices in polyphonic music).

For example, OMClouds represents music by a sequence of variables where each variable uniformly stands for either an integer or a list of integers. The user can interpret these variables in various ways (e.g. variables may represent pitches or durations). Nevertheless, OMClouds does not support any construct which denotes what different variables mean. In effect, all OMClouds variables in a CSP usually represent the same musical parameter type. Only a small range of CSPs is expressible in a system where all variables represent either pitches *or* all variables represent durations.

Compared to OMClouds, the music representation of Situation is more expressive because the objects in its sequential representation are potentially more structured. The Situation user defines the structure of the objects by specifying a set of points (i.e. variables) and distances between these points. The user freely interprets what points and distances stand for (e.g. pitches and pitch intervals). However, the definition of complex musical CSPs becomes inconvenient because the Situation music representation does not provide any constructs to mark what musical concepts are expressed by its objects and variables (e.g. concepts such as chord, note or pitch). This problem is similar to the problem pointed out for OMClouds above. In contrast, score-PMC defines a data abstraction (Sec. 2.6) with non-ambiguously named accessors (e.g. *getPitch*(*myNote*)).

CSP Definition Support

Existing systems restrict not only what information is expressed by a solution score, but also restrict the information that can be constrained. In a musical CSP, constraints restrict variables which are related to each other in some way: this text introduced the term *score context* to denote an arbitrary set of inter-related score objects (see Sec. 2.4). Each application of a compositional rule constrains variables in one or more score contexts. For example, a melodic rule may constrain the pitches of note pairs that are direct neighbours in a voice or may pose a restriction on a pair of neighbouring local pitch maxima (between these maxima there are further notes in the voice).

In a most generic system, users can freely constrain arbitrary score contexts. Existing systems support only a very limited set of score contexts. For instance, Baroque counterpoint restricts all notes in a voice to pitches of the underlying harmony; exceptions are allowed for a number of specific circumstances including passing notes. A passing note (or passing tone) is a non-harmonic note of weak rhythmic quality that is reached and left by stepwise motion in the same direction.

To check whether a single note forms a passing note requires access to a complex score context which consists of the following variables: the pitch of the note, the pitches of its predecessor and successor notes, the metric position of the note with respect to the measure it belongs to (e.g. either strong or weak beat) and whether or not the note pitch is consonant with respect to the chord the note belongs to. Few systems support the access to such a score context and thus few systems are able to express the passing note concept – a decisive concept in many counterpoint rule sets.

Most systems (e.g. PWConstraints and Situation) are limited to a finite set of predefined constrainable score contexts. For instance, all systems discussed in this text support constraining neighbouring objects in a sequence. However, only a few systems support the constraining of a context consisting of simultaneous objects which belong to different voices.

A system with particual strong support for polyphonic CSPs is score-PMC (Sec. 3.3.1.2) and its is likely the only system capable of accessing score contexts as complex as the context that is required to express the passing note concept. This system explicitly represents and provides access to a number of score contexts. For example, the neighbouring

notes of each voice (melodic context) are collected in a doubly linked list (i.e. a sequential data structure where each element features pointers to its predecessor and successor [Cormen et al., 2001]). This data structure supports accessing one or more melodic predecessors (or successors) of any note. Similarly, another doubly linked list contains the simultaneous notes (harmonic context).[1] Further score context information is accessible by predicates (e.g. a function to check the metric position of a note in a measure, the metric context).

The explicitly represented information in the score-PMC music representation is not user-extendable. For instance, the harmonic context of score-PMC consists only of the actual note pitches. The representation does not explicitly represent more abstract information about the underlying harmony such as the chord root nor does it allow the user to add this information. This restriction complicates the definition of complex harmonic rules (e.g. rules on the chord progression such as a rule which demands a I-II-V-I progression).

Moreover, the internal representation of score contexts in score-PMC makes it difficult to extend the set of supported score contexts. The system predefines a set of score contexts for each note (see Sec. 3.3.1.2) by explicitly storing each context: a score context is not derived from information stored somewhere in the score but each context for each note is stored explicitly as a score-PMC note attribute [Laurson, 1996, p. 225]. The approach is plausible because in the case where these contexts are constrained, score-PMC must access them for every constraint validation during the search process. However, this approach complicates user-defined score contexts, because it requires the user to change the internal details of the system (e.g. it requires replacing the note object by an extended data structure). Yet, restricting the set of available score contexts restricts what musical CSPs can be defined.

4.1.2. Rule Formalism

A music constraint system provides a broad range of constraints (e.g. numerical constraints or set constraints). A user applies these constraints to variables in the music representation to express restrictions on these variables. For instance, a composer may express a melodic restriction which constrains the distance between the pitches of two consecutive notes so that they do not exceed the interval of a fifth (Fig. 4.1). Here, the interval is measured in semitones, seven denoting a fifth.

$$7 \geq |Pitch_1 - Pitch_2|$$

Figure 4.1.: Constrain the interval between $Pitch_1$ and $Pitch_2$ not to exceed a fifth

[1] Laurson [1996] calls the set of all simultaneous notes a *harmonic slice*. In score-PMC parlance, a *harmonic context* is the set of simultaneous notes which are already determined at a certain time during the search process.

The constraint in Fig. 4.1 is only defined for the two specific pitches $Pitch_1$ and $Pitch_2$. In many cases, however, such a restriction should be more general. For instance, a composer may wish that this restriction holds for every melodic interval. Yet, for complex CSPs with a large number of rules and variables, it is of paramount importance to define the CSP in a modular way. Consequently, existing systems introduced the concept of a compositional rule.

A *compositional rule* encapsulates constraints on score objects. As a rule is more general, its application often constrains multiple sets of variables (i.e. multiple score contexts). For example, a compositional rule may encapsulate the expression in Fig. 4.1. The application of this rule may constrain the interval between every pair of consecutive note pitches in any voice. To this end, the rule is applied to each of these score contexts (e.g. each consecutive note pair) by some control structure (e.g. iteration). The term *rule scope* denotes the set of variable sets that is constrained by a certain rule (see Sec. 3.3.1.1).

In several musical styles, specific compositional rules restrict only specific parts of the music. For instance, an imitation – constraining notes at the same position in different voices – often holds only for the first few notes of a section. Another rule – only applied to the end of a section – may cause a cadence.

In a most generic system, the user can freely control the scope of each rule. Existing systems, however, introduce convenient but proprietary rule formalisms which allow only limited control on the scope of a rule.

Many existing systems exhibit a principle problem: they do not separate the definition of a rule from its application. In these systems, the rule definition implicitly defines the rule application. This approach simplifies the rule formalism for the user. Yet, as the rule application is not fully user-controllable, it is limited to the rule application mechanisms provided by the respective system.

The rule formalisms of the two most important existing systems, PWConstraints and Situation, have this problem. These formalisms are now briefly discussed with regard to their generality (see Sec. 3.3 for more details on PWConstraints and Situation at large).

PWConstraints

PWConstraints proposes a pattern matching mechanism to apply rules to the score. This rule application mechanism plays such an important role for the system that subsystems were named accordingly. For instance, the name PMC stands for *pattern matching constraints*.

In PWConstraints, a rule is applied to any set of score objects which matches the pattern matching expression of the rule. This rule application mechanism is highly convenient for a sequential score context (e.g. for melodic rules) but is less suitable for any non-sequential context. However, in polyphonic CSPs many contexts are of a non-sequential nature. Moreover, the pattern matching mechanism cannot be extended by the user and is restricted to cases considered by the system designer.

The system score-PMC (the polyphonic subsystem of PWConstraints) therefore introduces further concepts. In contrast to the more basic PWConstraints subsystem PMC, the score-PMC music representation introduces a data abstraction (Sec. 2.6) with accessors which can be called within a rule definition (Sec. 3.3.1.2). That way, a few further score contexts can be accessed within the rule (namely the sequence of melodic notes, simultaneous notes and the metric position of a note). However, the music representation of score-PMC is not extendable and only predefined contexts are accessible (e.g. no context is predefined for a canon-rule).

To reduce such limitations (which were pointed out by Anders [2003]), Laurson and Kuuskankare [2005] propose a novel syntax extension for score-PMC rules which provides convenient pattern-matching-based access to a number of score contexts such as successive notes in a voice (melodic context), simultaneous notes (harmonic context), all notes in a measure or on a single beat. This means that more score contexts can be expressed by convenient pattern-matching expressions. Still, the user can hardly extend the set of score contexts supported.

Moreover, a pattern matching mechanism is axiomatically limited to what can be expressed by a pattern and the PWConstraints pattern matching language cannot express every possible combination of elements in a sequence (apart from more complex score contexts). For example, a PWConstraints pattern can easily express 'any two neighbouring variables in a sequence': for the sequence $[a, b, c, d]$ the pattern $[*, X, Y]$ matches three times, namely $X = a \wedge Y = b$, $X = b \wedge Y = c$, and $X = c \wedge Y = d$. However, a single pattern cannot express 'any pair of neighbouring variables in a sequence such that pairs do not overlap': for the same sequence $[a, b, c, d]$, there is not a pattern that only matches $X = a \wedge Y = b$ and $X = c \wedge Y = d$.

Situation

The rule application mechanism of Situation is tailored for Situation's sequential music representation: compositional rules are applied to sets of objects which are identified by their numeric indices in the sequence.

For a sequential music representation, any possible rule scope can be expressed by Situation's index-based rule application mechanism. For a unary rule, the rule scope can be specified by a list of indices. For an n-ary rule, the rule scope can be specified by a list of lists of indices. Also, index specifications may use short-hand notations for index ranges (e.g. $[1, 3\_5, 7]$ to denote $[1, 3, 4, 5, 7]$).

Nevertheless, an index-based rule application mechanism is limited to a sequential music representation and positionally related score contexts (e.g. neighbouring notes). This rule application mechanism is insufficient for more complex score topologies (e.g. hierarchically structured scores, see Sec. 2.4). For instance, it would be highly inconvenient to express the context required for the passing note concept (see Sec. 4.1.1) only by score object indices.

4.1.3. Search Strategy

Musical constraint programming problems can be extremely complex, that is, the search space of these problems can be huge. While search strategies such as chronological backtracking (Sec. 3.3.1.3) are practical for problems with a relatively small search space, more complex problems require more sophisticated techniques which can be optimised for the problem at hand [van Roy and Haridi, 2004].

Different musical CSPs require different search strategies to perform the search efficiently. Such strategies may be inspired by the way a composer works with pen and paper. For example, in the process of composing a tonal piece, a composer often sketches the underlying harmonic progression first before writing the actual note pitches. Other composers prefer to fully work out the rhythmical structure of the music first. Polyphonic music is often written 'from left to right' by jumping between the voices and writing the rhythmical structure and the pitches at the same time.

The deeper reason for the many limitations of existing systems are their underlying search strategies. These strategies are optimised for specific problem classes. Indeed, systems are sometimes purposefully crippled and restrict their users to CSPs which they can solve efficiently. For instance, score-PMC does not allow the user to constrain the temporal structure of music, because a determined temporal structure is required by the polyphonic music search approach of score-PMC to compute an efficient static search order (Sec. 3.3.1.3). In Situation, the search strategy only performs constraint propagation – an optimisation technique to distinctly reduce the search space – for its specific flat music representation format (Sec. 3.3.2.3).

Some systems acknowledge the importance of a user-controlled search strategy. For example, PWConstraints allows its user to mix forward-checking rules and the standard backtracking rules in a single CSP [Laurson, 1996]. PWConstraints' forward-checking rules constitute a simple form of constraint propagation. These rules reduce the domain of variables before these variables are visited and determined in the search process (the validity of a 'normal' rule in PWConstraints is only checked after all variables constrained by this rule are determined). However, this approach is not generic. For example, the user cannot change the order in which variables are visited during the search process. Also, changing the search strategy requires changing the problem definition.

Surprisingly, most existing systems come with their own specifically developed constraint solver instead of applying an already established solver.[2] Most systems extend existing general computer-aided composition environments by custom-made constraint programming means. That way, the resulting solver is integrated tightly into the general CAC environment. Yet, this approach is also a reason for limitations of existing systems: ad-hoc developed solvers can hardly compete with solvers resulting from specialised research.

A most generic music constraint system is not optimised in a hard-wired way for a specific class of problems, but allows the user to customise the search strategy for each

[2] PWConstraints includes an interface to the Common Lisp constraint programming extensions Screamer [Siskind and McAllester, 1993], but only as an additional feature [Laurson, 1996].

CSP. Such a system employs a constraint solver in which the user can program the search strategy – independently of the CSP definition.

4.2. The Research Goal

The present research aims to develop a generic music constraint system. On one hand, such a system facilitates the definition of musical CSPs when compared with a general constraint system (e.g. a programming language with support for constraint programming such as Eclipse or Oz). A generic music constraint system predefines the building blocks required for these problems. These building blocks fall into three categories: (i) components for expressing what is known about a score (the music representation), (ii) mechanisms to express compositional rules and their application to the music (the rule formalism), and (iii) mechanisms to express how the search is conducted.[3]

On the other hand, a generic music constraint system allows the user to define and solve a large set of musical CSPs. A most generic music constraint system allows the implementation of any music theory model (Sec. 1.2) conceivable which is fully formalised by a set of rules.

A software application must perform with reasonable efficiency to be useful in practice. Therefore, the present research does not aim for maximum generality. Instead, this research aims to create a system which is highly generic but still reasonably efficient.

The analysis in Sec. 4.1 revealed how existing systems perform with respect to this research goal:

1. The music representations of existing systems can express complex solution scores and provide for a large set of score contexts. However, the range of score information supported by these systems is limited. In particular, complex score contexts – important for implementing many music theories – are hard or impossible to access and constrain.

2. The rule formalisms of existing systems support rule definitions with virtually no restriction, and concise rule application mechanisms capable of expressing complex rule scopes. However, existing rule application mechanisms are not generic: rule scopes important for music constraint programming are not expressible.

3. The search strategies of existing systems allow for the solving of complex musical CSPs. Yet, these systems implement hardwired optimisations which are only suitable for specific CSP classes. The search cannot be adapted for different CSPs. Some systems even support only the CSPs for which they are optimised.

The present research aims to create a system which is more generic than existing music constraint systems but still performs reasonable efficiently.

[3] In many systems, the search strategy is hidden from the user.

In general, a generic music constraint system must be a highly programmable and user-extendable system. A system designer cannot foresee every possible usage (e.g. every required score context or rule scope). The new system must therefore allow the user to adapt and extend the music representation, to define novel rule application mechanisms and to control the search process according to their needs.

Firstly, the music representation of this new system allows the user to define which information the score exhibits. For this purpose, fundamental concepts from previous research on music representation (Chap. 2) are combined into a consistent music representation conceived for music constraint programming. Specifically, the music representation allows the user to constrain arbitrary score contexts by a rule.

Secondly, the new system introduces a programmable and generic rule formalism. The user has full control over how a rule is applied to the score.

Finally, the new system is based on a state-of-the-art constraint system, which allows the user to program how the search is conducted. In particular, it allows for a user-defined dynamic variable ordering (also known as variable selection, i.e. the order in which variables are visited during the search process). CSPs for which a proven variable ordering exist can be solved with this ordering. Moreover, problems which were explicitly excluded by previous systems for efficiency reasons can be solved with suitable novel orderings.

4.3. Strasheela Design Overview

The main contribution of this research is the design of Strasheela, a generic music constraint system. Strasheela constitutes a highly programmable and extendable system. Strasheela supports musical CSPs which were extremely difficult or even impossible to solve in previous systems. Despite its more generic design in comparison to existing music constraint systems, Strasheela still performs reasonably efficiently even on highly complex CSPs.

The present section explains the principle design decisions for the three core aspects of this contribution. Firstly, Strasheela employs a constraint model based on computational spaces and customises this approach for musical CSPs (Sec. 4.3.1, the overview starts by discussing the search strategy, because Strasheela's search approach constitutes its foundation). Secondly, this research applies a well-understood formalism to define and apply compositional rules (Sec. 4.3.2). Thirdly, a rich and generic music representation has been specifically designed for the definition of complex musical CSPs (Sec. 4.3.3). The design of Strasheela will be explained later in more detail (Chap. 5 to Chap. 7).

4.3.1. Search Strategy

The present research applies a constraint model based on computational spaces. Computational spaces encapsulate constraint-based computations (which are speculative com-

putations) and that way make this constraint model programmable at a high level [Schulte, 2002]. The present research applies this model for the first time to a generic music constraint system (to the knowledge of the author). Customised to music constraint programming, this model eliminates many of the performance problems of previous systems and makes it possible to solve musical CSPs which were too complex for these systems. Computational spaces are discussed later in more detail (see Sec. 7.2).

Propagate and Search

The space-based constraint model supports an approach to search called *propagate-and-search* (also known as propagate-and-distribute). The success of propagate-and-search is based on three important ideas [van Roy and Haridi, 2004]. Firstly, the approach maintains partial information. Two pieces of partial information are 'the pitch of a note n_1 is above middle c' and 'note n_1 is lower than note n_2' (4.1).[4]

$$pitch(n_1) > 60 \quad \wedge \quad pitch(n_1) < pitch(n_2) \tag{4.1}$$

Secondly, propagate-and-search performs local deductions (i.e. constraint propagation). For instance, it can be deduced from the previous two partial information pieces that note n_2 is above $c\sharp$ next to middle c (4.2). Constraint propagation considerably reduces the search space.

$$pitch(n_2) > 61 \tag{4.2}$$

Thirdly, propagate-and-search supports a programmable search process [Schulte, 2002]. Whenever no further local deductions can be done, a decision must be made by the constraint solver. Such a decision is potentially wrong, it is possible that it must be taken back. To conduct the search process efficiently, it is therefore of crucial importance to make decisions where a solution is found primarily by local deductions. This approach reduces the risk of making 'wrong' decisions. The power of propagate-and-search lies in the fact that this decision-making process is fully programmable (Sec. 7.2 explains the propagate-and-search approach in more detail).

Score Search

The present research customises the general space-based constraint model for music constraint programming. Strasheela's music representation design allows the decision-making process to fully exploit a partial solution score (e.g. all information accessible in the music representation) – at the time a decision a due. Suitable decisions can thus be automatically computed *during* the search for a solution score.

For example, the search may progress in temporal order from left to right in the score – even if the temporal structure is undetermined in the problem definition. Alternatively,

[4] In this example, pitches are represented by integer keynumbers, 60 represents middle c.

the user may have extended the music representation such that it explicitly expresses a Schenkerian hierarchy [Forte and Gilbert, 1982] of foreground, middleground and background. The user controls the order in which the search progresses. A user-defined decision strategy may first determine the background, then the middleground and foreground before finally determining the actual musical notes.

Applying the space-based constraint model fixes a fundamental search strategy flaw of systems which rely on a fixed and static search order pre-computed before the search starts. The system score-PMC, for instance, does not support constraining the temporal structure of music because this structure is required to pre-compute an efficient search order. Applying propagate-and-search makes it possible for the first time to efficiently solve complex polyphonic CSPs where the rhythmical structure, the pitch structure and arbitrary further parameters are constrained.

This research proposes a number of novel decision making strategies or *distribution strategies* (branching strategies) optimised for various musical CSP classes such as polyphonic or harmonic CSPs. These include strategies which first determine temporal parameters or which determine parameters from 'left to right', i.e. in increasing order of start times of the musical objects they belong to.

Because propagate-and-search decouples the CSP definition and the decision making strategy, users can easily try out different strategies to find the strategy most suitable for their CSP. Furthermore, users can also define their own decision making strategy if so required.

In addition to this, with the propagate-and-search approach the local deductions (constraint propagation) are independent of the data structure. This fixes a fundamental limitation of Situation. Situation – the only generic music constraint system supporting constraint propagation so far (to the authors knowledge) – performs propagation only on a specific sequential music representation format.

The proposed search approach constitutes the foundation of the present research. Strasheela supports very complex CSPs. Solving such complex problems requires efficient search strategies to make these problems feasible in practice.

4.3.2. Rule Formalism

Previous systems introduced the concept of a compositional rule which encapsulates a number of constraints (cf. Sec. 3.3.1.1 and Sec. 4.1.2).

A rule is often applied by a dedicated rule application mechanism – which is part of the rule definition. Common rule scopes are shared by many musical CSPs. For example, in many CSPs a rule is applied to all pairs of neighbouring notes in a melody. Existing systems define convenient rule application mechanisms which cover typical cases. Yet, the rule application mechanisms of existing systems only support specific rule scope cases and exclude others (see Sec. 4.1.2).

Strasheela's rule formalism adopts these two principle ideas from existing systems. Firstly, compositional knowledge is expressed in a modular way by compositional rules. Secondly, the scope of a compositional rule is controlled by a rule application mechanism for convenience.

In addition, Strasheela's design overcomes a fundamental limitation of existing systems. In previous systems, only the compositional rules are freely programmable. In Strasheela, the rule application mechanisms are freely programmable as well. As a result, the Strasheela user can freely control the scope of a compositional rule.

The present research bases its rule formalism on a well-understood formalism. A rule is encapsulated in a function as a first-class value with lexical scope [Abelson et al., 1985]. The application of a rule is consequently a function application.

Unlike previous rule formalisms, the formalism proposed here fully decouples the definition and the application of a rule. That way, the user can freely apply each rule to arbitrary sets of score objects by any control structure.

Strasheela supports the free definition of convenient rule application mechanisms. Such a mechanism is implemented as a higher-order function which expects a rule (i.e. another function) as argument. In contrast to previous systems which usually provide a single and limiting rule application mechanism, the Strasheela user can freely select from a range of predefined rule application mechanisms. The Strasheela prototype already models the index-based mechanisms of Situation (Sec. 6.2.5), the pattern matching mechanism of PWConstraints (Sec. 6.2.6), and several mechanisms not supported by previous systems (e.g. the application of a rule to any score object which meets some user-defined condition, Sec. 6.2.7). Additionally, the user can easily define new rule application mechanisms if so required (this is demonstrated in Sec. 6.2.4).

The rule formalism proposed here forms the basis for several contributions introduced later. For instance, Strasheela's music representation is highly extendable – the programmable rule application mechanism can be adapted for any extensions. First-class functions are also essential for some Strasheela extensions (e.g. the motif model outlined in Sec. 9.1.3.3).

4.3.3. Music Representation

The music representation design for Strasheela is guided by three requirements. The music representation

- Represents the solution score
- Facilitates the expression of the musical CSPs
- Supports the search process

In the present section, the music representation design is motivated by explaining these three requirements. The actual music representation design is only explained later in Chap. 5.

4.3.3.1. Representation of the Solution Score

Any solution returned by Strasheela is expressed in the music representation defined by this system. Consequently, whatever information should be represented in a solution (e.g. the rhythmical and pitch structure of a polyphonic score, motif boundaries, or instrument playing technique specifications) must be representable in the music representation.

Section 2.5 explained that a music representation designer cannot foresee every possible use of the representation and that music representations are therefore often highly programmable and user-extendable. Following this tradition, Strasheela's representation design aims to (i) predefine representation building blocks required for standard musical CSPs and (ii) support user extensions. This way, Strasheela user controls what information is expressed by the representation.

4.3.3.2. Facilitating the CSP Definition

In addition to representing the solution, the music representation of a music constraint system plays also an important role in the actual definition of a musical CSP.

For a generic music constraint system, it is paramount that the user is able to apply a rule to every score context conceivable. This is potentially every possible set of variables in the score.

Moreover, it is important to access score contexts in a way which is easy to remember for the user, because otherwise the definition of complex CSPs becomes impracticable. For instance, the expression of complex score contexts (e.g. the context for a passing note definition) is hard to remember if defined by explicitly specifying numerical indices of score objects. Instead, a score context should be accessed by specifying high-level information such as 'the metric position of *myNote*'. In case such a specification become highly complex it can be encapsulated in a new data abstraction interface function with a name with is easy to remember.

Strasheela provides highly expressive means to access complex score contexts. For instance, Strasheela's music representation is based on the data abstraction concept (Sec. 2.6) and the data abstraction interface includes higher-order functions which can collect all objects in the score fulfilling a user-defined condition.

4.3.3.3. Supporting the Search Process

The third requirement of the Strasheela music representation is related to Strasheela's search approach. This approach allows the user to select a score decision strategy (distribution strategy) optimised for the specific CSP at hand or even to define novel decision

strategies if so required (see Sec. 4.3.1 and Chap. 7). All information available in the score must be accessible to make the definition of arbitrary decision strategies possible.

Strasheela supports the definition of complex score decision strategies which can exploit any information available in the score at the time of the decision. Strasheela compares score parameter objects (e.g. note pitches or voice durations) which encapsulate constrained variables to decide which parameter to distribute. Each parameter object provides access to the score object (e.g. note or voice object) that the parameter belongs to. In turn, each note or voice provides access to the whole score as seen from this score object (e.g. the neighbouring notes in a voice).

5. The Strasheela Music Representation

This chapter introduces the Strasheela music representation. This representation complements concepts established in existing music representation research (and presented in Chap. 2) by the notion of partial information (i.e. constrained variables). The representation was designed with the requirements of a generic music constraint system in mind.

Chapter Overview

After a brief overview of the essential Strasheela score object types (Sec. 5.1), this chapter first discusses fundamental music representation design decisions. The Strasheela representation is based on the concept of data abstraction. Its abstract data types are instances of classes organised in a class hierarchy (Sec. 5.2). The abstract representation is complemented by a textual representation (Sec. 5.3).

Subsequent sections explain the organisation and usage of a Strasheela score, in particular how score information is stored and accessed. Section 5.4 details the hierarchical structure of a Strasheela score and Sec. 5.5 discusses the data abstraction interface. Finally, Sec. 5.6 shows how information required to define complex musical CSPs can be accessed.

5.1. Overview: Score Object Classes

The Strasheela music representation adopts several established music representation concepts which were discussed in Chap. 2. These concepts are implemented by a set of score object types (classes) in Strasheela. Similar score object types can be found in existing music representations (although they possibly differ in terminology).

As an orientation for the remainder of this chapter, Tab. 5.1 provides a brief overview which links fundamental music representation concepts already discussed before with the related Strasheela score object types. Each score type listed is introduced in more detail later in this chapter.[1] The table also references the section of this thesis where the respective concept or type is discussed.

[1] These score object types are documented in detail in the music representation section of the prototype reference documentation at `http://strasheela.sourceforge.net/strasheela/doc/StrasheelaReference.html`.

Type (Class)	Purpose	Section
scoreObject	The most general score data type.	
	Musical magnitudes (cf. Sec. 2.3)	
parameter	A composite data type representing a basic musical magnitude whose value can be constrained.	5.4.1
timePoint	A *parameter* representing a time point numerically (e.g. the start time of a note).	5.4.1.1
timeInterval	A *parameter* representing a time interval numerically (e.g. a duration).	5.4.1.1
pitch	A *parameter* representing a pitch numerically.	5.4.1.1
	Score topology representation (cf. Sec. 2.4.3 and 2.4.4)	
item	A generalisation of the types *container* and *element* (explained below). An *item* has *parameters* and *containers*.	5.4.2
container	An *item* which contains one or more other score *items* (e.g. *notes* or other *containers*).	5.4.3
element	An *item* which does not contain other *items*.	5.4.2
	Temporal objects (cf. Sec. 2.2 and 2.4.3)	
event	An *element* which produces sound when the score is played.	5.4.2
note	An *event* which always has a parameter *pitch*.	5.4.2
sequential	A *container* which expresses that the *items* contained in it follow each other in a sequential manner in time.	5.4.4
simultaneous	A *container* which expresses that the *items* contained in it occur parallel in time (contained *items* can specify a start time offset).	5.4.4

Table 5.1.: Fundamental Strasheela score object types (classes)

5.2. Class Hierarchy

Existing music representations often organise score data types by means of object-oriented programming (OOP, see Sec. 2.7). Strasheela adopts this proven software design approach for multiple reasons. This representation

- Consists of abstract data types (i.e. data types which are defined by their interface).
- Defines a hierarchic typing structure.
- Makes the representation highly user-extendable.

The UML diagram in Fig. 5.1 shows an excerpt of the Strasheela music representation class hierarchy. Depicted are the relations between the classes used to represent temporal information. Most of these classes were briefly explained in Tab. 5.1 and will be explained in subsequent sections.

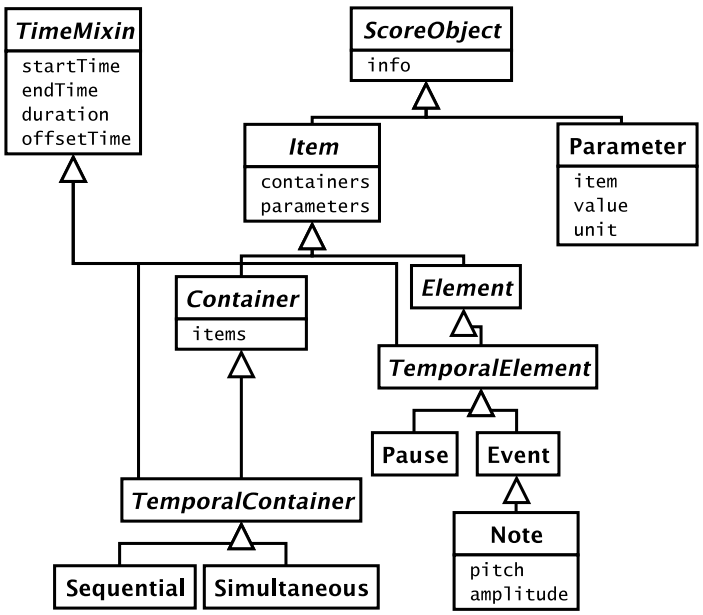


Figure 5.1.: Temporal classes of the Strasheela music representation class hierarchy

The following paragraphs explain further the main reasons why Strasheela uses the object-oriented programming paradigm.

Abstract Data Types

OOP provides a suitable way to define abstract data types (ADT). Section 2.6 explained the advantages of ADTs for a music representation. Data abstraction encapsulates the implementation of data – the data is only used via the data abstraction interface.

An expressive data abstraction interface plays a very important role in the definition of musical CSPs. For example, many CSP require accessing information derived from other information stored in the representation (e.g. to access and constrain the interval between two note pitches). Data abstraction can express such information in a concise and highly legible way. In addition, complex CSPs require to access arbitrary score contexts (as was explained in Sec. 4.1.1).

Hierarchic Typing Structure

The Strasheela music representation defines a large number of score types which are related by is-a relations (e.g. a *note* is an *item* is a *scoreObject*, cf. Tab. 5.1). Many musical CSPs make use of this hierarchic typing structure. For instance, a CSP may collect all instances of the class *item* in the score to constrain them in a certain way – regardless whether these *items* are instances of the *note* class or of some *container* class.

The inheritance scheme of OOP conveniently models such is-a relationships. Subclasses inherit the interface of their superclasses (e.g. the class *pitch* inherits the type check function of its superclass *parameter*) and may extend (and sometimes even overwrite) this interface. This results in a much more concise and modular implementation of the representation than an implementation of separate types would be.

User-Extendable Representation

Furthermore, inheritance allows for convenient user-defined extensions of the Strasheela music representation. A user class can inherit from a Strasheela class which provides a generalisation of the intended functionality.

Because of its extensible nature, the Strasheela design aims to keep the representation core small. The Strasheela representation core provides a number of style-independent concrete representation classes plus abstract classes which make implementing user extensions easier. For instance, the representation core provides concrete classes to represent the temporal structure of music (because music essentially happens in time). Explicit abstract classes such as the element class are described later in this chapter. The Strasheela prototype already provides several extensions of this style-independent representation core which are mentioned in Sec. 9.1.3.

A Side Note: Strasheela is Stateless

Object-oriented programming is often stateful, but the Strasheela music representation is *stateless*.[2] Stateless programming allows for declarative programming (including constraint programming), which is easier to reason about than stateful (or *imperative*) programming. Strasheela's music representation can incorporate constrained variables, that is, *partial* information. During the search process, more information about these variables becomes known (i.e. the size of each variable's domain decreases) and eventually

[2] Within a CSP and for the purpose of this thesis, the Strasheela music representation is purely stateless. Still, the Strasheela prototype (Chap. 9) also defines stateful operations, for example, to edit a solution.

all variables become determined. Still, the value of a variable is never statefully changed during search (i.e. it is not *bound* to any new value) – only their domain is narrowed down. Adding information about a partial value complies with declarative programming, whereas a stateful binding operation would not.

5.3. Textual Representation and Data Abstraction

Section 2.6 compared the respective benefits of a textual music representation and a representation based on the concept of data abstraction. It was argued that a textual representation allows easy editing: this is of great importance for a music composition tool. An abstract representation, on the other hand, can hide the implementation's details: this is crucial when representing something as complex as a score.

The Strasheela design combines the benefits of a textual and an abstract representation. The prime Strasheela music representation is a score ADT (consisting of instances of Strasheela classes), but Strasheela complements this abstract representation with a textual representation form.

Strasheela combines these two representational forms by making them transformable into each other. The textual form is transformed into the ADT by a special constructor function which is part of the score data abstraction interface. This interface also provides a function which transforms the abstract score representation into the textual form.

Transforming the textual representation into the ADT allows the Strasheela user to determine various score settings textually. For instance, the user may textually specify the number of voices in the score, determine the value of certain note parameters (e.g. their duration or pitch), or specify a domain for these parameter values.

Transforming the score ADT back into the textual form can be useful, for instance, to archive a solution score as a text file using the internal Strasheela music representation or even to edit a solution score such that it can form the input for another CSP.

The Textual Representation Format

This paragraph outlines Strasheela's textual representation format. This textual representation is explained fully below, after the main Strasheela classes are introduced (see Sec. 5.4.3.2).

The textual representation uses records to represent Strasheela objects. A *record* is a literate composite data structure consisting of a label, a number of features and feature values (see App. A.4). For instance, a note can be represented as a record with the three features *startTime*, *duration* and *pitch* as in Fig. 5.2.

Hierarchically nested scores are textually expressed with nested records, often in combination with lists (see the list of voice notes in Fig. 5.3). Section 5.4 explains the hierarchical structuring of a score in detail.

$$note(startTime\colon 0, duration\colon 4, pitch\colon 60)$$

Figure 5.2.: A single note, textually represented

Transforming a Textual Form into an Abstract Representational Form

Strasheela proposes a highly flexible transformation from the textual to the abstract representation which is sketched in this paragraph. The function *makeScore* performs this transformation from the textual to the abstract representation. In Fig. 5.3, *makeScore* performs such a transformation on a textual score consisting of a voice with two notes.

$$\langle abstract\ scoreObject\rangle = makeScore(voice(items\colon [note(pitch\colon 60),\\ note(pitch\colon 62)]))$$

Figure 5.3.: The function *makeScore* transforms a textually represented score into a score ADT (a class instance)

The transformation process conducted by *makeScore* is user-controllable. A record label always denotes the meaning of a textually represented score object. For example in *note*(*duration*:1), the label *note* denotes the meaning of this textual object. Usually, *note*(*duration*:1) means an instance of the class *note*. Strasheela predefines the meaning for record labels for all concrete classes in the Strasheela music representation core. However, the user can optionally extend or overwrite these default meanings by mapping any class (or constructor function) to a record label.

In Fig. 5.4, the user has overwritten the default meaning of the record labels *voice* and *note* by mapping a class to each label. Classes are first-class values in the programming language user-interface of Strasheela, as are functions. The freedom to overwrite the meaning of records with a label allows the user, for instance, to call some subsubclass of *note* still simply a 'note' in the textual representation, and to avoid names like *noteWithFeaturesFooAndBar*.

$$makeScore(voice(items\colon [note, note]),\\ types\,(voice\colon \langle default\ voice\ class\rangle,\\ note\colon \langle my\ special\ note\ class\rangle))$$

Figure 5.4.: For each type in the textual score, its class in the ADT (or a constructor function) can be specified as argument to the function *makeScore*

5.4. Hierarchic Representation

This section explains the hierarchic structure of a Strasheela score. The section surveys the most important Strasheela score object classes (which were briefly listed in Tab. 5.1) and their relationships in a score.

The structure of the present section is as follows. Section 5.4.1 introduces the parameter class. The event class and its textual representation is discussed in Sec. 5.4.2. Section 5.4.3 presents the container class, discusses Strasheela's score topologies and provides the full textual representation syntax. Finally, Sec. 5.4.4 describes how Strasheela expresses a temporal hierarchy and concludes with a more complex example of a textually represented score.

5.4.1. The Parameter Class: Variables in the Representation

Section 2.3 introduced the parameter concept as a basic magnitude in the music representation. Typical examples include the parameters pitch and amplitude. The literature also discusses the difficulties in representing parameters. For example, different music representations express parameters in clearly different ways (e.g. using different units of measurement for the parameter values).

Strasheela defines a composite data structure to represent musical magnitudes. The class *parameter* has a number of attributes. The UML diagram in Fig. 5.5 shows these attributes: *info*, *item*, *value*, and *unit*.

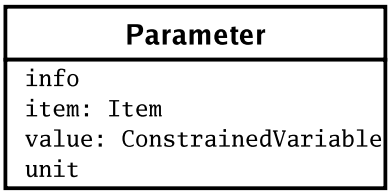


Figure 5.5.: The parameter class

The parameter attribute *value* stores the actual magnitude value of a parameter instance. It is important to note that this attribute points to a *constrained variable*, that is a variable with a domain (e.g. a finite domain integer). A determined value (e.g. a determined integer) is a special case for a constrained variable. In fact, constrained variables occur only as parameter values in the core Strasheela music representation.

The attribute *unit* explicitly denotes the unit of measurement for the parameter value attribute. This information is used, for instance, during the transformation of a score into different output formats (see Sec. 9.1.1). For example, a pitch parameter with the $value = 60$ and $unit = midi$ denotes middle *c*.

The attribute *item* contributes to the hierarchic nesting of a score. This attribute points to the item object (i.e. an event or container, see below) the parameter belongs to. For

example, in the case where the parameter instance represents a note pitch, then the attribute item points to the note instance to which the parameter belongs.

Finally, the attribute *info* stores arbitrary additional information about the parameter. Each score object has an info attribute. The info attribute can contain multiple pieces of information.

An example of parameter instances in the UML is given later together with an item to which these parameters belong (see Fig. 5.9).

5.4.1.1. Parameter Subclasses

Strasheela predefines several parameter subclasses. Parameter subclasses include *pitch*, *amplitude*, *timePoint* and *timeInterval*. These subclasses inherit the four parameter attributes *info*, *item*, *value*, and *unit*. Each parameter subclass supports specific data which can be stored by these attributes.

Parameter subclasses differ in the unit of measurement they support. Typical pitch units are *frequency* and *midi* (i.e. key-number). An example of an amplitude unit is (MIDI) *velocity*. Temporal units include *seconds* and *beats* (i.e. crotchets per second). Beat subdivisions are denoted with the temporal unit specification *beats*(n). This specification expresses that the parameter value n (an integer) stands for a single beat, $2n$ for two beats etc. For instance, with the temporal unit specification *beats*(3) the duration 1 expresses a quaver triplet.

Strasheela defines a number of parameter unit variants with smaller resolution such as *milliseconds* and *midicent* (in *midicent*, 6050 expresses a quarter note above middle c). These units are unnecessary for constraints on reals, but constraints on finite domain integers are particularly well supported by constraint systems which provide constraint propagation such as Strasheela (the consequences of constraint propagation is explained later in Sec. 7.2.3.1 and Sec. 10.1.2). A further pitch unit stands for alternative tunings used in microtonal music. For example, several composers (e.g. Ivan Wyschnegradsky, Franz Richter Herf, and Ezra Sims) applied a temperament which subdivides an octave into 72 equal steps, because such tuning allows for an almost just intonation of 11-limit intervals [Monzo, 2005, article 72-edo].

Some existing music representations define specific parameters with multiple values. For instance, CHARM (Sec. 2.3, [Harris et al., 1991]) represents a single pitch by three values: the note name, the accidental, and the octave. This pitch representation is more expressive than a single value representation but is restricted to Western music. The Strasheela core representation aims to be style-independent and therefore does not provide such complex parameter representations. However, parameter representations like this can be defined by the user, due to the extensible nature of Strasheela. In fact, the Strasheela prototype already defines a constrainable harmony model extension which includes a multiple-value pitch representation consisting of note degree, accidental, and octave (see Sec. 9.1.3.2).

5.4.2. The Event Class

Section 2.2 introduced the notion of an event as a score object which produces sound when the score is played. The Strasheela class *event* features two particularly important attributes: *parameters* and *containers* (see Fig 5.6). Both attributes contribute to the hierarchic nesting of a score. The attribute *parameters* references to all parameter objects which belong to an event. The attribute *containers*, on the other hand, references all containers the event is contained in (the container class is explained shortly in Sec. 5.4.3).

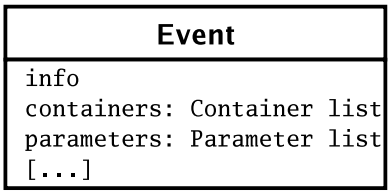


Figure 5.6.: The event class

An event constitutes a very general notion of a sound-producing score object. Because an event simply collects a number of parameters, a hand clapping (with temporal parameters and optionally a specific loudness), a note played on a piano (with a specific pitch, loudness etc.), or an arbitrary sound synthesis language event (possibly with dozens of parameters) are all representable as an instance of the event class (or some subclass of it).

Nevertheless, many Strasheela classes are defined such that their instances are guaranteed to contain a certain minimal set of parameters. For instance, an event always contains four temporal parameters: *offsetTime*, *startTime*, *duration*, and *endTime* (the precise meaning of these parameters and the reasons why this parameter set is not redundant is detailed in Sec. 5.4.4). In such an object, the attribute *parameters* contains this minimal parameter set plus user-specified additional parameters. For convenience, however, such a class usually defines an explicit attribute for each parameter of the minimal parameter set as well.[3] For example, the class *note* – an event subclass – inherits the explicit attributes for the four temporal parameters (start time, duration, etc.) and additionally defines two further explicit parameter attributes *pitch* and *amplitude* (see Fig. 5.7). An example of a note class instance is given below (after its textual representation was introduced, see Fig. 5.8 and Fig. 5.9).

Important functionality of the class *event* is inherited (see Fig. 5.1). The abstract class *item* is an important superclass of both the event class and the container class. The attributes *parameters* and *containers* are in fact inherited from the item class. Another event superclass is the abstract class *element*. Whereas events always produce sound, other element subclasses can be silent. For example, an initialisation statement for a sound synthesis language (such as a f-statement for Csound, which places values in a lookup-table [Boulanger, 2000]) could be represented by a special element subclass.

[3] The additional attributes point to the same parameter objects which are also contained in the parameter list.

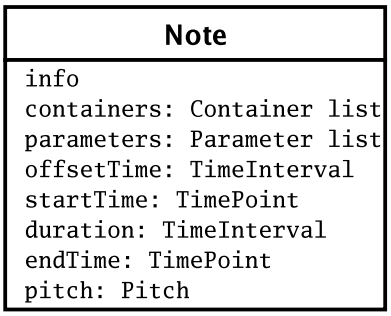


Figure 5.7.: The note class

5.4.2.1. Textual Representation of Events

Strasheela supports a textual music representation which can be transformed into an equivalent abstract representation (see Sec. 5.3). For convenience, parameter objects are not explicitly notated in this textual representation. Instead, the parameter values and parameter units are given directly to their containing object (e.g. a note). The expression in Fig. 5.8 creates a note object which corresponds to the note in common music notation next to it. This example shows how parameter values (e.g. *startTime*, *duration*, *pitch*) are given directly to their containing note. The meaning of the *midicent* pitch unit was explained in Sec. 5.4.1.1.

To express the declarative semantics (or logical semantics) of Strasheela examples, parameter instances as separate objects are dispensable. Instead, it would be sufficient to store the attributes of any parameter instance as attributes of their containing item instance – as suggested by the textual representation. Still, the parameter class abstracts pieces of related information such as the parameter value and its unit of measurement.

To express the operational semantics of Strasheela, on the other hand, it is essential to express parameter instances as separate objects. The search process traverses parameter objects individually and requires accessing arbitrary information about the score as 'seen' from a separate parameter. The operational semantics of Strasheela are detailed in Chap. 7.

makeScore(*note* (*info*: *testNote*,
startTime: 0,
duration: 2,
pitch: 6050,
timeUnit: *seconds*,
pitchUnit: *midicent*))

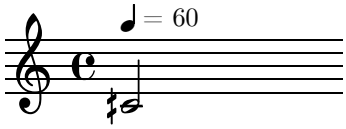


Figure 5.8.: The textual representation of a single note and this note in common music notation

The UML instance diagram in Fig. 5.9 shows the hierarchical structure of a single note:

the note object itself and multiple parameter objects.

It should be noted that the link between the note and its parameters is bidirectional: a note references its parameters (via its attribute *parameters*) and each parameter references the item it is contained in (via its attribute *item*).

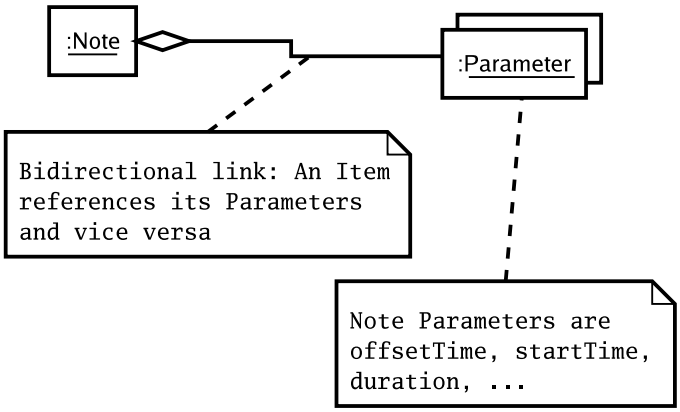


Figure 5.9.: The hierarchic structure of a single note and its contained parameters

For convenience, Strasheela defines default values for the attributes of most of its classes. These attributes are thus optional in the textual representation. For instance, all note parameter values default to a constrained variable (a finite domain integer). The only exception is the *offsetTime* attribute, which defaults to 0 (the reason for this exception becomes clear in Sec. 5.4.4.1). Parameter units also have a default value and are thus optional. For example, the *pitchUnit* default is *midi* (see Sec. 5.4.1.1).

5.4.3. The Container Class

Section 2.4 showed the importance of structuring a score by grouping sets of score objects with the help of containers. An instance of the Strasheela class *container* contains one or more score items (e.g. notes or other containers) and so allows the construction of hierarchically structured scores.

A container features three attributes which contribute to the hierarchic nesting of a score (see Fig 5.10). The container class inherits the attributes *parameters* and *containers* from the class item, and these attributes serve the same purpose as for other item subclasses (e.g. for the event class, see Sec. 5.4.2). The new attribute *items* references to all objects directly contained in the container.

An example of a container class instance is shown later (Fig. 5.11 and Fig. 5.12) after an explanation of the textual representation of containers.

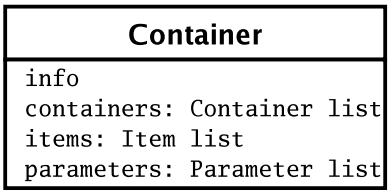


Figure 5.10.: The container class

5.4.3.1. Textual Representation of a Container with Contained Items

Strasheela supports a textual representation for a container and its contained items, which allows the convenient creation of a highly nested score ADT from its textual representation.

The attribute *items* of the textual container representation expects a list of the contained items. Figure 5.11 depicts a short musical fragment and shows how this nested example is notated in the textual form. The example creates a container with several notes (a *sequential* container expresses that its contained items are arranged sequentially in time, see Sec. 5.4.4). In the example, the *timeUnit* specification means that a beat (a crotchet) has the duration 4 (i.e. the duration 1 denotes a semiquaver) and the *pitchUnit* is implicitly set to its default *midi*.

$$
\begin{aligned}
&\mathit{makeScore}(\mathit{sequential}\,(\mathit{info}\colon \mathit{myVoice},\\
&\qquad\qquad \mathit{items}\colon [\mathit{note}(\mathit{duration}\colon 2, \mathit{pitch}\colon 60),\\
&\qquad\qquad\qquad \mathit{note}(\mathit{duration}\colon 1, \mathit{pitch}\colon 62),\\
&\qquad\qquad\qquad \mathit{note}(\mathit{duration}\colon 1, \mathit{pitch}\colon 64),\\
&\qquad\qquad\qquad \mathit{note}(\mathit{duration}\colon 4, \mathit{pitch}\colon 65)],\\
&\qquad\qquad \mathit{startTime}\colon 0,\\
&\qquad\qquad \mathit{timeUnit}\colon \mathit{beats}(4)))
\end{aligned}
$$

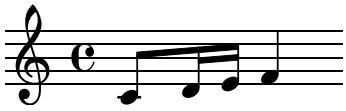


Figure 5.11.: The textual representation of a container with several notes and the corresponding music in common music notation

The resulting hierarchic structure is shown in Fig. 5.12 (the parameter objects of the notes and the container are omitted for clarity). As with the link between an item instance and its parameters, the link between an item and its containers is bidirectional. The 'back-links' (i.e. from an item instance to its container and from a parameter instance to its item) are not specified explicitly in the textual representation but are created

automatically for convenience.[4]

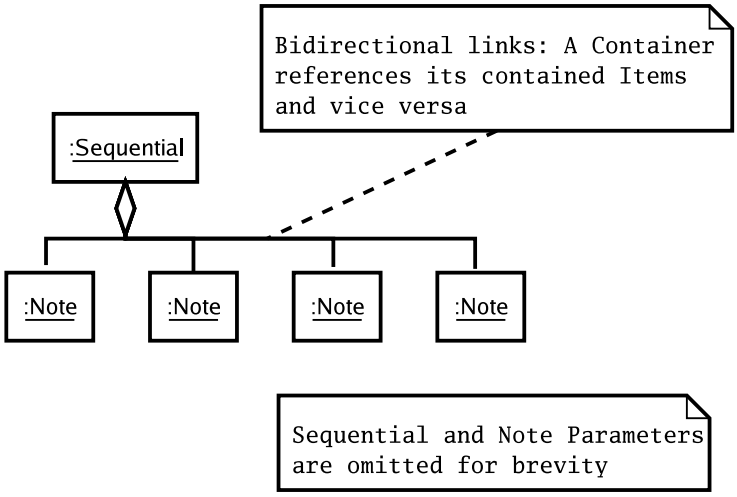


Figure 5.12.: The hierarchic structure of a container with several contained notes

5.4.3.2. The Full Textual Representation Syntax

Section 5.3 explained how the actual meaning of the textual Strasheela music representation is controlled by the user. The user defines the meaning of a textually represented item by mapping its label to either a class or a constructor function. Figure 5.13 presents the syntax of the textual Strasheela music representation more formally.[5]

This syntax specification shows that any score is an item and that an item can have several forms. Each textual item specification (either element or container) starts with a label (usually a symbol) and surrounds a number of feature-value pairs by parenthesis (the specification is slightly simplified by omitting the commas between feature-value pairs). Additionally, a container has a special feature-value pair for its contained items (feature-value pairs can appear in any order, the feature *items* is not necessarily the first). In general, all features of an item are optional (an item without any feature at all can also be notated without the parenthesis such as *note* instead of *note*()).

The textual music representation of Strasheela is actually a composite value of its programming language user-interface (i.e. mathematical notation in this text and the Oz

[4] The containers of an item instance can be specified explicitly, for instance, to express that the item is contained in multiple containers (see Sec. 2.4.4 and Sec. 5.4.3.3).

[5] The syntax is specified in the Extended Backus-Naur Form (EBNF) as used by van Roy and Haridi [2004]. Non-terminal symbols are surrounded by angle brackets as $\langle x \rangle$, literal terminals are surrounded by single quotes as 'x', alternatives are separated by a bar as $\langle x \rangle | \langle y \rangle$, and optional items are surrounded by brackets as $[\langle x \rangle]$. The ellipsis (...) specifies an incomplete EBNF rule, e.g., that at the end of the rule further alternatives are omitted. The suffix * (Kleene star) means that the preceding item is repeated zero or more times whereas the suffix + means that the preceding item is repeated at least once. Multiple symbols can be grouped by round parenthesis to specify a suffix for the whole group.

⟨*score*⟩ ::== ⟨*item*⟩

⟨*item*⟩ ::== ⟨*element*⟩ | ⟨*container*⟩ | ⟨*expression*⟩

⟨*element*⟩ ::== ⟨*label*⟩ '`(`' (⟨*feature*⟩ '`:`' ⟨*expression*⟩)* '`)`'

⟨*container*⟩ ::== ⟨*label*⟩ '`(`' ['`items:`' ⟨*items*⟩] (⟨*feature*⟩ '`:`' ⟨*expression*⟩)* '`)`'

⟨*items*⟩ ::== '`[`' ⟨*item*⟩+ '`]`' | ⟨*expression*⟩

⟨*label*⟩ ::== ⟨*symbol*⟩ | ⟨*variable*⟩

⟨*feature*⟩ ::== ⟨*symbol*⟩ | ⟨*variable*⟩ | ...

Figure 5.13.: Syntax of the textual Strasheela music representation (tree representation subset)

programming language for the Strasheela prototype). Consequently, any expression in this programming language is permitted in the textual representation. For example, subparts of the textual score may be expressed by a variable or generated algorithmically by a function call.

Furthermore, in the textual Strasheela music representation the textual and the abstract representation can be mixed. In the textual representation, an item is either the textual representation of the item or an item instance (i.e. an ADT). The latter option permits a modular definition of large musical CSPs where sub-CSPs on subparts of the score are defined separately. Later, these parts are combined and further constraints are possibly applied to the resulting score.

Such an incremental score construction provides also a convenient means to access the score context out of an existing score. Using this approach, the Strasheela user constructs parts of the score (e.g. individual voices) independently and binds these score parts to identifiers. These parts are then combined into the full score. In effect, each of these identifiers references an individual score context (e.g. a specific voice).

5.4.3.3. Score Topologies

The preceding sections showed that a Strasheela score is a highly nested data structure. Each Strasheela parameter contains a link to its respective item (i.e. an element or a container), each item links to its parameters and containers (which can be *nil*), and each container links to its parameters, containers, and contained items. This nested data structure allows the user to explore the whole score regardless of where in the score this exploration starts.

The Strasheela music representation supports different score topologies (see Sec. 2.4). If it is ignored for a moment that all links are bidirectional, the supported score topologies include the event list (i.e. a flat list of events contained in a container), the tree-shaped

nesting of containers and events, and the nesting of containers and events in an acyclic graph. In an acyclic graph topology, a container can feature any number of items and an item can feature any number of containers.

For the purpose of a generic music constraint system, the acyclic graph topology (Sec. 2.4.4) looks particularly promising at first sight, because it is the most expressive topology. This topology explicitly represents an unlimited number of orthogonal score contexts (i.e. sets of score objects, see Sec. 2.4 for an explanation of this term). Example contexts include the set of notes in a specific voice, the notes in a specific motif, the notes in a specific bar, the set of simultaneous notes in a certain time frame, etc. For a generic music constraint system, it is important to have all these contexts potentially available for the application of rules. When these contexts are explicitly represented by nesting its members in a container, these contexts are indeed easily accessible.

The practical use of this graph topology, however, is limited, because this topology often narrows the set of solutions for a musical CSP too much. For example, the rhythmic structure of a CSP is already very restricted if the hierarchic nesting already determines that a certain measure contains one set of notes, a certain time frame contains another set, and a certain motif contains a further set of notes. Such a situation allows the user to only apply rhythmic rules which do not conflict with the rhythmic structure caused by the hierarchic nesting.

Instead of the graph topology, Strasheela examples usually apply only a tree-shaped topology. This topology realises a pragmatic balance between adding useful information to the score by nesting (e.g. expressing that a certain set of notes belongs to a certain voice in the score) without predetermining the CSP's solution too much. Additional score contexts are then expressed in alternative ways which are detailed in Sec. 5.6 and Sec. 6.3.

However, for certain musical CSPs even the tree-shaped topology narrows the set of solutions too much. For example, the number of notes in a voice is already determined when a voice is represented as a container containing a certain set of notes. Section 8.3 proposes a less restricting approach which allows the user to constrain, for example, the number of notes in a voice by still retaining the additional information expressed by hierarchically nesting these notes in a voice.

5.4.4. Temporal Hierarchy

Research on music representation demonstrated the expressive power of temporal containers which allow the user to explicitly express the temporal structure of music in various ways (see Sec. 2.4.3). The Strasheela music representation allows the user to express and constrain the temporal structure of a score by defining a set of temporal items with temporal parameters. Every temporal element and temporal container features the four parameters *startTime*, *offsetTime*, *duration*, and *endTime*. Like any parameter domain, the domain of each temporal parameter value can be specified by the user: the domain of *startTime*, *duration*, and *endTime* defaults to $\mathbb{N}$ (i.e. a finite domain integer

where the maximum domain is platform dependent, for Oz it is $2^{27} - 2$) and includes 0, the default value of *offsetTime* is 0.

This set of parameters seems redundant at first. For example, if both the start time and the duration of an item are determined, the end time can be calculated and does not need an explicit representation. However, in the Strasheela music representation, parameter values are constrained variables which are not necessarily determined. If these four parameters are represented explicitly, then each parameter can be freely constrained.

Strasheela defines the two temporal classes sequential and simultaneous as subclasses of the container class. An instance of the class *sequential* expresses that its contained items follow each other in a sequential manner in time. Accordingly, an instance of the class *simultaneous* expresses that its contained items run in parallel with each other.

All attributes of the classes sequential and simultaneous are inherited (see Fig. [class hierarchy]) and have been discussed above. Figure 5.14 shows these attributes for the sequential class, the simultaneous class features an identical set of attributes (instance examples with these containers are shown in Fig. 5.11/5.12 and Fig. 5.18/5.19).

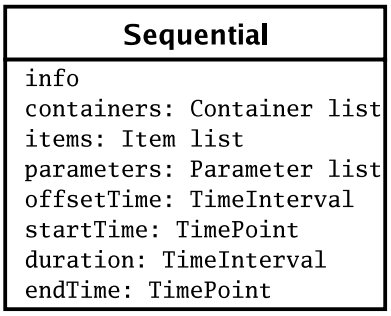


Figure 5.14.: The sequential class

The topology of the temporal hierarchy is a tree, but never a graph: an item must not be directly contained in multiple temporal containers. Otherwise, the temporal structure expressed by different containers would conflict.

5.4.4.1. Implicit Temporal Constraints

Strasheela ensures that the temporal structure of a score is always well-formed – even if the temporal structure is undetermined in a CSP definition. To this end, Strasheela defines a set of implicit constraints on the temporal parameters of temporal items. These are the only implicit constraints which the Strasheela core representation applies to a score; further constraints are explicitly applied by the user.

For all temporal items, Strasheela implicitly constrains the start time, duration, and end time of every temporal item to have the expected relation (Fig. 5.15).

The parameter *offsetTime* expresses a pause before an item (which defaults to 0). The way in which the offset time of an item affects the start time of this item depends on

$$EndTime_{\text{item}} - StartTime_{\text{item}} = Duration_{\text{item}}$$

Figure 5.15.: Implicit constraint for every temporal item

the type of its surrounding temporal container (the offset time has no further influence for a single note without a surrounding temporal container).

The offset time of an item contained in a simultaneous container denotes how much the start time of this item is delayed with respect to the start time of the container. Figure 5.16 shows the implicit constraints between the temporal parameters of the simultaneous container and all its contained items; n denotes the number of items in the container.

$$\begin{aligned}&\bigwedge_{i=1}^{n} StartTime_{\text{simItem}_i} = StartTime_{\text{sim}} + OffsetTime_{\text{simItem}_i}\\ &\wedge\ EndTime_{\text{sim}} = \max(EndTime_{\text{simItem}_1}, \ldots, EndTime_{\text{simItem}_n})\end{aligned}$$

Figure 5.16.: Implicit constraints for every simultaneous container and its contained items

The offset time of an item contained in a sequential container specifies a pause between the item and its predecessor item. For the first item in a sequential container, its offset time specifies how much the start time of the item is delayed with respect to the container start time (Fig. 5.17).

$$\begin{aligned}&StartTime_{\text{seqItem}_1} = StartTime_{\text{seq}} + OffsetTime_{\text{seqItem}_1}\\ &\wedge \bigwedge_{i=1}^{n-1} StartTime_{\text{seqItem}_{i+1}} = EndTime_{\text{seqItem}_i} + OffsetTime_{\text{seqItem}_{i+1}}\\ &\wedge\ EndTime_{\text{seq}} = EndTime_{\text{seqItem}_n}\end{aligned}$$

Figure 5.17.: Implicit constraints for every sequential container and its contained items

The correctness of these implicit constraints requires that all constrained temporal parameters have the same unit of measurement. Therefore, all temporal parameter units are constrained to be equal (i.e. they are unified). For this reason, Strasheela does not define a default temporal unit of measurement (otherwise, a user-specified unit of some temporal parameter and the default unit of other parameters may be inconsistent). This unit is therefore explicitly specified once in every score.

Strasheela also defines the temporal element *pause* as an alternative device to express a temporal gap before any item. Specific musical CSPs are more conveniently defined by using pauses as separate objects. However, often the explicit use of pauses narrows the solution set too much: the user may want to constrain whether or not a pause happens before a certain item. The parameter offset is therefore intended as a more flexible

substitute for the pause object. By constraining the offset of an item, the user can constrain whether the item is preceded by a pause (namely when the *OffsetTime* > 0) and the duration of this pause. Alternatively, the user may place explicit pause objects wherever a pause may occur in the solution and include 0 in the duration domain of these pauses.

5.4.4.2. Textual Representation of the Temporal Hierarchy

The textual representation of nested temporal items corresponds with the general textual Strasheela representation introduced before (see Sec. 5.4.3.2). The next example shows a score which is slightly more nested than previous examples. Firstly, Fig. 5.18 shows the topology of this nesting.

The root of this hierarchy is a simultaneous container which contains two sequential containers. In a polyphonic example, sequential containers can represent voices (cf. Sec. 2.4.3). Finally, each sequential container contains several notes. As before, parameters objects are omitted for brevity.

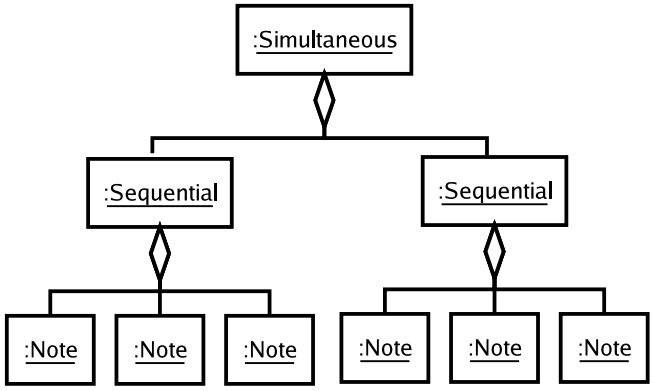


Figure 5.18.: The hierarchic structure of a polyphonic example (represented by a tree-like nesting of containers and elements)

Figure 5.19 shows this polyphonic example in common music notation. In all previous examples, only determined parameter values were specified. In this example, the note pitches are still undetermined and only denoted as a bar between the minimum and the maximum of the pitch domain.

Figure 5.20 shows how such a nested score can be created from the textual representation in a concise way. Previous examples specified the whole score explicitly in textual form. For the present example, an explicit textual form would be unnecessarily long. Therefore, the example first defines the auxiliary function *makeNotes* which returns a list of notes. When the actual score is created (with the call to the function *makeScore*), the function *makeNotes* is called in the specification of each sequential container (the example uses the shortcut labels *sim* and *seq* to denote the two temporal containers).

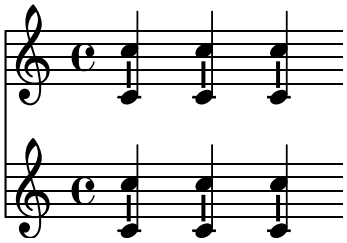


Figure 5.19.: A polyphonic example where the note pitches are undetermined

```
let /* The function makeNotes returns a list with n textual notes. For each note, the duration
       is 1 and the pitch is an undetermined integer with the domain {60,...,72} (created by the
       function fdInt). collectN calls a null-ary function n times and returns the collected results in
       a list: every note is created by a function call to create a fresh constrained variable for each
       note pitch (i.e. multiple notes do not share the same pitch variable).                    */
    makeNotes(n) =def collectN(n, f : f() =def note(duration: 1,
                                                    pitch: fdInt({60, ..., 72})))
in  makeScore(sim (items: [seq (info: alto,
                                items: makeNotes(3)),
                            seq (info: tenor,
                                items: makeNotes(3))],
                   startTime: 0,
                   timeUnit: beats))
```

Figure 5.20.: The textual representation of a polyphonic example (parts of the score are not specified explicitly but created by function calls)

In this example, the pitch value of each note is a constrained variable with a specified domain, namely all integers in $\{60, \ldots, 72\}$ (i.e. the pitches in the octave above middle c – the *pitchUnit* defaults to *midi*). These variables are created with the function *fdInt*.

Section 2.4 showed how a brief Bartók example (Fig. 2.3) can be represented by different score topologies. The topologies based on temporal containers (Fig. 2.6 and Fig. 2.7) can both be reproduced with Strasheela's temporal containers analogously to the examples already shown in Fig. 5.11 and Fig. 5.20.

5.5. The Data Abstraction Interface

Strasheela defines its music representation as abstract data types (Sec. 5.3), organised in a class hierarchy (Sec. 5.2). This section introduces the rich interface of Strasheela score objects.

The structure of the present section is as follows. Firstly, Sec. 5.5.1 discusses the basic interface (defining typical ADT functions such as constructors, accessors and type-checkers). In addition to the basic interface for single objects, Strasheela also defines

interface functions which process multiple hierarchically-nested score objects together (Sec. 5.5.2).

5.5.1. Basic Interface

The interface of an abstract data type often defines a standard set of operators. Typically, an interface subsumes a *constructor*, a *type-checker*, an *equality-checker*, and a number of *accessors* to attributes of the ADT (Sec. 2.6). Strasheela defines all these interface operations for each ADT of the representation (i.e. for each class).

As mentioned above, the Strasheela music representation is purely stateless (Sec. 5.2). Strasheela defines all interface operations as functions (or polymorphic methods) without side-effects. As a stateless representation, however, Strasheela does not define modifiers.[6] Similarly, Strasheela does not define destructors.[7]

The basic constructor for each score class is complemented by the previously introduced score constructor function *makeScore*, which transforms a nested textual score representation into the corresponding nested score ADT (see Sec. 5.3). In effect, the user applies the standard constructors only rarely.

Strasheela defines a hierarchy of types (i.e. a class hierarchy). A type-checker tests whether a certain object is an instance of a certain class or of a subclass of this class. For example, both methods *isNote* and *isEvent* return true for an instance of the class *note*.

Strasheela provides a binary function which tests whether two score objects are equal (i.e. are the idential value).[8]

For each score object, Strasheela defines a set of accessor methods to object attributes. For instance, the expression *getItems*(*myContainer*) returns the list of items directly contained in *myContainer*. Similarly, *getParameters*(*myItem*) returns all parameters of *myItem* in a list.

For convenience, Strasheela defines accessors to single parameters in two versions: one version directly returns the parameter *value* and the other returns the parameter *object*.

[6] The actual Strasheela implementation does indeed define modifiers as well: stateful operations allow the binding of ADT attributes to fresh logic variables. Modifiers are important, for instance, to implement efficient score transformation operations or a graphical score editor. Nevertheless, these examples are outside the scope of this thesis: for the purposes here, the Strasheela music representation is purely stateless.

[7] A Strasheela implementation language will perform garbage collection (as virtually all languages with support for functional programming) and destructors are therefore not needed.

[8] Such an equality checker will often be provided by the implementation language. In case of the Strasheela prototype, Oz provides this function. The Oz operator `==` tests whether two values are equal. For example, the operator returns true when called with two unified variables or two values for which unification would succeed. For values which cannot be unified (stateful values, including class instances) the equality checker only returns true when called with two identical values (e.g. two variables referencing to the same object). However, Strasheela defines the methods `==` and `unify` for score objects. These methods internally transform the – possibly nested – score object into its textual representation and then perform an equality check or unification.

The name of the first accessor simply reflects the parameter name, the latter accessor usually ends with the additional suffix *Parameter*. Thus, *getPitch*(*myNote*) returns a constrained variable representing the note pitch value and *getPitchParameter*(*myNote*) returns the pitch parameter object.

Strasheela defines many more accessor functions, which perform more complex processing operations instead of simply accessing the direct attributes of an object. Some of these functions are introduced in the subsequent sections.

A special accessor method is *hasThisInfo* which returns a boolean value. In Fig. 5.21, *hasThisInfo* checks whether one of the information chunks stored in *myNote* equals *marker*. Additional information chunks can be added to a score object with the method *addInfo*.

hasThisInfo(*myNote*, *marker*)

Figure 5.21.: The method *hasThisInfo* checks whether the *info*-attribute of a score object contains a specific piece of information

5.5.2. Generic Interface for Hierarchic Structures

The previous section surveyed interface functions which process single score objects. In addition, Strasheela features a number of interface functions which process multiple objects which are hierarchically nested in a score. Many of these functions are defined in a highly generic way as higher-order functions (see Sec. 2.8).

5.5.2.1. Processing Lists of Score Objects

Several accessors of the interface return a list of score objects. For instance, the items contained directly in a container can be accessed by the method *getItems* which returns these items in a list. Similar methods exist to access all parameters or direct containers of an item in a list.

Such a list can be processed by any list function defined in the Strasheela programming language user-interface (e.g. the Oz language in case of the Strasheela prototype). Many list functions are higher-order functions, which allow the user to process a list of score objects in a very concise and powerful way. For instance, the function *filter* provides a powerful device to select specific subsets of list elements: *filter* returns all elements in a list for which a predicate – a function given as an argument (often simply defined inline) – returns true.

Many higher-order list functions are available. Typical examples include *map* (collecting the results of a unary function applied to any list element), *count* (counting the number of elements for which a predicate returns true), *find* (returning the first list element for

which a predicate returns true), and *sort* (sorting a list according to a binary predicate used to compare two list elements),

Figure 5.22 demonstrates how these functions express a complex selection process in a concise way. In the example, the function *filter* processes a list containing all items which are directly contained in *myContainer*. This function selects only those items for which the test function f returns true, that is, those which are note objects and which are marked with a specific piece of information. The result of *filter* is further processed by the function *map*, which collects the pitches of the selected notes.

$$\begin{aligned} &map\,(filter\,(getItems(myContainer),\\ &\qquad f : f(x) \stackrel{def}{=} isNote(x) \wedge hasThisInfo(x, takeMe)),\\ &\quad getPitch) \end{aligned}$$

Figure 5.22.: A complex selection defined concisely: the expression returns the pitch values of all objects directly contained in *myContainer* which are notes and marked with the information *takeMe*

Because such selections occur frequently in musical CSPs, Strasheela defines a shorthand method *mapItems* which combines *getItems* with the functions *map* and *filter*. The example in Fig. 5.23 is equivalent to Fig. 5.22. Similar shorthand methods exist for the combination of *getItems* with other list functions (e.g. *countItems*, *findItem*, and *filterItems*).

$$\begin{aligned} &mapItems\,(myContainer,\\ &\qquad getPitch,\\ &\qquad test\text{: } f : f(x) \stackrel{def}{=} isNote(x) \wedge hasThisInfo(x, takeMe)) \end{aligned}$$

Figure 5.23.: A shorthand alternative of the expression in Fig. 5.22

For better readability, *mapItems* expects the predicate for filtering as a *named argument* (also known as keyword argument), called *test* (for the notation cf. App. A.2). The argument *test* is optional (the test defaults to $f(x) \stackrel{def}{=} true$).

5.5.2.2. Interface for Score Hierarchy

Strasheela complements the interface for processing a list of item instances (as discussed in the preceding section) by a similar interface for processing objects in the whole hierarchically-nested score.

All Strasheela items understand the higher-order accessor method *collect*. Similar to accessors like *getItems* and *mapItems* (see above), *collect* returns a list of score objects. However, whereas the former accessors only process the items directly contained in a

container, *collect* traverses recursively through the score hierarchy and potentially collects all score objects in the whole score. For example, the expression in Fig. 5.24 returns all pitch parameter objects in *myScore*, regardless of nesting depth. The optional named argument *test* serves the same purpose for *collect* as it does for *mapItems*: it expects a predicate used for filtering the list of score objects. The list *collect* returns can be processed by arbitrary list functions (e.g. those discussed in the previous subsection).

$$collect(myScore, test\colon isPitch)$$

Figure 5.24.: Collecting all pitch parameters in the whole score

Strasheela defines a number of similar higher-order methods traversing the whole score which mirror common list-processing functions. For example, Strasheela defines methods traversing the whole score which are equivalents of the the list functions *map* and *filter*.[9] The example in Fig. 5.25 collects the pitch values of all notes in the whole score which are marked by a certain information specification (the very similar example in Fig. 5.23 – using *mapItems* instead of *map* – returned only the pitches of the directly contained notes).

$$\begin{aligned} &map\,(myScore,\\ &\quad getPitch,\\ &\quad test\colon f : f(x) \stackrel{def}{=} isNote(x) \wedge hasThisInfo(x, takeMe)) \end{aligned}$$

Figure 5.25.: Collecting the pitch values of all notes in *myScore* (regardless of nesting depth) which are marked with the information label *takeMe*

5.6. Score Contexts

Section 4.1.1 demonstrated why a generic music constraint system should allow for the application of constraints on arbitrary score contexts. In order to apply a constraint, a score context must be somehow accessible. Also, it is important to express score contexts in a way which is easy to remember for the user, in order to make complex CSP definitions comprehensible (Sec. 4.3.3.2).

This section first summarises general principles for expressing score information in Strasheela (Sec. 5.6.1). Section 5.6.2 shows how the Strasheela user defines new score contexts by utilising this information. Section 5.6.3 concludes with a discussion of the generality of this formalism and also how this generality is limited (Sec. 6.3 resolves this limitation).

[9] Methods such as *map* and *filter* are polymorphic functions. The Strasheela methods traversing a score expect a Strasheela item instance as argument, whereas the list functions expect a list.

5.6.1. Principles for Expressing Score Information

Strasheela supports several basic principles to store score information. Some information is stored explicitly, while other information can be derived from explicitly stored information.

Explicitly Stored Information: The following principles are supported to explicitly express information about the score.

- The data type (class) of a score object is represented explicitly. Types are hierarchically organised (class hierarchy). For example, the duration of a note is a time interval, which is a parameter, which is a score object.
- Attributes of a score object are stored explicitly. For example, the value of a note pitch is stored explicitly.
- The value of a score object attribute can be a constrained variable. Such an attribute value is explicitly represented but can be unknown and can be constrained. For example, the pitch value of a note or the duration of a temporal container can be unknown but constrained.
- The nesting of score objects in arbitrary score topologies is stored explicitly. For instance, the representation explicitly represents which items are contained in a container. Moreover, the hierarchic nesting is expressed by bi-directional links between involved score objects (Sec. 5.4.2.1 and Sec. 5.4.3.1).
- The order of score objects on the same hierarchical level is explicitly represented. For example, a simultaneous container representing an event list preserves an order of its contained items which is independent of their temporal order (i.e. each item has its own offset time representing the difference between its start time and the start time of the container).

Derived Information: Derived information can be deduced by utilising any explicit or otherwise derived information about score objects. Deducing information which involves multiple score objects often makes use of the bi-directional links between score objects. For instance, the interval between the explicitly represented pitches of two neighbouring notes in a sequential container can be deduced even if only one of the two notes is given.

These fundamental principles of explicitly stated and derived information are completed by principles which make the representation more convenient.

Abstract Data Types (ADT): Score objects are defined as abstract data types (classes) which encapsulate their attributes and provide a rich user interface (Sec. 5.5).

- Strasheela defines a number of basic score data types which cover many musical CSPs and provides a standard interface for these predefined types (object creation, type checker, attribute accessors etc., Sec. 5.5.1).

- In addition, the interface features a number of generic higher-order functions (methods) to process information in the score (Sec. 5.5.2). For example, this interface allows the user to concisely define arbitrary score context accessors by collecting all score objects for which a test function returns true (cf. Fig. 5.24).
- Derived information can be encapsulated into new interface functions. For instance, the interface provides functions such as *getSuccessor* which returns the successor of an item instance in a container.

Incremental ADT Definition (Class Inheritance): Strasheela supports an incremental definition of ADTs. The basic Strasheela types are designed with various extensions in mind. New types (sub-classes) are defined by inheriting from existing types (super-classes). Only the required additional attributes and interface functions (or functions to be overwritten) must be defined by the user. For example, the prototype defines classes like *chord* and *scale* (to support harmonic CSPs) which inherit from the class *temporalElement*.

Textual Representation: Strasheela features a textual representation form which allows the user to define various score topologies and to concisely express fully determined scores, or scores which leaves all variables undetermined, and anything in between (Sec. 5.4.3.2). This textual representation is a composite datum of the interface language and can be created by other programs. The textual representation format and its ADT representation equivalent can be transformed into each other.

The Strasheela user ultimately controls the information that is represented in a score. The user combines instances of predefined score data types (class instances) to build a music representation. For instance, textual representation makes it easy to assemble the score from different score data types (and to implicitly define the score topology). Optionally, the user extends the interface of these types or even defines new types. Class inheritance greatly simplifies the definition of new types.

The fundamental principles used to represent explicit information are not strictly orthogonal. Some of these principles are interdependent. For example, different data types often differ also in their set of attributes, the nesting of score objects is essentially stated in terms of attributes, and the nesting of objects must correspond with the type of these objects. Morever, the same information can be stored in alternative forms. For example, the information related to hierarchic typing and the hierarchic nesting could alternatively be expressed by attributes. Hierarchic typing could be expressed by an attribute which stores a list of type and super-type labels. Hierarchic nesting could be expressed by an attribute which stores a label denoting a score context such as in_voice_1.

Nevertheless, substituting some representation form by another form can result in a less concise representation. Hierarchic nesting with bidirectional links allows the container to access *all* information within its contained objects and vice versa. For example, a

contained object can access its own position in the container. Representing all this information by additional attributes only (e.g. by additional attributes for every 'contained' object where these attributes store all the information about their 'virtual container') would result in a cumbersome representation. Moreover, representing hierarchic typing only by an attribute would not provide for the convenience of incremental type definitions by inheritance

5.6.2. User-Defined Score Contexts

Section 2.4 defined a score context as a set of score objects which are related. The relation between the elements defines the context. Typical relationship examples are 'simultaneous notes' or 'neighbouring chords in a chord progression'.

5.6.2.1. Accessing Score Contexts

The definition of a score context utilises the explicitly stored and derived information provided by Strasheela's music representation which was discussed above. For example, the music representation can explicitly express which notes belong to a certain voice by means of hierarchic nesting where the voice is represented by a sequential container *myVoice* and all voice notes are items contained in *myVoice*. The context 'all notes in the voice *myVoice*' is then accessed by accessing the items contained in *myVoice* (see Fig. 5.26).

$$getItems(myVoice)$$

Figure 5.26.: Accessing the context 'all items in the voice *myVoice*'

As this example shows, a context is accessed by an expression in Strasheela's programming language user interface (i.e. mathematical notation in case of Fig. 5.26). More complex expressions allow the user to access more complex score contexts. For example, the voice *myVoice* may contain both note instances and pause instances. The example in Fig. 5.27 accesses the context 'all items in the voice *myVoice* which are pauses' by filtering the items contained in *myVoice*. This context makes use of information derived from information expressed by hierarchic nesting and expressed by typing.

$$filter(\ getItems(myVoice),\ isPause)$$

Figure 5.27.: Accessing the context 'all items in the voice *myVoice* which are pauses'

In both examples a function call returns the full context (i.e. a list of all the items or pauses in *myVoice*). There are other cases in which the return value of a function is only

a subset of all score objects which belong to the full context: the missing part is often the argument to the function. For example, the context 'two neighbouring items in a temporal container' can be created by accessing the successor item of a given item (see Fig. 5.28). This context is crucial for the application of numerous melodic rules (e.g. to constrain the pitch interval of two neighbouring notes).

$$\{myItem, getSuccessorInTemporalContainer(myItem)\}$$

Figure 5.28.: The context 'two neighbouring items in a temporal container'

The successor of an item is not represented explicitly (i.e. there exists no explicit successor attribute), but this information can be derived from the information provided by the hierarchic nesting due to the bidirectional links between score objects (the function *getSuccessorInTemporalContainer* is defined in Fig. 5.29).

5.6.2.2. Abstracting Score Contexts

The expression which returns a score context (or complements a context) can be encapsulated in a function. Abstracting a score context definition this way has the same advantages as function definitions have for programming in general (e.g. the program is structured into modular subprograms with names which are easy to remember).

It is important for Strasheela as a generic music constraint system, that context accessor functions can be defined by the user. That way, the user can extend the set of predefined contexts. The following paragraphs introduce two typical techniques used to define such functions. The first technique relies on information expressed by hierarchic nesting to access context elements directly. The second technique defines a test which checks for context membership and filters the context elements out of score objects.

Exploiting Hierarchical Information

Strasheela defines several context accessors which evaluate the information provided by the hierarchic nesting of the score. These functions often make use of the bidirectional links between score objects. Because of these bidirectional links, the whole score can be explored as it can be 'seen' from any object.

The function *getSuccessorInTemporalContainer* expects an item instance as argument and returns the successor of this item in its temporal container. The function exploits the information provided by the hierarchic nesting of the score. Figure 5.29 shows how this function can be defined.[10]

[10] Like all examples in this text, the definition of *getSuccessorInTemporalContainer* in Fig. 5.29 aims for clarity and not efficiency. The definition could be optimised such that the list of items in the container is not traversed three times (by *position*, *length* and *nth*). Possible further improvements include replacing all lists in the Strasheela music representation by data structures with constant time access (e.g. arrays or records) [van Roy and Haridi, 2004] and memoizing [Norvig, 1992] the position of an item in a container.

$$
\begin{array}{rll}
\mathit{getSuccessorInTemporalContainer}(x) \stackrel{def}{=} & \textbf{let} & c \stackrel{def}{=} \mathit{find}(\ \mathit{getContainers}(x), \\
 & & \qquad\qquad \mathit{isTemporalContainer}) \\
 & & \mathit{items} \stackrel{def}{=} \mathit{getItems}(c) \\
 & & \mathit{nextPos} \stackrel{def}{=} \mathit{position}(x, \mathit{items}) + 1 \\
 & \textbf{in} & \textbf{if} \quad \mathit{nextPos} > \mathit{length}(\mathit{items}) \\
 & & \textbf{then} \ \mathit{nil} \\
 & & \textbf{else} \quad \mathit{nth}(\mathit{items}, \mathit{nextPos})
\end{array}
$$

Figure 5.29.: The context accessor *getSuccessorInTemporalContainer* returns the successor of an item x in its temporal container

The function first accesses the temporal container of the item x. Because each item can potentially be contained in multiple containers (in case the score topology is a graph), a filtering of all containers of x is performed. It is sufficient to find only the first temporal container c (hence the function *find*), as an item cannot be contained in multiple temporal containers (see Sec. 5.4.4).[11] The list of items contained in this container c is accessed and bound to *items*. The position of the successor of x in *items* is computed and bound to *nextPos*.[12] After a check whether the list *items* is long enough to hold a successor item at position *nextPos*, this successor item is finally accessed and returned.[13] *getSuccessorInTemporalContainer* returns *nil*, in case no successor item exists.

Note that the score object returned by *getSuccessorInTemporalContainer* is the *positional* successor of the given object in its temporal container. However, it is *not* necessarily the *temporal* successor. For example, the items contained in a simultaneous container are not necessarily ordered temporally.

Several score contexts can be expressed by hierarchically nesting score objects in a score. Examples include all part-of relations (e.g. the notes in a motif, voice, chord etc.), has-a relations (accessing item attributes and parameters) and positional relations (e.g. neighbouring items). However, other contexts require different approaches: the next paragraph explains another technique.

Filtering Context Members

An alternative technique closely resembles the common set-builder notation. For example, the context 'all temporal items in the score which are simultaneous to some item *item*' can be notated by set-builder notation as in Fig. 5.30. This context of simultaneous items is important for many rules (e.g. to apply harmonic rules).

In the following, this set-builder notation example is translated into a Strasheela program. The main idea here is to define a function which returns true for any context

[11] The function *isTemporalContainer* returns *true* for a temporal container.
[12] The function *position* returns the position of an element in a list.
[13] The function *nth* returns the list element at the specified position.

$$\{x : isTemporalItem(x) \\ \wedge isSimultaneous(x, item) \\ \wedge x \neq item\}$$

Figure 5.30.: The context 'all temporal items in the score which are simultaneous to some item *item*' in set-builder notation

element and to use this function to filter the context members out of score objects. This filtering is potentially performed on the set of all objects in the score.

The function *getSimultaneousItems* (Fig. 5.31) encapsulates the context 'all temporal items in the score which are simultaneous to some item *item*' in a function (i.e. this function is equivalent to the set-builder notation example of Fig. 5.30).

$$\begin{array}{lll} getSimultaneousItems(item) \stackrel{def}{=} & \textbf{let} & \textit{/* access temporal top-level container of } item \textit{ */} \\ & & top \stackrel{def}{=} getTemporalTopLevel(item) \\ & & \textit{/* auxiliary test function */} \\ & & f(x) \stackrel{def}{=} x \neq item \\ & & \qquad \wedge isTemporalItem(x) \\ & & \qquad \wedge isSimultaneous(item, x) \\ & \textbf{in} & collect(top, test{:}\, f) \end{array}$$

Figure 5.31.: The context 'all temporal items in the score which are simultaneous to some item *item*' in Strasheela

The definition of *getSimultaneousItems* accesses first the top-level temporal container of the argument *item* using the function *getTemporalTopLevel*.[14] Then, *getSimultaneousItems* defines an auxiliary function f which expresses the conjunction of three tests – much like the expression in Fig. 5.30. The function tests whether its argument x is unequal to *item*, whether x is a temporal item, and whether x and *item* are simultaneous in time. Finally, *getSimultaneousItems* collects all score objects recursively contained in *top* which fulfil the test function f.

The example requires three boolean functions. The type-checking (*isTemporalItem*) and the negation of object equality is provided by the basic Strasheela interface. The function *isSimultaneous* returns a boolean reflecting whether two temporal items are simultaneous in time or not (see Fig. 5.32). Although *isSimultaneous* is predefined by Strasheela, it is defined here to demonstrate how such functions can be defined by the user. The function tests whether the time frames occupied by two temporal items – which are denoted by the start and end time of each item – are overlapping.

[14] The function *getTemporalTopLevel* returns the top-level of the temporal hierarchy which contains a given item. It is defined in Fig. B.1

$$isSimultaneous(item_1, item_2) \stackrel{def}{=} (getStartTime(item_1) < getEndTime(item_2)) \\ \wedge (getStartTime(item_2) < getEndTime(item_1))$$

Figure 5.32.: The function *isSimultaneous* tests whether two items are simultaneous

5.6.3. The Generality of the Strasheela Score Context Formalism

The Strasheela music representation is highly generic. The Strasheela user ultimately controls what explicit information is stored in the score. The user can then freely deduce derived information from the explicitly stored information (Sec. 5.6.1). In particular, any score information can be obtained from any object in the score due to the bidirectional links between score objects.

The generality of the music representation results in a highly generic score context formalism. When accessing a score context, the user can freely make use of any information available in the score. For instance, Sec. 5.6.2.2 introduced two techniques to access score contexts – exemplified by the functions *getSuccessorInTemporalContainer* (Fig. 5.29) and *getSimultaneousItems* (Fig. 5.31). Both techniques derive information from explicitly stored information to access specific score contexts (and make use of the bidirectional links between score objects to access this information).

For accessing a score context, it is essential that the music representation provides enough information to unambiguously specify the set of score objects which belongs to this context (to *isolate* this context). Any score context is accessible in Strasheela – as long as the music representation provides the required information for isolating this context.

There are two cases where the music representation does not provide the required information to isolate a score context. In the first case, the music representation misses some explicitly stored information. For example, polyphonic music can be represented in a flat event list (e.g. all events are directly contained in a simultaneous container and the start time of events equals their offset time). However, in such a music representation the information 'which note belongs to which voice' is missing and the score context 'all notes of the first voice' cannot be accessed.

In the case where the music representation misses some explicitly stored information, the user can restructure or extend the representation. For instance, the user can restructure or extend an event-list representation of polyphonic music in several ways in order to make the score context 'all notes of the first voice' accessible. The user can either add an information tag to each event (e.g. using the *info* attribute) to denote the voice it belongs to, re-organise the hierarchic nesting of score objects to express voice membership, or even define a new class which extends the event class by a voice-membership attribute.

In the second case, the music representation explicitly stores the required information, but this information is not determined in the CSP definition. If so, the necessary information will only be determined during the search process. For example, simultaneous

objects are not accessible in the CSP definition when the rhythmical structure of a polyphonic score is undetermined and is constrained by rules as well. In such a case, it is impossible in the CSP definition to directly access the score context in question. As a solution to this problem, Sec. 6.3 discusses techniques that apply compositional rules to score contexts which are inaccessible due to undetermined information in the problem definition.

6. The Strasheela Rule Formalism

The preceding Chap. 5 discussed how a score is expressed in the Strasheela music representation. The present chapter explains how a musical constraint satisfaction problem is stated in Strasheela: this chapter introduces Strasheela's concept of constraints and rules and details how constraints and rules are applied to a score.

This introduction only provides the declarative semantics of a musical CSP in Strasheela in order to make this explanation more easy to comprehend. The declarative semantics is based on a programming model which subsumes two programming paradigms well-known in the computer music community: constraint programming (Sec. 3.1) and functional programming, including higher-order functions (Sec. 2.8). Examples are notated in a way that has already been used throughout this thesis. This notation uses first-order logic notation complemented by the where-notation to notate functions as first-class values (App. A).

The operational semantics of Strasheela, however, are based on a different programming model. Whereas this section explains the declarative semantics with the notion of functions, Strasheela's operational semantics is based on concurrent procedures (which are also first-class values). Nevertheless, the notation of concurrent procedures would have required a notation which has been used little so far in computer music research.[1] Instead, the present chapter uses a common mathematical notation, although this notation covers only the declarative semantics. Strasheela's operational semantics will be explained in Chap. 7.

Chapter Overview

Section 6.1 defines what a constraint is and what a composition rule is in Strasheela. Strasheela's rule formalism allows for user-defined rule application mechanisms, which is demonstrated in Sec. 6.2. The last section discusses an issue which arises in many complex musical CSPs. These problems constrain score contexts which are inaccessible in the problem definition because the partial music representation misses certain information. Section 6.3 discusses three different approaches to address this issue.

[1] Concurrent procedures can be notated, for example, with the π-calculus notation [Parrow, 2001] or the Oz notation [van Roy and Haridi, 2004], and these notations could have been used in the present chapter. The literature discussing *PiCO* – a calculus which subsumes concurrent objects and constraints to provide a foundation for composition systems – even proposes its own syntax and expresses musical examples in this notation [Rueda et al., 2001].

6.1. Constraints and Rules are First-Class Functions

In the declarative semantics of Strasheela, a *constraint* is a function which returns a boolean value. All arguments of a constraint are constrained variables (i.e. unknowns as the variables in first-order logic for which a domain of possible values is defined, see Sec. 3.1). Also the boolean value returned by a constraint is a constrained variable (i.e. its value can be unknown).

The variable values given to the constraint as arguments satisfy the constraint's restriction if and only if the returned value is true. By convention, the truth values are represented by integers: 1 represents *true* and 0 represents *false* (the reason for this convention will become clear later, see Sec. 7.2.3.4).

Formally, a constraint $c(X_1, X_2, \ldots, X_n)$ maps the set of its variable domains (here $\{D_1, D_2, \ldots, D_n\}$) to a truth value as in

$$c : D_1 \times D_2 \times \cdots \times D_n \mapsto \{0, 1\}$$

For convenience, a constraint can be a composite expression. The operations in such a composite expression (notated with the common mathematical symbols) may return other values instead of truth values. However, the full expression returns a truth value. For example, the interval between two pitches can be constrained by the composite expression shown below, where the operations $X - Y$ and $|X|$ return numeric values instead of truth values.

$$(Interval = |Pitch_1 - Pitch_2|) \Leftrightarrow \{0, 1\}$$

A *compositional rule* is a concept very similar to a constraint. Like a constraint, a rule is a function which returns a boolean variable. However, a rule is a slightly more general concept when compared with a constraint. Whereas all arguments of a constraint are variables, the arguments of a compositional rule can be any score data. This implies that any constraint is also a compositional rule.

A rule applies constraints to some variables in a CSP. These variables are either directly given to the rule as arguments (in that case the rule is actually a constraint) or are accessible from its arguments.

The example in Fig. 6.1 defines two variants of a rule which encapsulates the melodic restriction that the interval between two pitches must not to exceed a fifth (cf. Fig. 4.1). The example defines two variants for this rule. The constraint (and rule) $limitInterval_{pitches}$ is a function which expects two pitches which are variables in a CSP. The rule $limitInterval_{note}$, on the other hand, expects a single note object: the rule accesses the successor note of a given note, and the pitches of both notes, and then applies the constraint $limitInterval_{pitches}$ (i.e. the variant constraining two variables) to these pitches.[2]

[2] As explained in App. A.1, variables in the CSP start with an upper-case letter (e.g. $Pitch_1$) and other

$$limitInterval_{pitches}(Pitch_1, Pitch_2) \stackrel{def}{=} 7 \geq |Pitch_1 - Pitch_2|$$

$$limitInterval_{note}(note) \stackrel{def}{=} limitInterval_{pitches}\,(getPitch(note), getPitch(getSuccessor(note)))$$

Figure 6.1.: Two rule variants which constrain the interval between two pitches not to exceed a fifth

Rules can be composed from other arbitrary functions of the system including other rules. For example, in Fig. 6.1 the rule $limitInterval_{note}$ calls the interface function *getPitch* and the rule $limitInterval_{pitches}$.

In the Strasheela model, rules are applied to a highly expressive music representation. This representation supports access to score contexts in a generic way (Sec. 5.6). For example, Fig. 6.1 shows how the successor of a note or the note's pitch can be accessed within a rule definition. The full Strasheela music representation interface is available in a rule definition and thus complex score contexts can be accessed and constrained in a convenient way.

In Strasheela, functions are first-class values (Sec. 2.8, [Abelson et al., 1985]) and so constraints and rules are also first-class values. First-class functions allow for higher-order functions which expect other functions as arguments. This idea forms the foundation for Strasheela's programmable rule application.

6.2. Programmable Rule Application

The concept of a function clearly separates its definition and application. Accordingly, the Strasheela rule formalism fully decouples the definition and the application of a compositional rule (or constraint). The user defines a rule and then explicitly applies this rule to the score. This approach is in clear contrast to previous systems (e.g. PWConstraints or Situation, see , Sec. 3.3.1 and Sec. 3.3.2) in which the definition and the application of a rule was amalgamated in a single operation.

In Strasheela, rule application mechanisms are user programmable due to Strasheela's functional programming foundation. A rule application mechanism is a higher-order function which expects a rule (i.e. a function) as argument. A rule application mechanism abstracts an arbitrary control structure which traverses a score to apply a rule to selected score objects and variables. This approach combines a user-controlled rule scope (i.e. the set of variable sets the rule constrains, Sec. 3.3.1.1) with a concise notation to express even complex rule scopes.

---

identifiers start with a lower-case letter (e.g. $limitInterval_p$). However, the variables in the CSP are not always literally represented: often the variables returned by accessor functions (e.g. *getPitch*) are constrained directly as in $limitInterval_{pitches}(getPitch(note_1), getPitch(note_2))$.

In the remainder of this subsection, the programmable rule application mechanism of Strasheela is illustrated by a number of examples. These examples are shown with the intention of demonstrating the expressiveness and generality resulting from this approach. Most examples only show the application of a rule application mechanism to demonstrate its effect. Nevertheless, all these mechanisms can be defined by the user, which is demonstrated in Sec. 6.2.4. To be more easily comparable, the first examples are highly similar and apply either $limitInterval_{pitches}$ or $limitInterval_{note}$ (Fig. 6.1) to pitches of neighbouring notes in a voice.

6.2.1. Direct Rule Application to a Single Object

In the most simple case, a rule is applied directly to a single score object. In Fig. 6.2, the rule $limitInterval_{note}$ (see Fig. 6.1) is directly applied to the first note in *myVoice*. In effect, the interval between this note and its successor is constrained not to exceed a fifth.

$$\begin{array}{ll} \textbf{let} & myNote \stackrel{def}{=} first(getNotes(myVoice)) \\ \textbf{in} & limitInterval_{note}(myNote) \end{array}$$

Figure 6.2.: Direct application of a rule to a single note

The whole example returns the boolean variable which is returned by the rule (see Sec. 6.1). This text adopts the convention that a boolean variable returned by a full example is implicitly constrained to be *true*.

In most cases, a rule should be more general and not hold for only a single pair of pitches. Any control structure can be used to apply a rule to multiple sets of score objects. For example, the rule $limitInterval_{note}$ can be applied directly to all notes in a voice which feature a successor note by iterating through all these notes in a loop.

6.2.2. Applying a Rule to Every Element in a List

A programming technique very similar to iteration is mapping. The higher-order function *map* applies a function to every element in a list, collects the results, and returns a list with all results. In Fig. 6.3, *map* applies the unary rule $limitInterval_{note}$ to all but the last note in *myVoice* and so constrains the interval between all pitch pairs in this voice. Within the rule definition, the successor note is accessed and the interval is constrained (see Fig. 6.1). The last note of *myVoice* is omitted (using the function *butLast*) because it has no successor and thus the application of $limitInterval_{note}$ will fail.

The function *map* returns a list of boolean variables and the operator $\bigwedge$ returns the conjunction of all these values. Again, the boolean returned by the example is implicitly constrained to be true, thus all $limitInterval_{note}$ applications in the example are constrained to return true.

$$\begin{array}{l}\textbf{let}\ \ myNotes \stackrel{def}{=} butLast(getNotes(myVoice)) \\ \textbf{in}\ \ \bigwedge map(myNotes, limitInterval_{note})\end{array}$$

Figure 6.3.: Constrain the interval between all pitch pairs of *myVoice* (using the unary function $limitInterval_{note}$)

6.2.3. Applying a Rule to Neighbours in a List

The previous example (Fig. 6.3) may be somewhat difficult to comprehend: the score context constrained by the rule is partly given as an argument to the rule and partly accessed within the rule definition. Because the succeeding note is accessed within the rule, the list of objects to which the rule is applied must be explicitly reduced with the function *butLast*.

The meaning of this example can be expressed more clearly using the higher-order function *map2Neighbours*. This function applies a binary function to each pair of neighbouring elements in a list. In Fig. 6.4, *map2Neighbours* applies the rule $limitInterval_{pitches}$ (Fig. 6.1) to each neighbouring pair of note pitches in *myVoice*.

$$\begin{array}{l}\textbf{let}\ \ myPitches \stackrel{def}{=} map(getNotes(myVoice), getPitch) \\ \textbf{in}\ \ \bigwedge map2Neighbours(myPitches, limitInterval_{pitches})\end{array}$$

Figure 6.4.: Constrain the interval between all pitch pairs of *myVoice* (using the binary function $limitInterval_{pitches}$)

6.2.4. User-Defined Rule Application Mechanisms

The examples shown so far (Fig. 6.3 and Fig. 6.4) demonstrate how higher-order functions are used as rule application mechanisms, but these mechanisms can also be defined by the user. This fact marks an important distinction between Strasheela and existing systems.

The higher-order function *map* (applied in Fig. 6.3) is a standard means in functional programming and its definition can be studied in many textbooks on functional programming languages.[3] Figure 6.5 defines the higher-order function *map2Neighbours* which was used in the example above (Fig. 6.4). The definition is very brief and consists of only a call to the function *zip* which is a relative of the function *map*.

The function *zip* itself expects two lists *xs* and *ys* and a binary function *fn* and collects the results of all calls $fn(x_i, y_i)$, where x_i and y_i are the i-th element of *xs* and *ys*. For instance $zip([1,6,3],[4,5,6], max)$ returns $[4,6,6]$. The function *zip* is defined in the appendix in Fig. B.2.

[3] Abelson et al. [1985] define *map* in the Lisp dialect Scheme and [van Roy and Haridi, 2004] define it in Oz.

$$map2Neighbours(xs, fn) \stackrel{def}{=} zip(butLast(xs), tail(xs), fn)$$

Figure 6.5.: Rule application functions can be defined by the user: the definition of the higher-order function *map2Neighbours*

The function *map2Neighbours* calls *zip* with three arguments, namely two sublists of its list argument *xs* (one containing all but the last element and the other all but the first element of *xs*) plus the binary function argument *fn*.

6.2.5. Index-Based Rule Application

Rule application mechanisms provide a concise and convenient means to express the scope of a rule. Different systems introduce different rule application mechanisms. For example, Situation (Sec. 3.3.2) offers a set of index-based rule application mechanisms whereas PWConstraints provides a pattern-matching based mechanism (Sec. 3.3.1).

In contrast, this thesis proposes a rule formalism where rules are first-class functions and rule application mechanisms are higher-order functions. In this text it is argued that a rule formalism based on first-class functions is more generic than the rule formalisms of existing systems. To substantiate this claim, this section reproduces the rule application mechanisms of Situation and Sec. 6.2.6 the application mechanism of PWConstraints as higher-order functions.

Situation offers a set of index-based rule application mechanisms. Situation introduces a proprietary language (which cannot be extended by the user) to select and control a mechanism for a rule. The present section describes three fundamental index-based rule application mechanisms. These mechanisms are reproduced in Strasheela.

The first Situation mechanism applies a unary rule to every object in an object sequence whose position (or *index*) in this sequence is specified. The Strasheela function *mapIndex* reproduces this behaviour. This function expects three arguments: a list *xs*, an indices declaration *decl*, and a unary function f. The list *xs* consists of arbitrary values (e.g. score objects). The declaration *decl* is notated as a list containing a mix of single integers and integer ranges. An integer range is notated here with the # operator as follows.

$$startIndex\#endIndex \text{ denotes } startIndex, \ldots, endIndex$$

In the example in Fig. 6.6, *mapIndex* applies the rule $limitInterval_{note}$ to the first three and the fifth note of *myVoice*. The index specification is the list $[1\#3, 5]$ which denotes the indices $[1, 2, 3, 5]$. The function *mapIndex* is defined in Fig. B.3.

The second rule application mechanism applies an n-ary rule to subsequences of objects in an object sequence. This mechanism can be reproduced by generalising the function *mapIndex* to *mapIndices*. The function *mapIndices* expects similar arguments to

let $indices \stackrel{def}{=} [1\#3, 5]$
in $\bigwedge mapIndex(getNotes(myVoice), indices, limitInterval_{note})$

Figure 6.6.: Reproduced Situation rule application mechanism: apply the rule $limitInterval_{note}$ to all objects which are identified by their numeric index

mapIndex (i.e. a list of arbitrary values *xs*, an indices specification *spec*, and a function *fn*), but the indices specification uses a different format. Whereas *mapIndex* expects a flat list of specifications, *mapIndices* expects a list of specification lists. Each specification sublist for *mapIndices* has the format of the specification expected by *mapIndex*. For *mapIndices*, the function *fn* expects a list and is applied to the sublist of elements of *xs* which match a single sublist specification in *spec*. For example, the following expression

$$mapIndices([a, b, c, d, e, f], [[1\#3], [2, 5]], fn)$$

results in these two calls of *fn*

$$[fn([a, b, c]), fn([b, e])]$$

The function *mapIndices* can be defined analogously to *mapIndex* (cf. Fig. B.3).

Finally, the third rule application mechanism applies an n-ary rule to every sublist of n neighbours in a list. This Situation mechanism can be reproduced by generalising the function *map2Neighbours* (Fig. 6.5). Whereas *map2Neighbours* applies a binary rule to every pair of two neighbouring list elements, the generalised function *mapNeighbours* applies an n-ary rule to every sublist of n neighbours in a list (the value n is obtained by accessing the arity of the rule passed to *mapNeighbours*). The actual definition of *mapNeighbours* is not shown for brevity.

The Situation rule application mechanism reproductions presented here are even more general then the original. The original mechanisms are restricted to the sequential score topology of Situation. Instead, the mechanisms of this section can be used on any score context which can be represented as a sequence. For example, Fig. 6.6 applies a rule to the notes of a specific voice (which might be only one of several voices from a polyphonic score). The score context 'notes of voice *myVoice*' has no equivalent in Situation.

6.2.6. Rule Application with a Pattern Matching Language

This section describes and reproduces the rule application mechanism of PWConstraints (the reproduction was motivated in Sec. 6.2.5). In PWConstraints, a rule is applied to all object sets which match the pattern matching expression of the rule. PWConstraints' pattern matching language was introduced in Sec. 3.3.1.1.

The present section defines a very similar pattern matching language in order to reproduce this rule application mechanism. This language also defines three symbols. The two place-holder symbols ? and $*$ retain their meaning: the symbol ? exactly matches one sequence element, and $*$ matches zero or more elements.

The pattern matching language introduced here features no pattern-matching variables. Instead, this language provides the symbol x: every occurrence of a pattern-matching variable in PWConstraints is substituted by the unvarying symbol x here. The following example shows an expression which matches any but the first pair of two successive elements (cf. the PWConstraints example with the same meaning in Fig. 3.7).

$$[?, *,\ x,\ x]$$

The rule application mechanism of PWConstraints is reproduced by the higher-order function *mapPM*. This function expects three arguments: a list *xs* (of arbitrary values, e.g., score objects), a pattern matching expression *pattern* (using the syntax introduced above), and a unary function f which expects a list. The function *mapPM* applies f to every sublist of *xs* which matches the *pattern*. For example, the following expression

$$\begin{array}{ll}\textbf{let} & pattern \stackrel{def}{=} [x, ?, *,\ x,] \\ \textbf{in} & mapPM([a, b, c, d], pattern, f)\end{array}$$

results in two function calls

$$[f([a, c]), f([a, d])]$$

Because the symbol $*$ matches zero or more elements, the *pattern* matches multiple sublists of *xs* in case an instance of this symbol occurs in the *pattern* (multiple occurrences of $*$ are possible). Also, the order of elements which match some instance of the symbol x corresponds with their order in *xs*.

Figure 6.7 uses *mapPM* to apply a melodic rule to the pitches of the notes in *myVoice*. The example is equivalent to the PMC example presented in Fig. 3.8. The function *mapPM* is defined in Fig. B.4.

The function *mapPM* can be used on any sequential data, like the index-based rule application mechanisms introduced in Sec. 6.2.5. For example, *mapPM* can apply rules to any score context of a highly nested music representation which is represented by a list.

6.2.7. Rule Application to Selected Objects in a Score Hierarchy

All rule application mechanisms introduced so far apply a rule to specific elements (or element sets) in a sequence. These mechanisms were implemented by higher-order functions which traverse a sequence in order to apply a rule. Higher-order functions can define

$$
\begin{array}{l}
\textbf{let}\ \ myPattern \stackrel{def}{=} [*, x, x] \\
\textbf{in}\ \ \bigwedge\ mapPM(mapItems(myVoice, getPitch) \\
\qquad\qquad\qquad myPattern, \\
\qquad\qquad\qquad f : f([Pitch_{predecessor}, Pitch_{successor}]) \stackrel{def}{=} \\
\qquad\qquad\qquad\quad \textbf{let}\ \ Interval \\
\qquad\qquad\qquad\quad \textbf{in}\ \ \ Interval = |Pitch_{predecessor} - Pitch_{successor}| \\
\qquad\qquad\qquad\qquad\quad \wedge\ Interval \in \{0, \ldots, 7\})
\end{array}
$$

Figure 6.7.: Reproduced PWConstraints rule application mechanism: constrain the interval between two consecutive pitches of notes in *myVoice* not to exceed a fifth (cf. Fig. 3.8)

control structures which traverse arbitrary data structures. The present section proposes a rule application mechanism which applies a rule to all score objects in a hierarchical music representation which meet a certain condition.

Section 5.5.2.2 introduced the higher-order interface for a hierarchically nested score. That section also introduced a generalised function *map* which traverses a score hierarchy (instead of a flat list, as the basic version of *map*). This variant (method) of *map* expects an optional argument *test* to select the score objects for processing.

Functions like this *map* variant can be used to traverse a score hierarchy and apply a rule to specific elements in the score. The example in Fig. 6.8 constrains all note objects in *myScore* to diatonic pitches (i.e. pitches from a specific scale such as the *C*-major scale). In this example, each constrained note can be arbitrarily nested within *myScore*.

$$\bigwedge\ map(myScore, hasDiatonicPitch, test\colon isNote)$$

Figure 6.8.: Every note in *myScore* is constrained to a diatonic pitch

The rule *hasDiatonicPitch* is defined in Fig. 6.9. The principal part of this function is expressed by the function *isDiatonicPitch*, which constrains the pitch class of its argument *MyPitch* so that it fits into a diatonic scale. For simplicity, *isDiatonicPitch* only constrains whether *MyPitch* is a member of the pitch classes of the *C*-major scale. A more general definition would also allow for transpositions, for example, for the pitches of the *E*♭-major scale. Finally, the function *hasDiatonicPitch* only applies the constraint *isDiatonicPitch* to the pitch of a given note.

Such a mapping-based approach allows the user to apply a rule to arbitrary score objects. The first example applied a rule to events (i.e. notes). The next example uses this template to apply a rule to containers.

Figure 6.10 applies the rule *hasUniqueMelodicPeak* to every voice in *myScore*, namely to every sequential container which is marked with the info-tag *voice*. The rule *hasUnique-*

$$hasDiatonicPitch(myNote) \stackrel{def}{=} isDiatonicPitch(getPitch(myNote))$$

let /* Set of the C-major scale pitch classes */
  $\mathbf{ScalePCs} \stackrel{def}{=} \{0, 2, 4, 5, 7, 9, 11\}$
in $isDiatonicPitch(MyPitch) \stackrel{def}{=} (MyPitch \bmod 12) \in \mathbf{ScalePCs}$

Figure 6.9.: The function *hasDiatonicPitch* constrains whether the pitch of a note fits into the C-major scale and the function *isDiatonicPitch* performs this constraint for a pitch variable

MelodicPeak (Fig. 6.11) constrains that the melodic peak (i.e. the maximum pitch) occurs exactly once in the list of all note pitches in *myContainer*.

$$\bigwedge map(myScore, hasUniqueMelodicPeak, test{:}\, f : f(x) \stackrel{def}{=} isSequential(x) \wedge hasThisInfo(x, voice))$$

Figure 6.10.: The rule *hasUniqueMelodicPeak* is applied to every sequential container marked as voice

$hasUniqueMelodicPeak(myContainer) \stackrel{def}{=}$
  let $pitches \stackrel{def}{=} map(myContainer, getPitch, test{:}\, isNote)$
    $MaxPitch$
  in $MaxPitch = max(pitches)$
    /* The constraint $once$ demands that $MaxPitch$ occurs only once in $pitches$ */
    $\wedge\, once(MaxPitch, pitches)$

Figure 6.11.: In the list of all note pitches in *myContainer*, the maximum pitch occurs exactly once

6.2.8. Implicit Rule Application

In most cases, a rule is *explicitly* applied by the user in the CSP definition. However, rules can also be applied *implicitly* when the music representation for the CSP is created. For example, the temporal constraints between temporal items (such as events and temporal containers, Sec. 5.4.4.1) are applied implicitly (i.e. by the initialisation method of these classes). Used carefully, implicit constraint applications make the definition of CSPs more convenient for the user. For instance, by using temporal items the user can rely on well-formed temporal relationships between these objects.

6.2.9. Rule Application to Arbitrary Score Contexts

This subsection demonstrates how rules are applied to arbitrary score contexts. However, there is one requirement to apply the technique shown in this section (and the techniques shown in the previous sections in this chapter): it must be possible to access a score context in order to constrain this context. In other words, the music representation must provide enough information to isolate this context. This restriction will be addressed in Sec. 6.3.

Figure 6.12 applies a harmonic rule to a polyphonic score. The rule constrains simultaneous notes to be consonant. The context of simultaneous notes is not explicitly represented in the score. Items which are simultaneous to some given item are easily accessed procedurally as was shown by the definition of *getSimultaneousItems* (Fig. 5.31) – as long as the score provides enough information to isolate this context (e.g. the rhythmical structure of the music is fully determined). Whereas *getSimultaneousItems* returns arbitrary items (e.g. including containers), simultaneous notes are accessed similarly (e.g. the output of *getSimultaneousItems* is filtered, cf. Fig. 2.15).

$$
\begin{array}{l}
\textbf{let}\ \ bassNotes \stackrel{def}{=} getNotes(getBass(myScore)) \\
\textbf{in}\ \ \bigwedge map(bassNotes, \\
\qquad harmonicRule : harmonicRule(bassNote) \stackrel{def}{=} \\
\qquad\quad \textbf{let}\ \ simNotes \stackrel{def}{=} getSimultaneousNotes(bassNote) \\
\qquad\quad \textbf{in}\ \ \bigwedge mapPairwise(simNotes, \\
\qquad\qquad\qquad\qquad g : g(note_1, note_2) \stackrel{def}{=} \\
\qquad\qquad\qquad\qquad\quad \textbf{if}\ \ getVoice(note_1) \neq getVoice(note_2) \\
\qquad\qquad\qquad\qquad\quad \textbf{then}\ \ isConsonant_{lessStrict}(note_1, note_2) \\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ true) \\
\qquad\quad \wedge\, map(simNotes,\ h : h(simNote) \stackrel{def}{=} \\
\qquad\qquad\qquad\qquad isConsonant_{strict}(bassNote, simNote)))
\end{array}
$$

Figure 6.12.: Apply harmonic rules which constrain simultaneous notes to form consonant intervals. The rule between an upper-voice note and a bass note is more strict than between two upper-voice notes

In Fig. 6.12 (second line), the function *map* processes all bass notes in *myScore*. For any bass note, the pairwise intervals between all its simultaneous notes (from higher voices) as well as the interval between the bass note and any higher note are constrained. For these two cases (interval between two higher notes vs. interval between a bass note and a higher note), slightly different rules are applied according to common practice [Jeppesen, 1930]. The less strict rule $isConsonant_{lessStrict}$ allows for perfect and imperfect consonances including the fourth. This interval is prohibited in the strict rule version to avoid $^{6}_{4}$-chord inversions.

Both rules are again applied by a higher-order function (*mapPairwise* and *map*). The function *mapPairwise* applies a binary function to all pairwise combinations of a given list (here the list of all simultaneous notes). In function g, the constraint $isConsonant_{lessStrict}$ is only enforced for note pairs where the notes belong to different voices. This precondition assures that the rule application even works for non-homophonic music.

The rule $isConsonant_{lessStrict}$ (Fig. 6.13) defines that two notes are consonant, in case the absolute distance between the pitches of these notes (expressed in key-numbers) is either a prime, a minor third, a major third, or a fourth, etc.

$$
\begin{aligned}
&isConsonant_{lessStrict}(note_1, note_2) \stackrel{def}{=} \\
&\quad \textbf{let } Interval \\
&\quad \textbf{in } \quad Interval \in \{0, 3, 4, 5, 7, 8, 9, 12, \dots\} \\
&\qquad \wedge\, Interval = |getPitch(note_1) - getPitch(note_2)|
\end{aligned}
$$

Figure 6.13.: Constrains the pitches of a note pair to form a consonant interval (less strict version which allows for prime and fourth)

The definition for $isConsonant_{strict}$ is very similar to $isConsonant_{lessStrict}$, only the intervals prime and fourth were removed from the set of allowed intervals.

$$
\begin{aligned}
&isConsonant_{strict}(note_1, note_2) \stackrel{def}{=} \\
&\quad \textbf{let } Interval \\
&\quad \textbf{in } \quad Interval \in \{3, 4, 7, 8, 9, 12, \dots\} \\
&\qquad \wedge\, Interval = |getPitch(note_1) - getPitch(note_2)|
\end{aligned}
$$

Figure 6.14.: Constrains the pitches of a note pair to form a consonant interval (strict version which does not allow for prime and fourth)

This example accessed and constrained simultaneous notes, but any score context can be accessed and constrained in this manner – as long as the music representation provides enough information to isolate this context.

6.3. Constraining Inaccessible Score Contexts

6.3.1. The Inaccessible Score Context Problem

The Strasheela rule application techniques introduced in the preceding Sec. 6.2 allow for freely constraining the variables in any score context. For example, Sec. 6.2.3 demonstrated how melodic rules are applied to neighbouring notes in a voice and Sec. 6.2.9 showed how harmonic rules are applied to simultaneous notes. A Strasheela CSP can utilise any information available in the score to access score contexts.

However, the only score contexts that have been constrained so far are those which are accessible in the CSP definition. The music representation must contain sufficient information in order to allow access to the required score context. For instance, neighbouring notes in a voice can be accessed when the position of these notes in the voice is known. Similarly, sets of simultaneous notes can be accessed, in case the temporal structure of the music is already determined in the problem definition.

There are cases which require constraining score contexts which are inaccessible in the problem definition, because sufficient information is missing in the definition. These contexts only become known during the search process. An example is a melodic rule which constrains the intervals between neighbouring local pitch maxima (i.e. melodic peaks) in voice. Such local maxima stand out and are treated with special care in many musical styles (e.g. only a small interval might be permitted between melodic peaks). With the techniques introduced so far, a rule concerning these maxima can only be applied in cases where it is already known in the problem definition which notes will become the melodic peaks in the solution.

A second example which exhibits the restriction caused by missing information is a harmonic rule which constrains the intervals between simultaneous note pitches. In the case where the rhythmic structure is undetermined in the problem definition, it is also not known which notes occur simultaneously and it is thus not known which note sets the rule should be applied to.

This *inaccessible score context problem* is in some form shared by all music constraint systems. The only exceptions are systems which perform a naïve generate-and-test search: the *generate-and-test* algorithm first generates a fully determined solution candidate and then tests whether this candidate fulfils all constraints. This approach results in a vastly inefficient search process. Other systems – and in particular systems which perform constraint propagation to increase efficiency such as Strasheela – have to deal with missing or partial information in the problem definition.

6.3.2. Delayed Rule Application

The 'inaccessible score context problem' can be addressed in a number of ways. A first approach is to delay the constraint application. A *delayed rule application* waits until enough information is available before anything is done. For instance, a delayed rule application may wait until some context – which was inaccessible in the problem definition – becomes accessible, and then it applies its constraint to it. For example, a harmonic rule may wait until the search process decided on the rhythmical structure and the context 'simultaneous notes' can be accessed. Then the harmonic rule applies its constraints to variables (e.g. note pitches) in this context.

This approach shows similarity to the way a composer works. A composer might also delay certain decisions until enough information is available. For example, when creating a rhythmically complex texture a composer might first decide on the rhythmical structure before writing the actual note pitches and considering the voice leading (which depend

on information about the rhythmical structure). Therefore, delaying decisions appears a natural solution to the inaccessible score context problem.

However, delaying the application of rules requires the introduction of new concepts to the underlying constraint formalism which make use of the notion of time (i.e. the time spent searching). For example, a delayed rule application could run concurrently in its own *thread* and could be applied to a *dataflow variable* [van Roy and Haridi, 2004]. Initially, the value of this variable is undetermined, but it becomes bound to the respective score context once enough information is available. The rule application mechanism waits until this happens. It then applies the rule to this score context. The notation of such CSPs would require new constructs, for example a notation for concurrent processes (as in the π-calculus [Parrow, 2001] or the Oz notation [van Roy and Haridi, 2004]) Moreover, such CSPs introduce a procedural element (the delaying of the rule application) to the otherwise declarative problem statement. This procedural element can make the problem harder to understand, because the CSP effectively changes during the search process.

6.3.3. Reformulating the Problem Definition

A second approach addresses the 'inaccessible score context problem' simply by reformulating the CSP such that any undetermined context is avoided. For example, a typical problematic case is a polyphonic CSP which constrains both the rhythmic structure and the pitch structure and which applies rules to simultaneous notes as well as to notes which are neighbours in a voice. In fact, polyphonic CSPs which allow users to constrain the rhythmical structure as well as the pitch structure have proven to be so problematic that very few existing systems support such problems.

All polyphonic Strasheela score examples shown so far applied a tree-like score topology established by nesting *simultaneous* and *sequential* containers (Sec. 5.4.4). In this representation, the information required to access neighbouring notes in a voice is provided. On the other hand, the information required to access simultaneous notes is missing in cases where the rhythmical structure is not determined.

An alternative CSP definition could instead apply a lattice-like score topology similar to the one used by Humdrum which was discussed in Sec. 2.4.1. A container of a lattice-like topology represents its contained elements in a two-dimensional array in contrast to the one-dimensional item sequence of the Strasheela containers. In the two-dimensional topology, one dimension represents simultaneous score objects of the same duration in time slices (corresponding to the Humdrum records) and the other dimension represents a temporal sequence of score objects (corresponding to the Humdrum spines). Homophonic musical CSPs – in which all voices share the same rhythmic structure, but which can still feature different rhythmic values – can be expressed very well with such a container for a lattice-like topology. Even if the rhythmic structure of the score is undetermined in the problem definition, such a topology always provides the information required to access simultaneous notes as well as neighbouring notes.

This approach can even be generalised for non-homophonic CSPs. To this end, the music representation introduces an additional parameter for each score object in the lattice-container whose value is a boolean variable. This parameter indicates whether its score object is tied to its predecessor in the voice. A score object tied to its predecessor serves a similar purpose as the null token in the Humdrum representation.

Additional constraints are applied to an object with a tie-parameter. For example, the pitch of a note tied to its predecessor note is the same as the pitch of this predecessor note. The application of these additional constraints again may require the user to access a context which is undetermined in the CSP definition: in the problem definition, it is not necessarily known which notes are tied and which are not: the variable of the tie parameter may be undetermined. Therefore, reformulating the CSP is not always sufficient to avoid inaccessible score contexts. Another approach is required to address this problem in a generic way.

6.3.4. Using Logical Connectives

The 'inaccessible score context problem' can also be addressed by introducing logical connectives such as disjunction, implication and equivalence as constraints. These logical connectives constrain the validity of logical expressions and can thus constrain other constraints. For example, *implication* can express that a constraint must hold in cases where certain conditions are met. The information required to decide either way can be missing from the problem definition (i.e. this information becomes available only during the search process). The expressive power is thus similar to delayed rule application. Still, first-order logic (the formalism which defines logical connectives, [Kelly, 1997]) constitutes a well-established foundation to define complex systems.

Section 6.1 defined constraints and rules as functions which always return a boolean variable. This variable reflects the validity of the constraint/rule. The rule formalism was designed with logical connectives in mind. So far, the returned boolean variables were only used to explicitly connect multiple rule applications by conjunctions and constraints/rules were always constrained to hold.

The following example demonstrates how the 'inaccessible score context problem' is addressed by logical implication. This example consists of a polyphonic CSP which constrains both the rhythmic structure and the pitch structure. If the rhythmical structure is undetermined, the context 'simultaneous objects' is inaccessible (depending on the score topology used in the problem as was discussed above).

Figure 6.15 defines the harmonic rule *simultaneousNotesAreConsonant* which constrains the interval between two notes to be consonant in case these notes are simultaneous. Instead of directly accessing the context of simultaneous notes, this context is expressed by a rule. The implication expresses that if the rule *isSimultaneous* (defined in Fig. 5.32) is fulfilled (i.e. returns 1) then the rule *isConsonant* (defined in Fig. 6.14) must hold as well.

$$simultaneousNotesAreConsonant(note_1, note_2) \stackrel{def}{=}$$
$$isSimultaneous(note_1, note_2) \Rightarrow isConsonant(note_1, note_2)$$

Figure 6.15.: A harmonic rule which makes use of an implication constraint: $note_1$ and $note_2$ being simultaneous implies that their pitches form a consonant interval

Instead of directly accessing the score context of simultaneous notes to apply a rule, the rule *simultaneousNotesAreConsonant* is applied to all object sets which *potentially* form this score context, that is, all note pairs which are possibly simultaneous in a solution. In an extreme case (where note duration domains contain extremely different values) any note of one voice can occur simultaneously with any note of another voice in a solution. To guard against this extreme case, it is safer to apply the rule to any possible combination of two notes from different voices, that is, to each pair in the *Cartesian product* of all notes of a first voice and all notes of a second voice.

As before, a rule which is applied to multiple score object sets can be applied using a suitable higher-order function. The higher-order function *mapCartesianProduct* expects two lists *xs* and *ys* and a binary function *fn* and applies this function *fn* to all possible combinations of *xs* and *ys*. For example, the call

$$mapCartesianProduct(xs, ys, fn)$$

results in

$$[fn(xs_1, ys_1), fn(xs_1, ys_2), \ldots, fn(xs_1, ys_n), fn(xs_2, ys_1), \ldots, fn(xs_n, ys_n)]$$

Using *mapCartesianProduct*, the rule *simultaneousNotesAreConsonant* is applied conveniently to all note pairs which are potentially simultaneous in a solution (Fig. 6.16).

$$\bigwedge mapCartesianProduct(getNotes(voice_1),$$
$$getNotes(voice_2),$$
$$simultaneousNotesAreConsonant)$$

Figure 6.16.: Application of *simultaneousNotesAreConsonant* to all potentially simultaneous note pairs

6.3.5. Comparison

Comparing the different approaches used to address the inaccessible score context problem is difficult, because these approaches are very different in nature.

It is always a good solution to reformulate the CSP definition such that inaccessible score contexts are avoided. However, this approach might not be generic enough: some musical CSPs cannot be expressed that way. For example, the lattice-like score topology is suited to solving homophonic CSPs where both the rhythmic as well as the pitch structure is undetermined in the definition. However, non-homophonic CSPs again lead to inaccessible score contexts: it may be unknown in the problem definition whether a note is tied to its predecessor note.

The approach based on delayed rule application, as well as the one based on logical connectives (e.g. the implication constraint), are both more generic. For example, both approaches allow for polyphonic CSPs with rhythmically independent voices where both the rhythmic as well as the pitch structure is undetermined. A logical connective is more declarative whereas a delayed application is more procedural. Logical connectives constitute a well-established formalism with a clear semantics. CSPs using delayed application, on the other hand, can be harder to comprehend because of their procedural nature.

Furthermore, a logical connective such as implication is more expressive than a delayed application because it works 'both ways' whereas delayed rule application only works 'one way'. For example, the implication in the rule *simultaneousNotesAreConsonant* affects both its arguments. If the two notes are simultaneous, then it follows that they must also be consonant. At the same time, if the two notes are not consonant, then it follows that they must not be simultaneous.

A delayed rule, on the other hand, only reacts to the availability of a score context. If during the search process two notes turn out to form a dissonant interval, then the delayed rule application cannot propagate this information (i.e. it does not deduce that these notes cannot be simultaneous; constraint propagation was informally introduced in Sec. 4.3.1 and will be explained in more detail in Sec. 7.2.1).

For constraining inaccessible score contexts, logical connectives are preferred in this research to the other approaches because of their clear semantics and their generality. Nevertheless, it should be noted that Strasheela also supports delayed rule applications because Strasheela's approach to search is based on concurrent constraint programming (see Sec. 7.1).

7. How Strasheela Finds a Solution

This chapter explains how Strasheela finds a solution for a musical constraint satisfaction problem.

Chapter Overview

The first two sections of this chapter provide requisite background information. Section 7.1 very briefly outlines a declarative concurrent programming model which forms the foundation for the constraint model introduced in Sec. 7.2. Section 7.3 explains how this constraint model is customised to solve musical CSPs.

7.1. The Underlying Programming Model

The previous Chap. 5 and Chap. 6 explained the declarative semantics of Strasheela in terms of three essential programming paradigms: constraint programming, object-oriented programming, and functional programming. Constraint programming forms the core of any music constraint system. Object-oriented programming is the foundation of Strasheela's music representation. Finally, Strasheela's rule application mechanisms utilises functional programming techniques.

This multi-paradigm computational model which subsumes these three paradigms was chosen in the chapters above to explain *what* Strasheela does, because these programming paradigms are well-known in the computer music community. Furthermore, a CSP based on constraint programming plus functional programming can be conveniently notated by a familiar mathematical notation. However, Strasheela's actual foundation is a different computational model, namely a model which has the following fundamental features [Schulte, 2002]:

Partial Values: *Logic variables* provide partial values: a variable can be free (nothing is known about its value), partially determined (e.g. it is known that it is a list without knowing all list elements) or fully determined. Constraints such as unification add information about variable values.

Concurrency: Computations can be executed in multiple concurrent *threads* (threads are created explicitly).

Synchronisation of Threads on Variables: The computation in a thread *blocks* if a logic variable used in a statement of the thread lacks required information. Another

thread might provide this information. That way, threads communicate with each other via variables (dataflow variables). This form of concurrency is stateless and thus declarative (no conflicts of shared resources can occur).

First-Class Procedures: Computations can be abstracted in procedures which are first-class values and support lexical scoping.

This basic model introduces some notion of variables and constraints. However, this model still lacks support for searching. Section 7.2 extends this model to a constraint model which performs a complete search.[1] The basic model itself is explained in more detail in [Schulte, 2002].[2]

The functional programming and object-oriented programming capabilities of Strasheela are based on this basic model. First-class functions are a special case of first-class procedures: a function is a procedure which returns a single value. The object-oriented programming model can be defined in terms of procedures (see [Abelson et al., 1985; van Roy and Haridi, 2004]).

7.2. The Constraint Model Based on Computational Spaces

Strasheela is founded on a constraint programming model which introduces the notion of computational spaces [Schulte, 2002], a programming construct for encapsulating speculative computations such as constraint-based computations. This model makes the search process programmable at a high-level. Computational spaces are the main extension of this model when compared with the basic model outlined above.

For a new user, two aspects of this constraint model are apparent: (i) the model performs local deductions (constraint propagation) automatically, whereas (ii) speculative decisions during the search process (branching, distribution) are programmed by the user. This observation led to the slogan *propagate-and-search* [van Roy and Haridi, 2004] (also known as *propagate-and-distribute* [Schulte and Smolka, 2004]). This labelling focuses on two important features of this model (propagation and distribution) but masks other features. The following Sec. 7.2.1 will therefore first introduce the propagate-and-search approach (the main ideas of this approach were already introduced in Sec. 4.3.1, Section 7.2.2 explains propagate-and-search by an example). Section 7.2.3 will discuss other essential features of the space-based constraint model.

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

[1] Additionally, this basic model can be extended to support committed choice constraint programming (don't care non-determinism) by introducing a parallel conditional [Schulte, 2002].

[2] Schulte [2002] calls this model *Oz Light*, and this model is basically the same as the *Message-Passing Concurrent Model* in [van Roy and Haridi, 2004].

choice points (alternatives) in the search tree, although the approach supports n-ary choice points.

The space-based constraint programming model is only outlined in the present text. Schulte [2002] describes in detail the notion of computation spaces and the consequences of their introduction. For example, the author discusses the features of a constraint model based on spaces (such as user-defined search strategies), but also presents a comparison of search approaches based on spaces (which perform *copying* and can perform *recomputation*, explained below) and approaches based on *trailing* (which undoes previous decisions if required) – which are used by most constraint systems. The programming textbook by van Roy and Haridi [2004] explains how this model is defined on top of more fundamental programming models such as declarative programming (i.e. functional and logic programming without the notion of search) and concurrent programming. Schulte and Smolka [2004] provide a tutorial on constraint programming with finite domain integers using this model, and Müller [2004] a tutorial on finite set constraints. Duchier et al. [1998] apply this model to natural language processing and explain many aspects of the constraint model as required for the purpose of the book.

7.2.1. Propagate and Search

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

A *computation space* encapsulates the information available at a certain stage during the search process. The information is expressed by (i) a constraint store, and (ii) a number of constraint propagators which are contained in a space (see Fig. 7.1). The *constraint store* stores partial information about the values of variables. A *propagator* represents a constraint on these variables. A computation space also contains a distributor (explained below).[3]

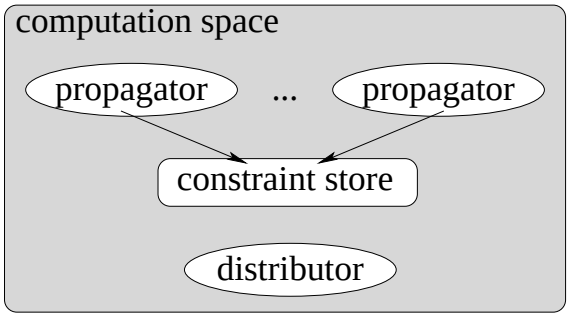


Figure 7.1.: Propagate and search: a computation space encapsulates the information available on a CSP at a certain stage during the search process

[3] A computation space contains further entities which are not discussed in this introduction (e.g. a mutable store, cf. [van Roy and Haridi, 2004]).

A constrained variable is a logic variable about which additional information is known: it has a domain of possible values. The constraint store expresses this domain information as well as which variables are equal. The store has the form of a conjunction of basic constraints. A *basic constraint* is a piece of information which is restricted to very few simple forms.

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

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

All constraints which don't fit into this form of basis constraints are *non-basic constraints*. Most constraints are non-basic constraints. Examples include $X < Y$ or 'all values in $[X_1, \ldots, X_n]$ are distinct'. Non-basic constraints are represented by propagators. A propagator strives to amplify the information in the constraint store by adding new basic constraints to the store which are consistent with the constraint store and follow from the constraint expressed by the propagator itself. For example, for a constraint store

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

the propagator $X < Y$ narrows the variable domains such that the constraint store is updated to

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

Each propagator is a software agent, that is a procedure running in its own thread. Therefore, multiple propagators can reduce the domain of variables in turn. Constraint propagation performs local deduction which reduces the search space (by narrowing the domains of variables) without actually searching (i.e. without performing any decision which may be wrong and requires undoing) and without excluding any solution. A propagator disappears when it is entailed by the constraint store (i.e. all its variables are determined). A space is *solved* when it contains no propagator (usually, all its variables are determined then). A space is *failed* when it contains a failed propagator, that is a propagator which is inconsistent with the constraint store.

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

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

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

In this situation, constraint *distribution* comes to a decision which results in two new computation spaces which are easier to solve (see Fig. 7.2). A distributor is also a software agent which waits until its hosting computation space becomes stable. The distributor then yields two child spaces of the stable parent space. Both child spaces inherit all information available in the parent space (e.g. the constraint store and the propagators). Additionally, the distributor applies an arbitrary constraint C to any variable in one child space and its complement $\neg C$ to the corresponding variable in the other child space. Thus, the distributor executes a nondeterministic choice operation where C and $\neg C$ denote alternatives. For example, the constraint C may determine some variable to a certain value (e.g. to the first value in its domain), in which case $\neg C$ excludes this value from the variable domain.

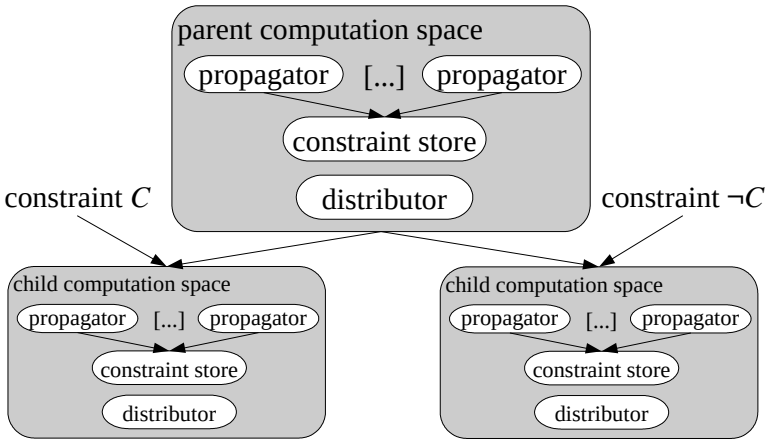


Figure 7.2.: Constraint distribution creates two child spaces which are the result of two complementary decisions (expressed by the two added constraints C and $\neg C$)

Adding a constraint and its complement (i.e. two equivalent alternatives) at the same time does not change the set of solutions. It will, however, often restart propagation.

A *distribution strategy* defines the decision that is performed by the distributor. A common strategy is *first fail*: this strategy determines some variable with the smallest domain to its leftmost domain value (and removes this value from the variable domain in the second child space). The following section 7.2.2 explains this distribution strategy with an example.

The branching caused by constraint distribution results in a search tree. The example in the following section also depicts the corresponding search tree in Fig. 7.4. This search tree is investigated by an exploration strategy for solutions, for example by the depth first search algorithm (the exploration strategy concept is explained later in more detail in Sec. 7.2.3.3).

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

7.2.2. An Example

The following paragraphs illustrate constraint propagation and distribution by computing an all-distance series. An all-distance series is a musical CSP very similar to an all-interval series (Sec. 3.2.2). Both CSPs define a series which consists of unique pitch classes and unique intervals between these pitch classes. The CSPs only differ in the way the pitch classes and the intervals are related. In an all-distance series, the intervals between the pitch classes are absolute distances (instead of inversional equivalent intervals as in an all-interval series). That is a fourth upwards and a fourth downwards are regarded as the same interval (instead of a fourth upwards and a fifth downwards in the case of an all-interval series).

Figure 7.3 shows the definition of the all-distance CSP of length 4 (i.e. the solution series consists of only 4 pitch classes). Figure 7.4 shows the full search tree for all solutions of this CSP. Each tree node represents a computation space. Solved spaces are drawn as diamonds ◊, failed spaces as boxes □ and distributable spaces as circles ○.

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

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

/* Constraints relation between xs and dxs. */

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

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

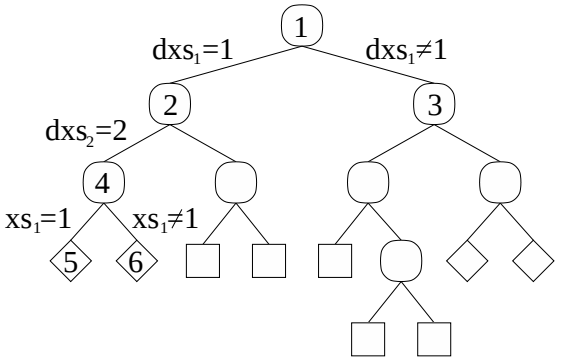
$\wedge\ distinct(xs)$

$\wedge\ distinct(dxs)$

Figure 7.3.: All-distance series definition of length 4

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.

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



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

Figure 7.4.: All-distance series: search tree for all solutions of length 4

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

7.2.3. Features of the Space-Based Constraint Model

The constraint programming model based on the notion of computational spaces has a number of beneficial features. The following paragraphs list those features which are particularly influential for music constraint programming. These features are partly the result of the programmable nature of this constraint programming model and therefore characteristic of this model (in particular recomputation, explained in Sec. 7.2.3.5). Other features do not necessarily depend on spaces and are in some form shared by other popular constraint systems (e.g. SICStus Prolog and ECLiPSe both support reified constraints, explained in Sec. 7.2.3.4). Nevertheless, all these features are listed here because existing music constraint systems often do not support any of these features.[4]

[4]The literature on music constraint programming already introduced the term *constraint reification* in the discussion of the BackTalk system (cf. Roy and Pachet [1997]), but with a clearly different meaning. The Smalltalk programming language – in which BackTalk is implemented – allows a program to reason about itself: Smalltalk is a highly reflective language. A *reified* Smalltalk object can be processed as data. For example, the Smalltalk stack is reified which allows a program to influence the order in which statements on the stack are executed.

7.2.3.1. Constraint Propagation Between Domain Specific Variables

A constrained variable features a finite domain which constitutes a finite set of possible values for this variable. This domain is uniform; all possible values for a variable belong to the same mathematical domain (i.e. constrained variables are quasi 'typed'). Particular common domains include finite domain integers (i.e. natural numbers, often simply called finite domain) [Schulte and Smolka, 2004] and finite sets (of natural numbers) [Müller, 2004]. Further domain examples are real intervals [Díaz et al., 2005], graphs [Dooms et al., 2005b], records [Duchier et al., 2004], and maps (domain of functions) [Deville et al., 2005].[5]

Propagators maintain constraints between variables of specific domains. For example, an addition constraint $X + Y = Z$ is defined for three numbers (e.g. finite domain integers or real intervals). Customarily, constraints are named accordingly (e.g. finite domain constrains or finite set constraints). Many constraints express a relation between variables of the same domain, while some constraints define a relation between variables of different domains. For example, the membership constraint $X \in \mathbf{S}$ often expresses a relation between a finite domain integer and a finite set of integers.

Constraint propagation is important, because it reduces the domains of variables by making local deduction. In this way, propagation significantly reduces the search space without making any decision which is potentially wrong and requires undoing.

Constraint domains allow the combination of natural modelling with efficient solving. Domains are based on mathematical concepts which allow for universal application. Constraints on domain specific variables usually implement highly-optimised domain specific propagation algorithms.

Many existing music constraint systems, on the other hand, support variable domains which consist of arbitrary values. However, these systems do not support constraint propagation which considerably affects their performance.

7.2.3.2. Distribution Strategy

The propagate and search approach allows the user to program how the search process is conducted. Two independent aspects of the search process are controlled separately: how the search tree looks like and how this tree is explored. The former aspect is controlled by the distribution strategy, the latter by the exploration strategy (explained below).

The distribution strategy defines the shape and content of the search tree. For example, it controls how many nodes exist and what constraints are added at each branching of the search tree. The user can define simple distribution strategies 'from scratch' with a few lines of code (cf. [Schulte, 2002, p. 37]). However, a high-level interface can make the

[5] The references given in this paragraph point to Oz-related literature, that is literature of immediate relevance for the Strasheela prototype. Schulte [2002, p. 1] provides more general citations on variable domains.

definition of distribution strategies more convenient. This section presents an interface which is a distinctly simplified version of the Oz procedure `FD.distribute` [Duchier et al., 2004], which is the standard means for solving finite domain integer CSPs in Oz.

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

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

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

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

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

The high-level interface for creating distributors cannot be defined in the notation used so far, because it requires a notation with support for procedures and concurrency.[6]

It depends on the actual CSP as to which distribution is suitable (e.g. efficient) for this problem. A variable ordering which follows the well-proven *first-fail* principle first determines those variables which are particularly hard to solve [Bartak, 1998]. For instance,

[6] The high-level interface described in the present section for defining distribution strategies is defined below in Oz notation [van Roy and Haridi, 2004]. This definition is best understood together with [van Roy and Haridi, 2004, Chap. 12], which explains in a step-by-step manner how a distribution strategy is executed in terms of space operations.

The procedure `MyDistributor` expects as arguments the order specification `Ord` and value specification `Val` as discussed above, together with a list `Xs` containing all variables in the CSP. `MyDistributor` waits until its hosting space is stable. It then filters out already determined variables. In case there are no undetermined variables left (i.e.`Vars` is `nil` and the CSP is solved) the distributor does nothing. Otherwise, the variable to distribute is selected (`Order` is a higher-order function – a relative of `Sort` – which returns the list element fitting best the given ordering function), bound to `Var`, and a domain specification `Dom` is computed for this variable. The binary choice states that the domain of `Var` is either `Dom` or its complement. These two alternative constraints are added to the two child spaces (which are created by the choice statement). The recursive call of `MyDistributor` adapts the distribution for these child spaces to the reduced list of variables (once filtered out, variables will never be considered for distribution again).

Again, the example aims for clarity and not efficiency (the variables are traversed twice: once when filtered and then again when selecting the variable to distribute).

the previously discussed first-fail strategy first distributes some variable with the smallest domain. An alternative approach distributes some variable to which most constraints are applied.

Common decisions for a particular domain value in addition to the domain minimum include the median or the maximum. Alternatively, the domain of a variable can be reduced to its lower half instead of deciding on a specific domain value (this technique is sometimes called domain splitting). A well-proven rule of thumb for value orderings is the *succeed-first* principle which decides on a promising domain value first [Bartak, 1998].

A distribution strategy can define heuristics to place a better solution earlier in the search tree. Such a *best-first* principle can be interpreted as a variation of the succeed-first principle. A typical heuristic for a musical CSP avoids repetition and uniformness. For many musical CSPs it is appropriate to randomise the decision for a domain value. Still, it is important that a distribution strategy is deterministic, that is, if executed multiple times a distribution strategy must make the same decision in each computation space. This is particularly important for large CSPs which require recomputation (Sec. 7.2.3.5). A distribution strategy can be seemingly random but still deterministic when based on pseudo-random values.

Dynamic Variable Ordering

It is important to note that a distribution strategy decides which constraint to add only when it is actually required and thus can make use of all the information available at the current state of the search (dynamic variable and value order). In contrast, most existing music constraint systems apply search strategies in which the order of decisions is determined before the search starts (static order). Section 7.3 will discuss the consequences of static versus dynamic variable orderings for musical CSPs.

```
proc {MyDistributor unit(order:Ord value:Val) Xs}
    {Space.waitStable}
    local
        Vars={Filter Xs fun {$ X} {FD.reflect.size X} > 1 end}
    in
        if Vars \= nil
        then
            Var = {Order Vars Ord}
            Dom = {Val Var}
        in
            choice {FD.int Dom Var} [] {FD.int compl(Dom) Var} end
            {MyDistributor unit(order:Ord value:Val) Vars}
        end
    end
end
```

7.2.3.3. Exploration Strategy

While a distribution strategy defines the shape of the search tree, an exploration strategy traverses this search tree to find solutions. The definitions of a distribution strategy and an exploration strategy are independent of each other. A search engine (which implements an exploration strategy) expects a CSP together with a distributor encapsulated in a *script* (a procedure).

A basic exploration strategy such as *depth-first search* can be used to find a single solution, as well as multiple or all solutions. Depth-first search is particularly adequate for finding a single solution. In this case, the most critical aspect is a suitable distribution strategy.

Several exploration strategies have been proposed which do not require exploring the full search tree for finding a good or an optimal solution according to a user-defined formal criterion. *Limited discrepancy search* (LDS) [Harvey and Ginsberg, 1995] trusts that the heuristics which define the search tree (i.e. the distribution strategy in our case) is appropriate. This heuristic defines a preference for the first alternative of two child spaces. LDS limits the number of discrepancies, that is, how often the path in the search tree takes 'the wrong turn' and follows the second alternative. Schulte [2002] adapts LDS to the space-based constraint model.

Informed strategies exploit information contained in partial solutions to decide where to continue the exploration. *Best-first search* or *A*-search* (Russell and Norvig [2002]) has been adapted for the space-based constraint model [Schulte, 2002; Tack and Botlan, 2005].

Best-solution search (branch-and-bound, BAB) [Schulte, 2002] searches for a series of solutions. Whenever the search finds a solution, the next solution is constrained to be better with respect to a user-defined comparison constraint. This additional comparison constraint performs propagation (i.e. excludes too poor solutions). When no further solution can be found, the last solution is optimal with respect to the comparison constraint.

The exploration strategies above expect a formal criterion for a best solution. For LDS, this criterion is the heuristics which creates the search tree (i.e. the distribution strategy), best-first search expects a cost function, and BAB a comparison constraint.

For some CSPs, however, it is extremely hard to formalise what a good solution constitutes. For example, for a musical CSP it is often impossible to state in a formal way what distinguishes a good solution from a poor one. Still, musicians are easily able to make this distinction.

The formalised musical knowledge expressed in a CSP can be combined with the non-formalised expertise of the user when the user interactively guides the exploration of the search tree. Schulte [1997] proposes the Oz Explorer to conduct a *visual search*. The Explorer features a graphical user interface which shows the search tree and allows for interactive exploration. For example, the user can examine the partial solutions of nodes in the search tree and decide which alternative to explore further. In addition, the

Explorer is an excellent device to analyse the search process (e.g. to improve or debug a distribution strategy).

Exploration strategies can be defined 'from scratch' (cf. [Schulte, 2002]). However, in doing so the user must take many subtleties into account. Therefore, Tack and Botlan [2005] propose high-level abstractions to make the definition of exploration strategies more convenient for the user. These abstractions automatically incorporate well-known optimisations (e.g. last alternative optimisation [Schulte, 2002]) and recomputation (see Sec. 7.2.3.5) and also provide a visual search interface.

An *exploration strategy* is often called a *search strategy* instead. The present text prefers the more explicit term "exploration strategy". This term cannot be confused with the broader notion of the term search strategy which covers all aspects of the search process (i.e. constraint propagation, distribution, as well as the exploration of the search tree).

7.2.3.4. Reified Constraints

Complex CSPs often call for constraining under which circumstances some constraints hold. For example, logical connectives such as disjunction, implication, and negation constrain the validity of other constraints. Section 6.3.4 discussed an example which applies an implication constraint to express that two notes are consonant in cases where they are simultaneous. A similar technique allows the user to solve over-constrained problems where some constraints cannot be satisfied.

A *reified constraint* (sometimes called meta-constraint) is a constraint C which reflects its validity in a boolean variable B. A boolean variable is expressed by a finite domain variable with the domain $\{0, 1\}$ and is often called a 0/1-variable (0 denotes *false* and 1 denotes *true*).

$$(C \Leftrightarrow B) = 1 \wedge B \in \{0, 1\}$$

The discussion of reified constraints clearly shows the difference between the declarative semantics of constraints discussed above (e.g. in Chap. 6) and the operational semantics discussed in the present chapter. In the declarative semantics, constraints are expressed by functions which expect variables as arguments and always return a boolean variable. Thus, all constraints are quasi-reified constraints. These declarative semantics were only introduced to simplify the discussion in this text (e.g. to avoid introducing a notation which expresses concurrent procedures).

In the operational semantics, constraints are expressed by propagators (i.e. concurrent procedures) which influence a set of variables but do not return any value. Such a constraint must always be true. Nevertheless, a propagator may constrain a 0/1-variable which reflects whether the values of the other variables fulfil the constraint expressed by the propagator. A reified constraint is thus a special case which is used when this additional expressive power is required. For example, reified constraints are required to

express a disjunction or an implication between two constraints in Strasheela's operational semantics. On the other hand, a conjunction does not require reified constraints (see below).

Many propagators come in two variants: a reified version and a non-reified version. The constraint model based on computation spaces also supports a generic reification operation for arbitrary constraints which also provides propagation [Schulte, 2002].

Like any variable, a 0/1-variable can be freely constrained. Popular constraints for 0/1-variables are logical connectives (i.e. conjunction, disjunction, exclusive disjunction, negation, implication, and equivalence).

The difference between the declarative semantics expressed by functions and the operational semantics expressed by propagators has an additional effect. Because every constraint returns a boolean variable in the declarative semantics, it is necessary to explicitly express a conjunction of all these constraints. In the operational semantics, a constraint (which is not a reified constraint) must always hold. Consequently, multiple independent constraints must all hold and explicit conjunctions are not required. In effect, less constraints are required in the operational semantics.

Modelling Soft Constraints

Composers often want to formulate a preference rather than a strict rule. For instance, a composer might prefer small melodic intervals but still wishes to allow for larger intervals. There are cases where composers merely want to relax a constraint and other cases where they want an optimal solution.

Soft constraints can be modelled by reified constraints. Besides, reified constraints also allow for solving over-constrained problems where only some of the stated constraints can be satisfied.

A 0/1-variable is a finite domain integer and all integer constraints can be applied to 0/1-variables. For instance, multiple 0/1-variables can be added. In Fig. 7.5, the constraint *howManyTrue* constrains how many 0/1-variables in the list *bs* equal 1 (i.e. *true*) by adding these variables (the examples still use the declarative semantics for convenience). Related examples are the common constraints *exactly*, *atLeast* and *atMost* (exactly/atLeast/atMost N 0/1-variables equal 1).

$$howManyTrue(bs, N) \stackrel{def}{=} N = \sum bs$$

Figure 7.5.: The constraint *howManyTrue* constraints that N (a finite domain integer) variables in *bs* (a list of 0/1-variables) are true (i.e. 1)

The constraint *probability* (Fig. 7.6) is a variant of *howManyTrue* and constrains that the probability for a variable in *bs* (a list of 0/1-variables) to equal 1 lies between the floating point numbers *min* and *max* (a small error is introduced by the rounding operation)

$$
\begin{array}{l}
probability(bs, min, max) \stackrel{def}{=} \\
\quad \textbf{let } l \stackrel{def}{=} length(bs) \\
\qquad minDomain \stackrel{def}{=} round(l \cdot min) \\
\qquad maxDomain \stackrel{def}{=} round(l \cdot max) \\
\quad \textbf{in } howManyTrue(bs, fdInt(\{minDomain, \ldots, maxDomain\}))
\end{array}
$$

Figure 7.6.: The constraint *probability* constrains the probability of a 0/1-variable in *bs* equaling 1

The compositional rule *preferSteps* expresses 'skips in the melody are allowed, but there should be more steps than skips' (Fig. 7.7). This rule can be expressed by *howManyTrue* or *probability*. Although the rule *preferSteps* is not a truly soft rule, it models the effect of a soft rule on each individual interval, by applying a strict rule on the sequence of all intervals.

$$
\begin{array}{l}
preferSteps(intervals) \stackrel{def}{=} \\
\quad probability(map(intervals, f : f(Interval) \stackrel{def}{=} Interval < 3), 0.7, 0.9)
\end{array}
$$

Figure 7.7.: Prefer intervals which are smaller than a minor third: the probability of intervals which are less than 3 is between 0.7 and 0.9

The rule *preferSteps* expects *intervals* as an argument, which is a list of the absolute distances between the pitches in a melody (which are expressed by key-numbers). In the following rule application example, the *intervals* between the *melodyNotes* are created by processing this list of notes with the help of *map2Neighbours* (*map2Neighbours* is introduced in Sec. 6.2.3).

$$
\begin{array}{l}
\textbf{let } melodyNotes \stackrel{def}{=} getItems(myMelody) \\
\quad intervals \stackrel{def}{=} \\
\qquad map2Neighbours(melodyNotes, \\
\qquad\qquad f : f(note_1, note_2) \stackrel{def}{=} \\
\qquad\qquad\quad \textit{/* Create and return the variable } Interval\textit{, which is constrained to the absolute distance between the pitches of two neighbouring notes. */} \\
\qquad\qquad\quad \textbf{let } Interval \\
\qquad\qquad\quad \textbf{in } Interval \textbf{ where} \\
\qquad\qquad\qquad Interval = |getPitch(note_1) - getPitch(note_2)|) \\
\textbf{in } preferSteps(intervals)
\end{array}
$$

The rule *preferSteps* constrains the probability of steps in a melody, but it does not optimise their number in any way. Sometimes, however, a composer is not only interested

in relaxing a constraint, but in finding a solution which optimally fulfills a soft constraint. For example, a composer may be interested in finding a solution which is guaranteed to have a minimal number of skips in the melody. However, strictly constraining all intervals to steps may not be possible, because some rules (e.g. harmonic rules) force some melodic skips.

In such a case, a rule like *preferSteps* can be redefined such that it constrains a goodness measure which expresses how well the rule is fulfilled (Fig. 7.8). An optimisation technique like best-solution search (Sec. 7.2.3.3) can then be used to find an optimal solution: this best-solution search would constrain any next solution to have a higher goodness measure than the previous one until no better solution can be found.

$$preferSteps2(intervals, goodness) \stackrel{def}{=}$$
$$howManyTrue(map(intervals, f : f(Interval) \stackrel{def}{=} Interval < 3), goodness)$$

Figure 7.8.: A variant of *preferSteps* (Fig. 7.7) which measures the goodness of the solution: a higher goodness value means more intervals are smaller than a minor third

This approach works well to optimise a single soft rule. Moreover, it can be extended for several soft constraints. Best-solution search optimises according to only a single comparison constraint. However, multiple goodness measures can be composed into a single 'parent' goodness measures (e.g. by weighting individual goodness measures with individual factors and summing all weighted goodness measures). Nevertheless, this approach is somewhat cumbersome, and the future work Sec. 11.2 points out a different approach.

7.2.3.5. Recomputation

Recomputation is a technique which saves memory (RAM), often at the expense of increased run time [Schulte, 2002]. Musical CSPs are often highly complex: these problems involve much information (memory-demanding data) and generate deep search trees. Recomputation is an essential technique for solving large problems which otherwise would require an excessive amount of memory.

Search performs decisions which can be wrong and possibly must be reversed (i.e. the path through the search tree must take 'the other turn' to find a solution). To this end, most constraint systems are based on *trailing*. Trailing records how the search process changes state so that it can go back to an earlier node of the search tree (backtrack) and explore a different branch.

The space-based constraint model replaces trailing by the two techniques: copying and recomputation. In this model, a node in the search tree and a copy of a computation space are independent concepts. Pure *copying* (also called pure *cloning*) leaves a clone of a parent space (with all the information it contains) at every branching node of the search tree. This approach allows the search process to move 'back' upwards to an 'earier'

space copy in the search tree and that way to reverse decisions. This approach makes distribution and exploration strategies fully programmable. However, it also requires a large amount of memory for large search trees. *Full recomputation*, in contrast, leaves only a single space copy in the root node of the search tree. Whenever a decision must be undone, the required computation space is reconstructed by repeating all decisions which led to it. Recomputing a node in the search tree only requires to know the path leading from a (direct or indirect) parent-node which has a space copy to the node to recompute. The original decisions are repeated by repeating the distribution (Sec. 7.2.3.2) for the nodes in this path. At each node 'on the way', those distribution constraints are added to the space copy which correspond with the continuation of the path to the node to recompute. The distribution constraints possibly cause constraint propagation before the next distribution 'on the way' is conducted.[7]

However, pure copying and full recomputation are two extreme cases. *Fixed recomputation* leaves a space copy at every n-th branch of the search tree, where n (the *maximal recomputation distance*) is defined by the user. *Adaptive recomputation* always leaves an additional space clone between a newly recomputed node and the parent node it is recomputed from. Adaptive recomputation results in more densely positioned space clones where failed spaces occur and so takes *clustered failures* into account (when search encounters a failed node, often the full subtree fails [Schulte, 2002]). Finally, *batch recomputation* [Choi et al., 2001] records all decisions of the distribution strategy and reconstructs a space by applying all recorded constraints in a single step which reduces the required runtime (e.g. omits the intermediate propagation steps). Recording all constraints applied by the distribution is only a very small overhead when compared with leaving a space copy at every search tree branch.[8]

In summary, recomputation allows the user to trade memory for run time and that way allows for the solving of complex CSPs. What's more, Schulte [2002] showed that recomputation can even require less run time than pure copying.

7.3. Specialising the Constraint Model for Music

The present section explains how Strasheela customises the space-based constraint model for a generic music constraint system.

7.3.1. The Basic Idea

7.3.1.1. Motivation

Experience in constraint programming in general has shown that the order in which variables are visited during the search process has an immense impact on the size of

[7] Recomputation requires a deterministic distribution strategy as was already pointed out in Sec. 7.2.3.2.

[8] Batch recomputation is presently not supported by Strasheela's implementation platform Mozart and thus not by the Strasheela prototype (Sec. 9).

the search space [Kumar, 1992]. This general experience has been confirmed for music constraint programming. For example, score-PMC (Sec. 3.3.1.2) applies a sophisticated variable ordering which is optimised for polyphonic music and is well-suited for arbitrarily complex rhythmical structures [Laurson, 1996]. The poor performance of Arno (Sec. 3.3.4), on the other hand, stems primarily from its naïve variable ordering which causes much redundant work [Anders, 2000].

Most existing music constraint systems apply a *static variable ordering*: the order in which variables are visited during the search process is settled before the search actually starts. However, musical CSPs can be extremely complex and it is often impossible to find an efficient static variable order. Consequently, only specific CSP classes are solved efficiently by static search orderings and some systems even make the definition of other CSPs impossible. For instance, score-PMC is limited to CSPs which fully determine the rhythmical structure in the problem definition: score-PMC depends on this rhythmical structure to compute its efficient static variable ordering [Laurson, 1996]. The search of score-PMC proceeds 'from left to right' in the polyphonic score: basically, notes with a smaller start time are visited earlier. To compute such a variable ordering before the search starts, it is of course essential that the temporal structure of the score is fixed.

In contrast to score-PMC, Arno allows the user to constrain the rhythmical structure and the pitch structure. Still, its static search order is only suited to a small class of musical CSPs. Arno determines the score voice-wise: after all notes of the first voice are determined, the search proceeds to the notes of the next voice. This approach detects conflicts between voices very late and only performs well in the case of canons.

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

A suitable variable ordering is highly problem-dependent. For example, a contrapuntal CSP might be solved best by processing from left to right in the score and completing all voices more-or-less in parallel. In contrast, a harmonic CSP might be solved best by first deciding upon an abstract harmonic progression (e.g. *I ii V I*), then for the pitches of the bass voice and the soprano voice and finally for the other voices.

A generic music constraint system therefore requires two features: it should allow for dynamic variable orderings and it should allow for defining variable orderings suitable for the problem at hand (and for selecting predefined variable orderings for convenience). As well as solving musical CSPs which could be solved by existing systems, such features allow for solving many CSPs which were intractable before. The importance of these two features (together with constraint propagation) can hardly be over-estimated for a system which aims to solve diverse musical CSPs reasonably efficiently.

7.3.1.2. Approach

The requirements detailed above led to the application of the space-based constraint model (Sec. 7.2) as a foundation for the generic music constraint system Strasheela. This model allows for a dynamic variable (and value) ordering and allows for the free definition of such orderings. A variable ordering is defined by defining a distribution strategy (Sec. 7.2.3.2) – independently of the problem definition.

The definition of a distribution strategy is orthogonal to the other features of the space-based constraint model. By defining special score distribution strategies, Strasheela preserves all other features of this constraint model. For example, Strasheela supports efficient constraint propagation, user-defined exploration strategies, reified constraints and recomputation (Sec. 7.2.3).

A distribution strategy bases its decisions on the information presently available in the partial solution. For example, Sec. 7.2.3.2 discussed the first-fail distribution strategy (which distributes some variable which – at the present stage of the search process – has the smallest domain). A generic music constraint system must allow for the definition of arbitrary score distribution strategies. Such a system requires that a score distribution strategy has access to any information contained in the partial solution score. The design of Strasheela fulfills these requirements. Instead of processing variables (as the distribution strategies discussed before), a score distribution strategy processes parameter objects (e.g. pitch instances, see Sec. 5.4.1). A parameter instance provides a link to its hosting item instance and so to the full music representation – a score distributor can thus make use of any information available for its decisions.

A distribution strategy reduces the domain of the distributed variable. A score distributor affects the value attribute of a parameter object – which is a variable. For example, a suitable technique for many musical CSPs is to determine this variable setting it to a (pseudo) randomly selected value of its domain.

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

Section 6.3 discussed the problem that musical CSPs often constrain score contexts which are inaccessible in the problem definition. As long as these contexts are inaccessible, constraints on these contexts propagate very weakly (if at all). For example, if simultaneous notes are constrained by a harmonic rule then the constraints of this rule rule can only propagate for notes which are known to be simultaneous. If the rhythmical structure is undetermined in the problem definition – and hence the context 'simultaneous notes' is inaccessible – propagation blocks until this context becomes inaccessible. A distribution strategy for a CSP which constrains inaccessible score contexts should therefore take special care to resolve these contexts at an early stage so as to make propagation

possible. In analogy to the *first-fail* principle for variable orderings (Sec. 7.2.3.2), the present research calls this guideline the *resolve-inaccessible-contexts* principle. For instance, in many musical CSPs the temporal structure should be determined relatively early in case-dependent contexts (for example where simultaneous notes are inaccessible but constrained).

7.3.2. Score Distribution Strategies

The present section proposes a number of score distribution strategies which cover a large number of musical CSPs. Further strategies can be defined by the user according to needs.

To make the discussion of score distribution strategies more concise, the high-level interface for the definition of distribution strategies introduced in Sec. 7.2.3.2 has been adapted for score distribution strategies. The original interface expects two functions: a binary *order* function which defines the variable ordering and a unary *value* function which defines how the domain of the distributed variable is reduced. In the adapted interface, the *order* function expects two parameter instances, but the *value* function remains the same.[9]

7.3.2.1. Adapting the First-Fail Principle

Section 7.2.3.2 discussed the first-fail principle as a rule-of-thumb used to formulate distribution strategies. This guideline is also well-suited to many musical CSPs. Typical strategy examples either distribute a variable with the smallest domain or alternatively a variable on which most constraints are applied.

[9] This adapted interface is very similar to the interface defined in footnote 6. The new definition substitutes occurrences of a single variable in the former definition by accessing this variable from a parameter object using the method `getValue`.

```
proc {MyScoreDistributor unit(order:Ord value:Val) Xs}
    {Space.waitStable}
    local
        Params={Filter Xs fun {$ X} {FD.reflect.size {X getValue($)}} > 1 end}
    in
        if Params \= nil
        then
            Var = {{Order Params Ord} getValue($)}
            Dom = {Val Var}
        in
            choice {FD.int Dom Var} [] {FD.int compl(Dom) Var} end
            {MyDistributor unit(order:Ord value:Val) Params}
        end
    end
end
```

The definition of the first-fail variable ordering function for parameter objects differs only slightly from the ordering function for plain variables. The order expressed by the function *isParamWithSmallerDomainSize* (Fig. 7.9) results in a variable ordering which distributes the value of some parameter with the smallest domain size. The only difference to the first-fail strategy for plain variables presented above is that the version for parameter instances has to access the parameter values by *getValue*.

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

Figure 7.9.: Variable ordering specification of first-fail strategy for parameters

A full distribution strategy specification complements the variable ordering with a decision as to how the variable domain is reduced (the term value ordering denotes a slightly less general concept but will still be used here for brevity). Commonly, the first-fail strategy decides on the smallest domain value. For musical CSPs, a heuristics which randomly selects a domain value is often preferable. Still, such a randomised decision must be deterministic such that recomputed decisions would decide the same way (cf. Sec. 7.2.3.2).

Order: *isParamWithSmallerDomainSize*

Value: For example *getMinimalDomainValue* or *getRandomDomainValue*

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

7.3.2.2. Resolving Inaccessible Score Contexts

Resolving a Single Context

The general first-fail principle is primarily suitable for CSPs in which all constrained score contexts are already accessible in the problem definition. Otherwise, inaccessible contexts which are constrained (Sec. 6.3) should be resolved early to make the propagation of these constraints possible (resolve-inaccessible-contexts principle, Sec. 7.3.1.2).

A typical example of an inaccessible but constrained score context is the context of simultaneous score objects in case the temporal structure of the music is undetermined in the problem definition. One variable ordering which addresses this inaccessible context determines all temporal parameters first and other parameters only later.

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

7.10). In the case where *isTemporalParameter* (which tests class membership) returns *true* for the first parameter, then the ordering function *preferTemporalParameter* returns *true* and this parameter will be distributed. Alternatively, if *preferTemporalParameter* returns *false* then this expresses a preference for $param_2$ – whether it actually is a temporal parameter or not. In the case where $param_2$ is not a temporal parameter, then this preference will be overruled as long as there is still any undetermined temporal parameter. In the case where there is no undetermined temporal parameter left, it is adequate to distribute a parameter which is not a temporal parameter.[10]

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

Figure 7.10.: Variable ordering specification which determines all temporal parameters first in order to make score contexts based on the temporal structure (e.g. simultaneous notes) accessible

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

Resolving Multiple Contexts in Order

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

[10] The efficiency of such a distribution strategy can be improved. The class membership of parameter objects does not change during the search process. Therefore, it is sufficient to compute a static variable ordering which the distribution strategy only executes without further checks during the search process. Such an improvement requires a minor change of the high-level interface definition `MyScoreDistributor` (fn. 9) for the definition of static variable orderings. The implementation of this adapted definition is omitted here for brevity: basically, the line calling the function `Order` is omitted and the distributor expects an ordered list of parameter objects.

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

The function *determineInOrder* defines such a generic means. It expects a specification stating the parameters affected and the order in which they are determined. This specification is given as a list of unary boolean functions – the order of the test functions specifies the order in which the corresponding parameters are determined. For example, Fig. 7.11 defines the variable ordering for the harmonic CSP proposed above. This ordering first determines all temporal parameters, then all chord parameters, and finally all note pitches.

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

Figure 7.11.: A variable ordering for harmonic CSPs which first determines the temporal structure of the score, then the harmonic structure and finally the actual pitches

The function *determineInOrder* is defined in Fig. 7.12. The function expects a list of test functions and returns a variable ordering function which compares two parameter objects and decides accordingly. For its decision, *determineInOrder* makes use of the function *getTestIndex*. This function expects a list and a boolean function *test* and returns the position of the first element for which this function *test* returns *true*.[11] The variable ordering function created by *determineInOrder* calls *getTestIndex* with both its parameters and decides on the parameter for which the smaller index is returned (in case *getTestIndex* returns equal values for two parameters, the strategy opts for the first parameter).[12]

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

Figure 7.12.: The function *determineInOrder* returns a variable ordering function which determines parameters in the order specified by a list of test functions

[11] The function function *getTestIndex* is defined in Fig. B.5.

[12] Again, distribution strategies which only exploit score information which does not change during the search process are implemented more efficiently by a static variable ordering which only performs the ordering once before the search starts.

Combining the Principles Resolve-Inaccessible-Contexts and First-Fail

The variable orderings discussed above distribute selected parameters (i.e. parameters for which some test function such as *isTemporalParameter* returns *true*) in an arbitrary order. For example, the variable ordering which first determines the temporal structure (Fig. 7.10) does not express any preference as to which temporal parameter to distribute in case there are multiple undetermined temporal parameters (or which non-temporal parameter to distribute in case all remaining undetermined parameters are non-temporal parameters).

For many CSPs, a more suitable strategy combines the resolve-inaccessible-contexts principle with the general first-fail principle. For example, instead of determining all temporal parameters (or non-temporal parameters) in no particular order, some temporal parameter with smallest domain size is distributed first. Alternatively, some parameter on which most constraints are applied is chosen. Figure 7.13 expresses a variable ordering which first tests whether one of two parameters is a temporal parameter, and then decides for the temporal parameter. If either both or none of the parameters is a temporal parameter, then the parameter with smaller domain size is distributed.[13] The function *isParamWithSmallerDomainSize* is defined in Fig. 7.9.

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

Figure 7.13.: Variable ordering specification which in general determines all temporal parameters before the non-temporal parameters, but determines a temporal (or non-temporal) parameters with smallest domain size first.

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

7.3.2.3. The Left-to-Right Variable Ordering

For many musical CSPs, a suitable distribution strategy determines parameter objects 'from left to right'. Such an approach distributes parameters in order of the start times of the items (events or temporal containers) these parameters belong to. The resulting

[13] Again, the efficiency of such a distribution strategy can be improved. The parameters could first be sorted into temporal parameters and non-temporal parameters before the search starts. During search, the distribution first processes temporal parameters to find the parameter with the smallest domain size. After all temporal parameters are determined, then the search process processes non-temporal parameters.

variable ordering is very similar to the variable ordering applied by score-PMC (Sec. 7.3.1.1, [Laurson, 1996]).[14] However, score-PMC estimates a static variable ordering before the search starts and therefore requires that the rhythmical structure of the music is fully determined in the problem definition. A distribution strategy, on the other hand, allows for a dynamic variable ordering. A distribution strategy can be defined in such a way that it proceeds 'from left to right' whether the rhythmical structure of the music is determined or not.

Figure 7.14 defines a left-to-right dynamic variable ordering. The main criteria for the variable ordering are the start times of the temporal items to which the parameter objects in question belong. The function *isLeftmostParam* first accesses the parameters' associated start times and checks whether these start times are already determined (i.e whether *getDomainSize* returns 1) and then bases its decision mainly on the values of these start times. In case only one of these two start times is already determined, the respective parameter is preferred (else-clause of outer if-expression).[15] In case both start times are determined, *isLeftmostParam* opts for the parameter to which the smaller start time belongs (else-clause of inner if-expression). Finally, in case both start times are equal then *isLeftmostParam* prefers a temporal parameter. This condition causes a relatively early determination of temporal parameters.

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

Figure 7.14.: A left-to-right dynamic variable ordering

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

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

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

The dynamic left-to-right variable ordering presented in this section turns out to be highly suitable for a wide range of musical CSPs. In particular, it allows for solving polyphonic problems which are very hard or even impossible to solve using existing systems. In addition, this variable ordering is applicable for arbitrarily nested score topologies and therefore is a good candidate for a default variable ordering for musical CSPs in general. Section 8.2.2 will compare the efficiency of this strategy with other strategies.

8. Strasheela Examples

This chapter shows Strasheela in action with concrete musical examples. In these examples, the three main aspects of this research – the music representation (Chap. 5), the rule formalism (Chap. 6), and the search approach (Chap. 7) – work together.

The purpose of this chapter is mainly to demonstrate Strasheela's usage on full examples. These examples model relatively simple and highly conventional music theories. The explanation of a conventional theory can be brief and refer to the literature for further details. The implementation of this theory as a computational model, on the other hand, can be discussed in full detail for a relatively simple theory. Nevertheless, this chapter is *not* intended to appraise the expressive power of Strasheela by a few examples. Instead, chapter 10 provides a more formal evaluation by comparing Strasheela and existing systems using a few general criteria.

Strasheela's capabilities go far beyond what is shown in this chapter. In fact, Strasheela users are free to implement their own rule-based music theory. Several further examples are provided with the Strasheela prototype. They can be found online at `http://strasheela.sourceforge.net/strasheela/doc/StrasheelaExamples.html`. These examples are given in Oz code and can thus be executed (and also edited at will). For example, Strasheela has been used to create music in extended just intonation where compositional rules control aspects of a microtonal harmonic progression such as a smooth transition between chords (e.g. by constraining the consonance/dissonance-degree and the root relations between neighbouring chords as well as their degree of familiarity with respect to chords in common practice).

Strasheela also allows its user to combine the constraint-based approach with other conventional techniques of algorithmic composition. For instance, Lindenmayer systems (L-system) [Prusinkiewicz and Lindenmayer, 1990] are an established device in algorithmic composition used to create highly self-similar musical structures [Supper, 2001]. An L-system has been applied to create the global musical form where different L-system symbols are interpreted as different motifs. The local details of the score however – in particular the pitch structure and their resulting harmony – are constrained by compositional rules.

Chapter Overview

Section 8.1 explains how Fuxian first species counterpoint for two voices can be defined and solved in Strasheela. The subsequent florid counterpoint example (Sec. 8.2) constrains both the rhythmic and the pitch structure. This calls for a suitable score

distribution strategy. Finally, Section 8.3 demonstrates how an important aspect of the musical form – the score topology – can be constrained.

8.1. Fuxian First Species Counterpoint

The example presented in this section formalises the first-species counterpoint for two voices as described by Fux [1725, Chap. 1]. This example will be presented in full so as to demonstrate how the different parts of Strasheela work together. At first, the example is stated in musical terms (Sec. 8.1.1). Section 8.1.2 formalises the example. Section 8.1.3 discusses the search process and presents a musical result. A conclusion (Sec. 8.1.4) summarises this example.

8.1.1. The Music Theory

In first species counterpoint for two voices, the task is writing a fitting counter-melody (the *counterpoint*) for a given melody (the *cantus firmus*).

First species counterpoint is homophonic. Note durations are irrelevant in the first species: notes of parallel voices always start and end together (i.e. all notes are of equal length, usually all notes are semibreves). Also, both voices start and end together (i.e. the cantus firmus and the counterpoint have the same number of notes).

Some rules restrict the melodic aspect of the counterpoint writing. Only specific melodic intervals are allowed. These are the intervals up to the fourth, the fifth, and the octave (no note repetition is permitted here). All notes must be diatonic pitches (i.e. there can be no augmented, diminished, or chromatic melodic intervals). The counterpoint remains in a narrow pitch range. Melodic steps are preferred (this rule is not mentioned by Fux [1725, Chap. 1], but it is very important for acceptable results and the Fuxian examples follow this rule as well).

Furthermore, some rules restrict the relation between both voices. Open and hidden parallels are forbidden, that is, direct motion in a perfect consonance is not allowed. Only consonances are permitted as intervals between simultaneous notes and there should be more imperfect than perfect consonances. The first and last notes, however, must form a perfect consonance. Finally, the counterpoint must be in the same mode as the cantus firmus.[1]

[1] Some Fuxian rules are omitted here for brevity. The omitted rules are the following:

- No melodic skips follow each other in the same direction.
- Skips must be compensated for.
- The last-but-one note pitch of the counterpoint must form a cadence where – depending on the mode – the counterpoint is raised by a semitone. The last-but-one pitch is always the *II* degree for the cantus firmus and the *VII* degree for the counterpoint. For example, in Dorian mode without any accidentals the last counterpoint pitch is always *c*♯.

8.1.2. The Formal Model

A musical CSP consists of a music representation containing the variables and the rules which constrain these variables. Firstly, the music representation is shown (Sec. 8.1.2.1). Section 8.1.2.2 then defines the rules one by one.

8.1.2.1. The Music Representation

The music representation models important aspects of the music theory described in Sec. 8.1.1. Only the actual rules are not modelled by the music representation itself.

The solution score contains two voices: the cantus firmus and the counterpoint. A voice can be represented by a sequential container (Sec. 5.4.4) which contains the notes of the voice. Two parallel voices can be represented by nesting their two sequential containers in a simultaneous container. The resulting full score topology is thus as shown below.

$$\begin{aligned} makeScore(sim(items\colon [seq(items\colon [note, \ldots])\\ seq(items\colon [note, \ldots])])) \end{aligned}$$

The cantus firmus is given, that is, this melody is fully determined. This example uses the first cantus firmus introduced by Fux which is shown in Fig. 8.1. Each note duration is a semibreve, that is, each duration has the value 4 if the *timeUnit* is set to *beats* (Sec. 5.4.1.1). When the pitches of this melody are measured in key-numbers (i.e. the *pitchUnit* is set to *midi*), these pitches are represented by the following sequence.

$$[62, 65, 64, 62, 67, 65, 69, 67, 65, 64, 62]$$

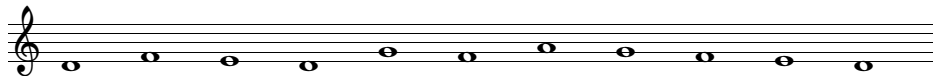


Figure 8.1.: The cantus firmus

Much information is available about the counterpoint. The number of notes in this voice is the same as in the cantus firmus (i.e. 11). The duration of each note is also determined to be a semibreve (i.e. 4). The only variables in this CSP are the pitches of the counterpoint.

- A tone can be repeated once at maximum (instead, the example shown below completely prohibits repetitions).
- The melody must not expose any tritone, even when this interval is reached stepwise (in the example shown below, only the tritone between two neighbouring notes is prohibited).
- From an interval larger than an octave contrary motion into an octave is not allowed.

These pitches are restricted to a small range (here, an octave and a third). The actual domain of each note pitch depends on whether the counterpoint is situated above or below the cantus firmus. In the present example, the counterpoint is the upper part and the pitch domain for each of its voices is set to $\{60, \ldots, 76\}$ (i.e. $c4$–$e5$)

Figure 8.2 shows the top-level definition of the example. The function *firstSpeciesCounterpoint* creates the music representation, applies all rules to this representation and returns the solution *myScore*. The rules are defined in the following section.

```
let /* Auxiliary function to create a single voice, represented by a sequential container. makeVoice
       expects a list of pitches (i.e. constrained variables) which are incorporated into note objects. */
    makeVoice(pitches) def=
      makeScore(seq(items: map(pitches,
                               f : f(Pitch) def=
                                 note(duration: 4, /* semibreve (four beats) */
                                      pitch: Pitch,
                                      pitchUnit: midi))))
in  firstSpeciesCounterpoint() def=
      let cantusFirmus def= makeVoice([62, 65, 64, 62, 67, 65, 69, 67, 65, 64, 62])
          /* fdList returns a list of finite domain integers. The list length is set to 11 (the length
             of the cantus firmus) where each integer has the domain {60, ..., 76}                  */
          counterpoint def= makeVoice(fdList(11, {60, ..., 76}))
          myScore def= makeScore(sim(items: [counterpoint, cantusFirmus],
                                     startTime: 0,
                                     timeUnit: beats))
      in  /* Return myScore and apply a number of compositional rules to the counterpoint */
          myScore where
            restrictMelodicIntervals(counterpoint)
            ∧ onlyDiatonicPitches(counterpoint)
            ∧ noDirectMotionIntoPerfectConsonance(counterpoint)
            ∧ onlyConsonances(counterpoint)
            ∧ preferImperfectConsonances(counterpoint)
            ∧ startAndEndWithPerfectConsonance(counterpoint)
```

Figure 8.2.: Top-level definition: *firstSpeciesCounterpoint* creates the music representation and applies all rules to it (the rules are defined in Sec. 8.1.2.2)

8.1.2.2. The Compositional Rules

The top-level definition *firstSpeciesCounterpoint* (Fig. 8.2) applies six rules to the counterpoint. The first two rules (*restrictMelodicIntervals* and *onlyDiatonicPitches*) implement the melodic restrictions specified in Sec. 8.1.1, the remaining rules (*noDirect-*

MotionIntoPerfectConsonance, *onlyConsonances*, *preferImperfectConsonances*, and *startAndEndWithPerfectConsonance*) implement the relation between both voices. The formalisation of these rules is explained in this subsection.

The rule *restrictMelodicIntervals* (Fig. 8.3) constrains the melodic intervals between all neighbouring note pitch pairs of a given voice. The actual constraints are expressed by the two auxiliary constraints *restrictInterval* and *preferSteps*. The rule *restrictMelodicIntervals* merely applies these constraints to the intervals between the neighbouring note pitches of a given voice (the auxiliary context accessor *getInterval* is defined later in Fig. 8.10 and *map2Neighbours* was introduced in Fig. 6.4).

$$\textbf{let}\ \ \mathit{restrictInterval}(\mathit{Interval}) \overset{def}{=} \mathit{Interval} \in \{1, \ldots, 5, 7, 12\}$$

/* *preferSteps* constrains the average of *intervals* to be in {1.5, ..., 3.0}, that is the average interval is between $1\frac{1}{2}$ semitones and a minor third. */

$$\mathit{preferSteps}(\mathit{intervals}) \overset{def}{=} \textbf{let}$$

/* Encoded floating point arithmetic mean value (see below) */

$$\mathit{Mean} \overset{def}{=} \mathit{fdInt}(\{15, \ldots, 30\})$$

in /* To represent a floating point value by a constrained variable with integer domain, the arithmetic mean is multiplied by 10. */

$$\mathit{Mean} = 10 \frac{\sum \mathit{intervals}}{\mathit{length}(\mathit{intervals})}$$

$$\textbf{in}\ \ \mathit{restrictMelodicIntervals}(\mathit{myVoice}) \overset{def}{=}$$

$$\textbf{let}\ \ \mathit{intervals} \overset{def}{=} \mathit{map2Neighbours}(\mathit{getItems}(\mathit{myVoice}), \mathit{getInterval})$$

$$\textbf{in}\ \ \bigwedge \mathit{map}(\mathit{intervals}, \mathit{restrictInterval}) \wedge \mathit{preferSteps}(\mathit{intervals})$$

Figure 8.3.: Only certain melodic intervals are allowed and small intervals are preferred

The constraint *restrictInterval* expresses that only specific melodic intervals are permitted, namely every interval between a minor second and a fourth, as well as a fifth and an octave. Implicitly, *restrictInterval* forbids repetitions as the unison is not permitted.

The constraint *preferSteps* expresses a preference for smaller intervals: *preferSteps* has the effect that a solution exhibits more steps than larger intervals. This function constrains the average interval (i.e. the arithmetic mean) to an interval between $1\frac{1}{2}$ semitones and a minor third.

Figure 8.4 defines the second melodic rule which constrains all notes contained its argument *myScore* (i.e. the counterpoint in this example) to diatonic pitches. These are the pitches of the Dorian mode starting at *d* (i.e. the 'white keys' on the piano). This rule is very brief, because it makes use of the constraint *isDiatonicPitch*, already defined in Fig. 6.9. As a demonstration, the rule is defined more generally than required for this specific example. The rule collects all notes contained either directly in the container *myScore* or in some sub-containers. This allows the user to apply this rule to a

voice which also contains explicit pause objects or to a polyphonic score consisting of a hierarchy of containers (the function *collect* was introduced in Fig. 5.24).

$$
\begin{array}{l}
onlyDiatonicPitches(myScore) \stackrel{def}{=} \\
\quad \textbf{let} \;\; pitches \stackrel{def}{=} map(collect(myScore, test\colon isNote), getPitch) \\
\quad \textbf{in} \;\; \bigwedge map(pitches, isDiatonicPitch)
\end{array}
$$

Figure 8.4.: All notes must be diatonic pitches (i.e. there can be no augmented, diminished, or chromatic melodic intervals)

The remaining rules constrain the relationship between both voices. The voice-leading rule *noDirectMotionIntoPerfectConsonance* (Fig. 8.5) forbids open and hidden parallels. This rule subsumes the 'four fundamental rules' of Fux [1725] into a single rule: perfect consonances must not be reached in direct motion. The last line of the rule expresses this statement: precisely if the interval (*Interval*) between two simultaneous notes is a perfect consonance then the directions of the voices leading into this interval (Dir_1 and Dir_2) must differ. The lines before this statement define these variables *Interval*, Dir_1 and Dir_2. The function f is applied to every pair *notePre* and *noteSucc* of neighbouring notes in the *counterPoint*. The simultaneous notes of these two notes (i.e. the notes of the cantus firmus) are accessed by the function *getSimNote* (an auxiliary function defined below in Fig. 8.9). The directions (Dir_1 and Dir_2) of the melodic intervals are constrained: the constraint *direction* encodes this direction by a finite domain integer (the actual encoding is irrelevant here as the directions must differ only; the function *direction* is defined in Fig. B.6). Finally, *Interval* is constrained to the absolute distance between the pitches of *noteSucc* and its simultaneous note by the auxiliary function *getInterval*.

The harmonic rule *onlyConsonances* (Fig. 8.6) constrains each interval between simultaneous notes to be consonant. The constraint *isConsonance* is defined in Fig. 8.11 below.

The rule *preferImperfectConsonances* expresses a preference for imperfect consonances. More specifically, the rule constrains the number of perfect consonances between simultaneous notes to less than half of the total number of voice notes. The constraint *isPerfectConsonance* is defined in Fig. 8.12.

The rule *startAndEndWithPerfectConsonance* (Fig. 8.8) constrains the first and the last note of the counterpoint to form a perfect consonance with the simultaneous cantus firmus note (i.e. either a prime, a fifth or an octave). In case the counterpoint is below the cantus firmus, however, their interval must form an octave. A fifth is not possible as it would change the mode of the cantus firmus [see Fux, 1725, p. 31].

The remainder of this section defines the auxiliary functions which have been used in the rules. The auxiliary context accessor *getSimNote* (Fig. 8.9) returns the (single) note which is simultaneous to *myNote*.

The function *getInterval* (Fig. 8.10) returns the constrained variable *Interval* and con-

$$
\begin{aligned}
&noDirectMotionIntoPerfectConsonance(counterPoint) \stackrel{def}{=} \\
&\quad \bigwedge map2Neighbours(getItems(counterPoint), \\
&\qquad\qquad f : f(notePre, noteSucc) \stackrel{def}{=} \\
&\qquad\qquad\quad \textbf{let}\ \ Dir_1\ Dir_2\ Interval \\
&\qquad\qquad\quad \textbf{in}\quad Dir_1 = direction(getPitch(notePre), \\
&\qquad\qquad\qquad\qquad\qquad\qquad getPitch(noteSucc)) \\
&\qquad\qquad\qquad\quad \wedge Dir_2 = direction(getPitch(getSimNote(notePre)), \\
&\qquad\qquad\qquad\qquad\qquad\qquad getPitch(getSimNote(noteSucc))) \\
&\qquad\qquad\qquad\quad \wedge Interval = getInterval(noteSucc, \\
&\qquad\qquad\qquad\qquad\qquad\qquad getSimNote(noteSucc)) \\
&\qquad\qquad\qquad\quad \wedge isPerfectConsonance(Interval) \Rightarrow Dir_1 \neq Dir_2)
\end{aligned}
$$

Figure 8.5.: Open and hidden parallels are forbidden: perfect consonances must not be reached by both voices in the same direction

$$
\begin{aligned}
&onlyConsonances(counterPoint) \stackrel{def}{=} \\
&\quad \bigwedge mapItems(counterPoint, \\
&\qquad\qquad f : f(note) \stackrel{def}{=} isConsonance(getInterval(note, getSimNote(note))))
\end{aligned}
$$

Figure 8.6.: The interval between every pair of simultaneous note pitches is consonant

$$
\begin{aligned}
&preferImperfectConsonances(counterpoint) \stackrel{def}{=} \\
&\quad \textbf{let}\ \ notes \stackrel{def}{=} getItems(counterpoint) \\
&\qquad simIntervals \stackrel{def}{=} map(notes,\ f : f(note) \stackrel{def}{=} \\
&\qquad\qquad\qquad\qquad getInterval(note, getSimNote(note))) \\
&\qquad NumberPerfectConsonances \\
&\quad \textbf{in}\quad \text{/* Summation of the truth values (0/1-variables) whether the intervals are perfect consonances. */} \\
&\qquad NumberPerfectConsonances = \sum map(simIntervals, isPerfectConsonance) \\
&\qquad NumberPerfectConsonances < \frac{length(notes)}{2}
\end{aligned}
$$

Figure 8.7.: Imperfect consonances are preferred over perfect consonances

$$
\begin{array}{ll}
\textbf{let} & \mathit{isSuitableInterval}(\mathit{CounterpointPitch}, \mathit{CantusPitch}) \stackrel{def}{=} \\
& \quad (\mathit{CounterpointPitch} - \mathit{CantusPitch}) \in \{-12, 0, 7, 12\} \\
\textbf{in} & \mathit{startAndEndWithPerfectConsonance}(\mathit{counterpoint}) \stackrel{def}{=} \\
& \quad \textbf{let} \ \mathit{firstNote} \stackrel{def}{=} \mathit{first}(\mathit{getItems}(\mathit{counterpoint})) \\
& \qquad \mathit{lastNote} \stackrel{def}{=} \mathit{last}(\mathit{getItems}(\mathit{counterpoint})) \\
& \quad \textbf{in} \quad \mathit{isSuitableInterval}(\mathit{getPitch}(\mathit{firstNote}), \\
& \qquad\qquad\qquad\qquad \mathit{getPitch}(\mathit{getSimNote}(\mathit{firstNote}))) \\
& \qquad \mathit{isSuitableInterval}(\mathit{getPitch}(\mathit{lastNote}), \\
& \qquad\qquad\qquad\qquad \mathit{getPitch}(\mathit{getSimNote}(\mathit{lastNote})))
\end{array}
$$

Figure 8.8.: The first and the last counterpoint note forms a perfect consonance with the cantus firmus

$$\mathit{getSimNote}(\mathit{myNote}) \stackrel{def}{=} \mathit{first}(\mathit{getSimultaneousNotes}(\mathit{myNote}))$$

Figure 8.9.: Access the simultaneous note of a given note

strains *Interval* to the absolute distance between the pitches of two given notes.[2]

$$
\begin{array}{lll}
\mathit{getInterval}(\mathit{note}_1 \mathit{note}_2) \stackrel{def}{=} & \textbf{let} & \mathit{Interval} \\
& \textbf{in} & \mathit{Interval} \ \textbf{where} \\
& & \quad \mathit{Interval} = |\mathit{getPitch}(\mathit{note}_1) - \mathit{getPitch}(\mathit{note}_2)|
\end{array}
$$

Figure 8.10.: Return the interval between the pitches of two notes (a constrained variable)

The last two auxiliary functions define the constraints *isConsonance* (Fig. 8.11) and *isPerfectConsonance* (Fig. 8.12). Both definitions are very similar and constrain a given *Interval* so that it is a member of a set of consonances (minor and major third, fifth, ...) respectively perfect consonances (prime, fifth, octave). Note that *isConsonance* omits the prime, which is not allowed here as an interval between simultaneous notes.

[2]This auxiliary function makes several rule definitions more clear but impairs efficiency. This function is called by several rules on the same note pairs and thus creates redundant constrained variables and propagators.

A possible optimisation of this rule which retains its modularity uses memoization [Norvig, 1992]. Such an optimised version of *getInterval* maintains an internal table for two note combinations. In case the interval between two notes was constrained before, the rule returns that interval. Otherwise a new constrained variable is created, stored in the table and returned.

$$isConsonance(Interval) \stackrel{def}{=} Interval \in \{3, 4, 7, 8, 9, 12, 15, 16\}$$

Figure 8.11.: Constrain *Interval* to a consonance

$$isPerfectConsonance(Interval) \stackrel{def}{=} Interval \in \{0, 7, 12\}$$

Figure 8.12.: Constrain *Interval* to a perfect consonance

8.1.3. Search Process and Results

This section discusses the search process for the example. Two different score distribution strategies are applied and their performance is compared. Finally, an example result is presented.

A distribution strategy consists of a variable ordering and a value ordering (see Sec. 7.3.2). In this particular CSP, all constraints can be applied 'directly': the music representation contains enough information in the problem definition such that there are no inaccessible constrained score contexts (Sec. 6.3).[3] Therefore, an established general variable ordering such as first-fail – which distributes some variable with smallest domain – is suitable (Sec. 7.3.2.1). The first-fail variable ordering has been complemented with the value ordering which selects the domain value that is closest to the arithmetical means of the domain boundaries.

It turns out that constraint propagation carries out most of the work to solve this CSP (i.e. to successfully determine all 11 pitches of the counterpoint). Figure 8.13 shows the resulting search tree for finding the first solution which consists of only 22 distributable spaces and 16 failed spaces – besides the solution (see Sec. 7.2.2 for the meaning of the tree node shapes). The tree has a depth of 23. The search process requires about 50 milliseconds.[4]

For comparison, the left-to-right variable ordering (Sec. 7.3.2.3) – which is particularly suited to solving polyphonic CSPs – has been applied as well (with the same value ordering, that is, using *getMediumDomainValue*). For this CSP, the first-fail and the left-to-right variable orderings are comparable in efficiency (in great contrast to the next example). The left-to-right distribution also makes 22 decisions until it finds the first solution (the shape of the search tree is different) and also requires about 50 milliseconds.

One solution of this CSP is presented in Fig. 8.14. For this solution, a randomised value

[3] In fact, the rule *noDirectMotionIntoPerfectConsonance* (Fig. 8.5) does constrain a context inaccessible in the problem definition (only during search does it become known which simultaneous notes form a perfect consonance). Still, this rule does not call for a special distribution strategy.

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

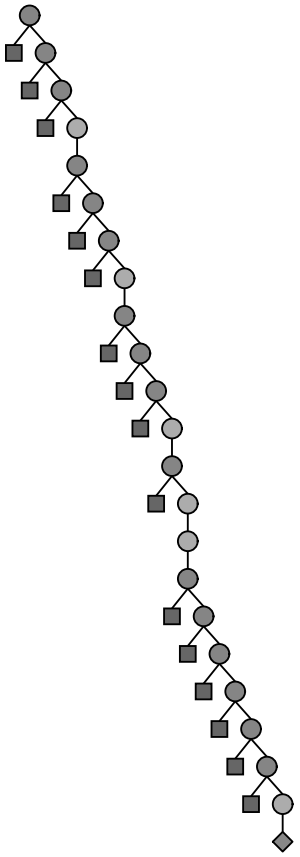


Figure 8.13.: The first-fail distribution strategy results in a relatively small search tree to find the first solution for this example (i.e. to determine all 11 pitches of the counterpoint)

ordering has been used. Still, the search process was comparable in efficiency to the figures reported above.

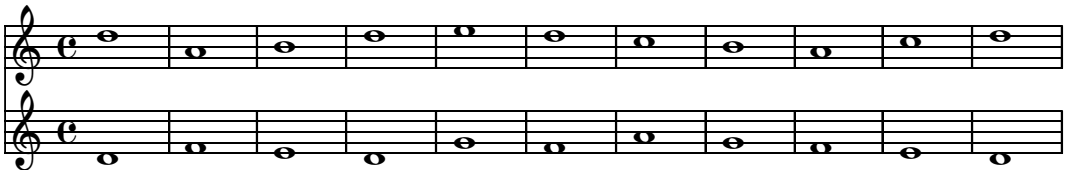


Figure 8.14.: One solution for the CSP (using random value ordering)

8.1.4. Conclusion

This section showed Strasheela 'in action' on a full example. The example demonstrated how the different aspects of Strasheela's design (its music representation, rule definition, rule application and search process) work together.

The top-level definition *firstSpeciesCounterpoint* (Fig. 8.2) creates the music representation used in this example. This representation consists of a hierarchy of score objects (containers and notes). The whole score is largely determined in the problem definition – only the counterpoint pitches are undetermined.

These pitches are constrained by six rules. The rules are defined in a modular way as functions (Fig. 8.3 to 8.8). In this particular example, every rule is applied directly to the counterpoint within the top-level definition *firstSpeciesCounterpoint*.

These rules constrain diverse score contexts (Sec. 2.4). Every score context constrained in this example is listed here to show their diversity (together with the corresponding rule).

- A single note pitch (*onlyDiatonicPitches*, Fig. 8.4)
- A pair of neighbouring note pitches (*restrictMelodicIntervals*, Fig. 8.3)
- A pair of simultaneous note pitches (*onlyConsonances*, Fig. 8.6)
- The set of all sets of simultaneous note pitches (*preferImperfectConsonances*, Fig. 8.7)
- A more complex note pitch set consisting of neighbouring and simultaneous note pitches (*noDirectMotionIntoPerfectConsonance*, Fig. 8.5)

In this example, the scope of each rule (Sec. 3.3.1.1) is defined within the rule (i.e. each rule was only applied once and accessed the set of constrained score contexts within the rule). Some rules apply their constraints uniformly to all instances of a particular score context. For example, *onlyConsonances* applies a constraint to every pair of simultaneous note pitches. Other rules constrain only specific instances of a certain context. For

instance, *startAndEndWithPerfectConsonance* constrains only the first and last pair of simultaneous note pitches.

A number of higher-order functions are also used to conveniently access score contexts or to apply a rule to the score. As a clarification, these functions are listed here.

- Rules are applied to every element of a list with *map* (e.g. in *onlyDiatonicPitches*, Fig. 8.4)
- Rules are applied to neighbouring elements in a list with *map2Neighbours* (e.g. in *noDirectMotionIntoPerfectConsonance*, Fig. 8.5)
- Complex score contexts are expressed by traversing the whole score hierarchy to collect all objects which meet a certain condition: *getSimNote* (Fig. 8.9) is implemented by *getSimultaneousNotes*, a variant of *getSimultaneousItems* (Fig. 5.31)
- Rules are applied to every element of a list which meets a condition (implicit filtering) with *mapItems* (e.g. in *onlyDiatonicPitches*, Fig. 8.4)

This section also discussed the search process for this example. It was shown that much of the search process is conducted by constraint propagation alone. Two distribution strategies were compared and revealed little difference in performance for this example. The subsequent example, however, will constrain the rhythmical and the pitch structure of the music and thus constrain score contexts which are not accessible in the problem definition. This subsequent example will demonstrate the enormous influence of a suitable distribution strategy on the size of the search space and thus the efficiency of the search process.

Comparison with Existing Systems

Strasheela has been designed for solving complex musical CSPs. Fuxian first-species counterpoint, however, was chosen as a first example because it is familiar and relatively simple. Although this section only reproduced a standard example, the Strasheela implementation still demonstrates some of Strasheela's features which are important for addressing more complex CSPs.

The Strasheela user creates the music representation explicitly (cf. Fig. 8.2) which results in a high degree of flexibility. For example, the user may reshape the score topology (i.e. further structure the score by introducing additional containers) or replace the predefined *note* objects by user-defined objects. Existing music constraint systems such as PWConstraints (Sec. 3.3.1) and Situation (Sec. 3.3.2) do not support such flexibility.

In Strasheela, the rule application is transparent (i.e. the user can understand how the rule application is working [Raymond, 2003]) and user-programmable. To simplify the rule definitions for the present example, the rules are directly applied to the *counterpoint* container and not to the whole score (cf. Fig. 8.2). PWConstraints and Situation support

neither accessing parts of the music representation outside a rule nor direct rule application. In the example above, each rule encapsulates its own rule application mechanism. For example, the rule *noDirectMotionIntoPerfectConsonance* employs the higher order function *map2Neighbours* (cf. Fig. 8.5). Each of these mechanisms can be defined and thus freely changed by the user (e.g. Sec. 6.2.4 defines *map2Neighbours*). PWConstraints and Situation, on the other hand, provide proprietary rule application mechanisms which are limited with respect to transparency and are not user-programmable. Although these systems support all score contexts required for this example (in Situation, it will be hard to express the context constrained by the rule *noDirectMotionIntoPerfectConsonance*), they will fail for score contexts which were not explicitly arranged for by their designers. Chapter 10 will conduct a systematic and detailed comparison of Strasheela with existing music constraint systems.

8.2. Florid Counterpoint

This example demonstrates Strasheela's capabilities for polyphonic CSPs where both the pitch structure as well as the rhythmical structure is constrained by rules. Previous systems either hardly support polyphonic CSPs at all or require that the temporal structure is determined in the problem definition (cf. score-PMC, Sec. 3.3.1.2).

This example was designed to be relatively simple. Therefore, it compiles rules from various sources instead of following a specific author closely (as did the previous example). For example, some rules are variants of Fuxian rules introduced before, but rhythmical rules were inspired by Motte [1981]. Accordingly, the result does also not imitate a particular historical style (but neither does Fux, cf. Jeppesen [1930]).

8.2.1. The Music Theory

This example creates a two-voice counterpoint like the previous example in Sec. 8.1. The music representations of both examples are hence very similar. The representation consists of two parallel voices ($voice_1$ and $voice_2$). These voices are represented by two sequential containers nested in a simultaneous container – as before. In this specific example, $voice_1$ contains 17 notes and $voice_2$ 15 notes.

The start time and end time of each voice is further restricted. $voice_1$ begins one bar before $voice_2$. This is expressed by setting the offset time of these two sequential containers (contained in a simultaneous container) to different values (the offset of $voice_1$ is 0, and the offset of $voice_2$ is a semibreve, i.e. 16 as the temporal unit is $beats(4)$, see Sec. 5.4.1.1). Both voices end at the same time (the end time of both sequential containers is unified).

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

The example defines rules for various aspects of the music. The example applies rhythmic rules, melodic rules, harmonic rules, voice-leading rules and rules concerning the formal structure.

- Rhythmical rules
  - Each voice starts and ends with a minim note value.
  - Note durations may only change slowly across a voice: neighbouring note values are either of equal length or differ by 50 percent at maximum (e.g. a quaver can be followed by a crotchet, but not by a minim).
  - The last note of each voice must start with a full bar.
- Melodic rules

  - Each note pitch is restricted to the diatonic pitches of the C-major scale.
  - The first and last note of $voice_1$ must start and end with the root c.
  - The melodic intervals between neighbouring pitches in a voice are limited to a minor third at maximum (i.e. the melodic intervals are more restricted here than in the Fuxian example).
  - An important rule constrains melodic peaks: the maximum and minimum pitch in a phrase occurs exactly once and it is not the first or last note of the phrase. In this example, a phrase is defined simply as half a melody. Finally, the pitch maxima and minima of phrases must differ. This rule concerning the melodic contour – inspired by Schoenberg – has great influence on the musical quality of the result (subjectively evaluated) but also on the combinatorial complexity of the CSP.

- Harmonic rules
  - Simultaneous notes must be consonant.
  - The only exception permitted here are passing tones, where $note_1$ is a passing tone (i.e. the intervals between the note and its predecessor as well as between the note and its successor are steps and both steps occur in the same direction) where a simultaneous $note_2$ started before $note_1$, and this $note_2$ is consonant with the predecessor of $note_1$. Because the rhythmical structure of the result is undetermined in the problem definition, the context of simultaneous notes cannot be accessed directly and is therefore constrained by logical connectives (see Sec. 6.3.4).

- Voice-leading rules
  - Open parallel fifths and octaves are not allowed. However, hidden parallels are unaffected here – in contrast to the Fuxian example.

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

8.2.2. Search Process and Results

Like the corresponding section for the Fuxian example, this section compares the performance of different distribution strategies for the example and finally presents a musical example of the output.

In contrast to the previous example, this example constrains a score context which cannot be accessed in the problem definition. The problem definition does not provide enough information to access simultaneous notes in the music representation. Nevertheless, the

relation between simultaneous notes is constrained by the harmonic and voice-leading rules.

A distribution strategy is therefore required which determines this score context at an early stage to support propagation of the constraints of rules applied to this context (Sec. 7.3.1.1). The left-to-right score distribution strategy (Sec. 7.3.2.3) was applied. This distribution strategy found the first solution in about 4 seconds (189 distributable spaces, 175 failed spaces, search tree depth 47).

For comparison, a distribution strategy which first fully determines the temporal structure and then searches for the pitches (Sec. 7.3.2.2) was applied as well. To determine the temporal structure, the distribution only needs to determine the note duration values. The note offset time values are already determined in the problem definition. Start time and end time values are determined by propagation once the durations are known. This distribution strategy did not find any solution within an hour (i.e. not even within about 900 times more time than needed by the left-to-right strategy)! After that time period, the search process was interrupted.[5]

This distribution strategy finds solutions for the rhythmical structure which indeed fulfils all rhythmic rules but are in conflict with rules concerning the pitch structure. However, these conflicts are detected very late which causes much redundant work.

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

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

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

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

[5] For this CSP, the first-fail distribution strategy (Sec. 7.3.2.1) is equivalent to a distribution strategy which first fully determines temporal structure. In this CSP, the duration parameters have the smallest domain (only three duration domain values vs. at least 14 pitch domain values) and are therefore determined first by first fail.

Figure 8.15 shows one solution for this CSP. $voice_1$ is depicted as the lower voice. Again, a randomised value ordering has been used (whose search process was comparably efficient to the mid-domain-value ordering used for the measurements).

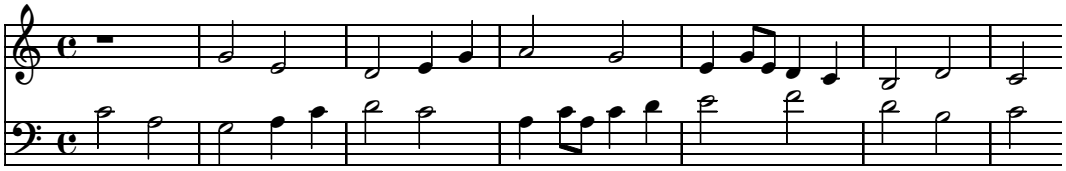


Figure 8.15.: One solution of the florid counterpoint CSP (using random value ordering)

In conclusion, it should be mentioned again that previous systems such as PWConstraints and Situation do not support musical CSPs like the one presented in this section due to their computational complexity (cf. Sec. 4.1.3). The left-to-right distribution strategy, proposed by the present research, solves problems like this in a few seconds and thus in a reasonable amount of time for practical use.

8.3. Constraining the Shape of the Temporal Hierarchy

8.3.1. Motivation

The temporal hierarchy (Sec. 5.4.4) expresses fundamental aspects of the musical form. For instance, in the counterpoint example of Sec. 8.1 the temporal hierarchy specifies the number of notes per voice and the number of voices in the score.

The user may want to somehow constrain this hierarchic structure as well. For instance, it may be required that the number of notes in a voice depends on other factors in the CSP such that it is only found during search. For example, a CSP may constrain so that the first and the last note of a voice starts with a new measure (cf. Sec. 8.2). The CSP may apply additional rhythmic rules which are in conflict with this condition and the fixed number of notes given to the voice (i.e. the CSP would require that either a few notes are added or removed to fit the melody into the measure boundaries).

In non-deterministic logic programming (i.e. the programming model of pure Prolog [Bratko, 2001]), a variable can reference a composite data structure which can contain any other data including other variables. For instance, a non-deterministic version of the list predicate $append(Xs, Ys, Zs)$ constrains three list variables (which can contain any data) and implicitly constraints the length of each list. Using this programming model makes it easy to constrain the shape of an hierarchic data structure. However, this model also easily leads to highly inefficient programs because this model does not support constraint propagation.

Constraint propagation greatly improves efficiency, but gives rise to domain specific variables (Sec. 7.2.3.1). These variables cannot reference a composite data structure with arbitrary data. Instead, the relation between such variables is controlled by constraints

which are defined for specific domains (e.g. a propagator expressing that some finite domain integer is a member of some finite set). The shape of a score consisting of nested class instances (i.e. score objects) cannot be constrained directly using propagation and domain specific variables: the constraint system (variables plus propagators) for the domain 'sequence of class instances' would be missing.

8.3.2. Approach

The present example proposes an alternative approach to constrain the shape of a score hierarchy which is based on domain specific variables and propagation. This approach is consistent with the Strasheela model as introduced in the Chap. 5 to Chap. 7. For this purpose, the example labels score objects with an additional 'existence attribute' represented by a variable. When the variable has a certain value (e.g. 0), the whole score object is regarded as non-existent. Such a 'non-existent' score object is only contained in the internal score, but omitted when the score is transformed to any output format.

For temporal items – which form the temporal hierarchy – an extra 'existence attribute' is not required. Instead, the duration parameter can also be interpreted as an 'existence attribute'. This example proposes that any temporal item is 'non-existent' if its duration equals 0 (regardless of the temporal unit of measurement). The constraint *isExisting* (Fig. 8.16) encapsulates this concept.

$$isExisting(myItem) \stackrel{def}{=} getDuration(myItem) \neq 0$$

Figure 8.16.: Constraint to express whether a temporal item exists

The notion of a 'non-existent' item is not limited to single events with the duration 0. A temporal container with duration 0 is also considered non-existent. If the duration of a temporal container equals 0, all its contained items are also constrained to this duration (Sec. 5.4.4.1), which means the contained items are considered non-existent as well. In an extreme case, even the top-level container can be of duration 0, meaning that the whole score tree is 'non-existent' or 'empty'.

8.3.3. Elimination of Symmetries

By introducing the notion of 'non-existent' items, multiple solutions become equivalent. For example, the only relevant parameter of a 'non-existent' item is its duration. All other parameters of such an item (such as a note's pitch) are irrelevant, and multiple solutions which differ only in the pitch of a 'non-existent' note are equivalent. Similarly, in a container with multiple items only the number of 'non-existent' items matters, but their position is irrelevant.

Multiple equivalent solutions are called *symmetries* [Schulte and Smolka, 2004]. Eliminating such symmetries can drastically reduce the search space. In the following, two constraints are defined to eliminate the two symmetries mentioned.

In case the constraint *isExisting* (Fig. 8.16) returns *false* for some item, then all parameters of this item must be determined to some fixed value (e.g. to their median domain value). That way, there is only a single solution for this item. The only exceptions are the parameters *startTime* and *endTime* which do not introduce any symmetries: once the *offsetTime* of an item is determined and its *duration* equals 0, then propagation deduces $startTime_{myItem} = endTime_{myItem}$ and both parameters only depend on the temporal position of other items in the score. For instance, if the end time of some item in a sequential container becomes determined and the offset time of the following item is determined as well, then the start time of this following item becomes determined by propagation alone. Figure 8.17 defines this first symmetry-elimination constraint: $eliminateSymmetries_1$ must be applied to all temporal items in the CSP. The constraint *determineVariables* determines all parameter values of *myItem* (except for its start time and end time).

$$\begin{aligned} &eliminateSymmetries_1(myItem) \stackrel{def}{=} \\ &\quad \neg isExisting(myItem) \Rightarrow determineVariables(myItem) \end{aligned}$$

Figure 8.17.: Eliminate symmetries I: determine all parameters of a 'non-existent' temporal item

To eliminate the symmetries caused by the position of 'non-existent' items in a container, the next constraint restricts all 'non-existent' items to the end of the item sequence stored in a container. That way, there is only a single solution for 'non-existent' items in a container. Figure 8.18 defines this constraint. $eliminateSymmetries_2$ must be applied to all temporal containers in the CSP (the function *map2Neighbours* was defined in Fig. 6.4, *mapItems* was introduced in Fig. 5.23).

$$\begin{aligned} &\textbf{let}\ \ zerosOnlyAtEnd(xs) \stackrel{def}{=} map2Neighbours(xs, f : f(X,Y) \stackrel{def}{=} X = 0 \Rightarrow Y = 0) \\ &\textbf{in}\ \ \ eliminateSymmetries_2(myContainer) \stackrel{def}{=} \\ &\qquad zerosOnlyAtEnd(mapItems(myContainer, getDuration)) \end{aligned}$$

Figure 8.18.: Eliminate symmetries II: 'non-existent' temporal items occur only at the end in a container

The effect of the constraint $eliminateSymmetries_2$ also allows for the definition of a rule *relevantLength* which constrains the number of items in a temporal container (Fig. 8.19). The number of 'existing' items in a container is the position of the last existing item (i.e. the item before an item with duration value 0). The definition uses the function *mapInd*,

which is a variant of the function *map*: *madInd* applies a binary function with the index of a list element as its first argument.

$$\begin{aligned}
&\textbf{let}\ \ aux(xs, N) \stackrel{def}{=} \bigwedge mapInd(xs, f : f(i, X) \stackrel{def}{=} X > 0 \Leftrightarrow N \geq i) \\
&\qquad\qquad\qquad\qquad \wedge N \leq length(xs) \\
&\textbf{in}\ \ \ relevantLength(myContainer, N) \stackrel{def}{=} \\
&\qquad aux(mapItems(myContainer, getDuration), N)
\end{aligned}$$

Figure 8.19.: The rule *relevantLength* constrains the number of 'existing' items in a *myContainer* to N

8.3.4. Discussion

The approach proposed here – the interpretation of the duration parameter as an 'existence attribute' – allows the user to constrain the shape of the temporal hierarchy. This approach makes the definition of new musical CSPs possible. For example, the user can constrain the rhythmical structure in a contrapuntal CSP as detailed above. The user can also encapsulate a musical segment such as a motif or a transitional sequence without specifying its exact length (e.g. encapsulated by a function which creates the segment and applies some constraints to it). An example of such a segment is a sequential container with notes where the intervals between the notes are constrained to form a sequence (cf. Fig. 8.20) – without exactly specifying the actual number – or the actual pitches – of the notes. This container could then be inserted in a score wherever such a phrase is required (e.g. as a transition) and additional constraints could further restrict the number of notes, for example, to fit into some metric structure and the pitches, e.g., to fit into some harmony.

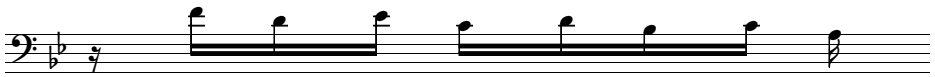


Figure 8.20.: The required length of a sequence (here falling thirds) may depend on other constraints (e.g. the metrical structure)

Nevertheless, the generality of the proposed approach to constrain the temporal hierarchy is restricted. The actual shape of the temporal hierarchy cannot dramatically change when comparing different solutions, subtrees can only either 'exist' or 'not exist'. Because the actual score topology formed by temporal items can only change within certain bounds, also the musical texture (e.g. whether a solution is homophonic or polyphonic) cannot change much. In other words, constraints on the musical texture are limited.

A simultaneous container (Sec. 5.4.4) can be used like an event-list where contained items can freely overlap temporally – thanks to the offset parameter of all its contained items. This allows the user to express various complex temporal structures (e.g. piano music,

where the textures is continuously changing) within a single simultaneous container. In this case however, the access of score contexts (e.g. melody vs. accompaniment) requires additional work (e.g. by marking of all members of a context by special info-tags).

Constraining the temporal hierarchy makes the whole CSP more complex. The definition of every rule must take 'non-existing' events into account. For example, a rule constraining the last note of a voice (e.g. the rule constraining the last interval to a perfect consonance in counterpoint example in Sec. 8.1) cannot be applied directly, because the score context 'last note of the voice' is inaccessible in the problem definition (Sec. 6.3).

9. The Strasheela Prototype

This chapter briefly introduces a software prototype of the Strasheela model proposed by this text and states the differences between this prototype and the model. The prototype can be downloaded at [Anders, 2006].

Section 9.1 surveys how the prototype extends the model described in this thesis both in general (e.g. by providing a number of output formats including MIDI, Csound, and Lilypond) as well as for specific CSPs (e.g. for harmonic or motivic CSPs). Section 9.2 discusses the user interface programming language of the prototype, how this language relates to the mathematical notation used in this thesis, and the requirements of alternative languages for later implementations. Section 9.3 explains the limitations of the prototype.

9.1. Relation to the Model Presented

The present research provides the software `Strasheela`, a full implementation of the generic music constraint system presented in this text. The prototype implementation supports every concept detailed in the preceding chapters (e.g. the full music representation, the rule formalism, the space-based constrained model and its customisation for musical CSPs). In addition, the prototype exceeds the model described in this text in a number of ways.

9.1.1. Input and Output

The prototype features the ability to export its internal music representation to other representation formats by score interface functions. Particularly interesting exchange options include music notation and sound synthesis software formats as well as the format of other computer-aided composition environments. Strasheela presently exports its internal music representation to the music notation software Lilypond [Nienhuys and Nieuwenhuizen, 2003], MIDI files, score files of the sound synthesis languages Csound [Boulanger, 2000], and Common Lisp Music (CLM) [Schottstaedt], as well as to the music representation of the composition system Common Music [Taube, 2004]. Also, the textual Strasheela music representation (Sec. 5.4.3.2) can be exported for archival and editing purposes and imported again. Further formats, as well as support for the import from various formats into Strasheela's internal format, can be added.

The prototype usually runs on its own, but it can also be integrated into existing composition systems. The prototype runs either in an Emacs-based interactive development environment (IDE) [Kornstaedt and Duchier, 2004] or as a server in the background which is fed code via Internet sockets. For example, the Strasheela server can be started and fully 'remote controlled' by a Common Lisp system – presently the lingua franca for computer-aided composition – such as Common Music [Taube, 1997], PWGL [Laurson and Kuuskankare, 2002], or OpenMusic [Assayag et al., 1997].

9.1.2. Extended Strasheela Core Music Representation

The prototype extends the music representation discussed in this text. For instance, the prototype extends the interface of score objects. Some methods introduced in this text are more general in the prototype, and many additional methods are provided. For example, in the prototype the interface for an instance of the class *note* consists of more than 100 methods. The present text introduced only the essence of this interface. Several methods in the prototype are more expressive than what was presented above. For instance, several higher-order methods such as *map*, *collect* or *filter* (Sec. 5.5.2.2) support additional arguments which allow the user to specify to which depth the score hierarchy is traversed and whether a subtree or the whole graph is traversed.

9.1.3. Extensions for Specific CSP Classes

The design of the Strasheela music representation core aims to be style-neutral. For instance, virtually any music requires the representation of time and of temporal relations between objects in the score. Therefore, the Strasheela representation core supports these aspects (the Strasheela model introduced so far is called *Strasheela core* in this chapter).

Independent extensions, on the other hand, augment the representation core by various style-specific representation details which facilitate the definition of specific musical CSPs. For example, the representation core features only a style-neutral pitch representation which consists of a single value. In the harmony-model extension (Sec. 9.1.3.2), this pitch representation is augmented into a composite representation consisting of pitch class, octave and accidental.

Strasheela has been designed for solving complex musical CSPs. Complex CSPs are defined more easily in a modular way, for example, by using relatively generic extensions such as those introduced in this section. Existing music constraint systems such as PWConstraints (Sec. 3.3.1) and Situation (Sec. 3.3.2), on the other hand, do not support user-extensions of their music representation (such as the harmony model or the motif model presented below). The following paragraphs outline some extensions provided by the Strasheela prototype.

9.1.3.1. Pattern Rules

Patterns play an important role in music composition. Therefore, composition systems such as Common Music [Taube, 2004] define a rich set of patterns. Also, patterns play an important role in music analysis. For instance, Conklin and Witten [1995] propose a formal model for patterns in various explicit and derived musical features (e.g. the sequence of note pitches in a melody, the intervals between these pitches, and the directions of these intervals).

Situation introduces the notion of pattern rules (constraints). Example pattern rules include a voice profile constraining the number of consecutive upward and downward movements, or a rule constraining the melodic intervals of a voice (not) to use a specified set of intervals. As these patterns are enforced by constraints, they can be combined with other constraints. Also PWConstraints proposes a technique which shows some similarities with pattern rules. PWConstraints users can define a contour by an envelope (break point function) and constrain a pitch succession to approximately follow this envelope.

The Strasheela prototype proposes a simple and generic pattern formalism which allows users to easily define pattern constraints. A pattern rule is formally a procedure (a function according to the rule formalism of this thesis, Chap. 6) which constrains a sequence of variables. A pattern rule is applied by accessing any sequence of variables from the CSP and applying the constraint to this sequence. For example, a pattern rule can be applied to the list of the note pitches of some voice, the intervals between these pitches, or the roots of a harmonic progression.

About fifty pattern rules are already predefined in the prototype. Examples include generalisations of common binary or ternary constraints for sequences (e.g. *min*, *max*, *increasing*, *decreasing*), constraints inspired by Common Music patterns (e.g. *cycle*, *palindrome*, *markovChain*), constraints on the contour, and constraints on sequences of boolean variables constrained by reified constraints (e.g. *oneTrue*, *someTrue*, *allTrue*).

9.1.3.2. Constrainable Harmony Model

The Strasheela prototype provides a harmony model. This model refines the pitch representation of Strasheela's core representation. The refined pitch representation introduces notions such a pitch class, octave, degree, and accidental and defines their relations by constraints (e.g. the constraint $pitchClassToPitch(PitchClass, Octave, Pitch)$). This pitch representation is highly generic. It supports the pitch class concept of set theory [Forte, 1973], the enharmonic spelling of western tonal music, and also microtonal music (e.g. pitch classes and accidentals can be specified in cents instead of semitones).

The model implements the harmony concepts interval, scale, and chord by music representation classes, and extends Strasheela's standard note class. These classes employ a refined pitch representation. For example, a note features the additional parameters pitch class, octave, degree, and accidental (besides the pitch parameter) and implicitly

constrains the relation between these parameters. A chord instance features parameters such as *root* (whose value is a pitch class) and *pitchClasses* (whose value is a pitch class set). The music representation of a harmonic CSP contains instances of these new classes (often together with other score objects such as temporal containers) and restricts their relation by a set of constraints.

The model predefines constructs to greatly simplify the definition of typical harmonic CSPs. The model supports the specification of implicit constraints on the intervals, scales and chords by databases. For example, the user may specify that all chords must be major or minor triads, but that these triads can be transposed freely. Usually, the pitch class of each note must be a member of the pitch class set of its simultaneous chord – except the note meets special conditions (e.g. the note is a passing note). The model predefines this rule – the user only passes the exceptions (e.g. passing note, auxiliary note) as reified rules on a single note.

Naturally, arbitrary further constraints can be applied by the user. For example, the user may demand that neighbouring chords must have common pitch classes, or that the dissonance degree across a progression of several chords should gradually increase.

9.1.3.3. Constrainable Motif Model

Motifs and gestures play an important role in music composition. In many musical styles, motif relations are a vital device for constructing compositional form. So far, there has been little research into how to represent and constrain motifs for music composition. Sandred [2003] introduces the idea of domains of fully pre-composed rhythmic motifs for OMRC and allows the user to apply further constraints on the music (e.g. a motif must fit into a bar structure without syncopation). OMRC itself is limited to purely rhythmical CSPs, however, similar techniques can be used in PMC for other parameters.

The Strasheela prototype proposes a highly generic model to represent and constrain motifs. This model distinguishes unrelated motifs (in music analysis often denoted by letters such as a vs. b, [Schoenberg, 1979]) as well as variations of a single motif (e.g. a^1 vs. a^2). The user freely constrains both 'dimensions'.

A motif is represented by a new container subclass (containing arbitrary items, e.g., a sequence of notes) with two additional parameters – *motifIndex* and *motifConstraint* – whose values are finite domain integers. The first parameter controls the motif identity whereas the second parameter controls the variation.

The user defines the set of possible motif identities and variations in two databases. The parameter *motifIndex* is an index into the motif identity database and the parameter *motifConstraint* is an index into the motif variation database. The motif model applies implicit constraints to each motif instance which are based on its two parameters (*motifIndex* and *motifConstraint*) as well as the content of the two databases (motif identity database and motif variation database).

On the one hand, each motif identity database entry defines arbitrary motif features. For example, the motif database entry in Fig. 9.1 defines absolute note durations and the pitch contour of the motif (i.e. a sequence of pitch interval directions between notes of the motif). A motif database with this entry will usually define further entries which specify different note duration sequences and pitch contours. Specifying the two aspects note durations and pitch contour is only an example, other databases will specify other motif features.

$$motif\,(durations : [2, 2, 2, 8]\ pitchContour : [=, =, -])$$

Figure 9.1.: A possible declaration of the first motif from Beethoven's 5th Symphony

On the other hand, each motif variation database entry (a motif constraint) is a first-class procedure which arbitrarily constrains the motif instance using its motif identity features. For instance, in case the motif identity database specifies note durations and the pitch contour (see above), then the entries of the motif variation database use these specifications when applying constraints to a motif instance. For instance, the first motif variation database entry may constrain the motif's note durations and note pitches to follow the sequence of durations and the pitch contour specified in the motif identity database entry which will be chosen by the search process. The second entry of the motif variation database may instead constrain the motif pitches to follow the inverse of the pitch contour defined by this database entry. The motif identity database entry and the motif variation database entry that is finally used for a specific motif instance is only decided during the search process.

The set of solutions of a single motif thus depends on three user-controlled dimensions: (a) the set of entries in the motif identity database, (b) the set of entries in the motif variation database, and (c) the ambiguity implicit in the motif constraints in the variation database. For example, constraining only the pitch contour allows many motif variants with the same motif index and the same motif constraint.

Each of these dimensions can be further constrained. For instance, in a succession of several motifs the motif index and motif constraint may be constrained to follow some pattern. The pitches of the motif notes constrained to follows some motif contour may additionally be constrained to follow some harmonic progression (Sec. 9.1.3.2).

However, this motif model is even more general. Entries in the motif identity database may contain undetermined variables. For instance, the user may constrain a set of motif score instances to be the same motif (e.g. all motif indices are equal), but the actual shape of the motif may be undetermined in the CSP definition and may depend, for instance, on contrapuntal motif combinations (e.g. the actual shape of a fugue subject and counter-subject may only be found during search).

Furthermore, the number of items in a score motif instance may be constrained (Sec. 8.3). For instance, different entries in the motif database maybe of different length.

To model free, non-motivic sections between motivic sections, a specific motif constraint may not apply any constraints to the motif instance at all. A decision for this motif constraint then means a decision for a non-motivic section.[1]

The formalism proposed here may also serve to define higher-level formal relations. Whereas a plain motif contains notes, a 'higher-level motif' contains motifs. A higher-level motif constrains the sub-motifs in the same way a basic motif constrains notes by deciding upon the parameters motif index and motif constraint and thus deciding for an entry in the motif database and an entry in the motif constraint database. For example, the user may want to constrain the sequence of motifs contained in a higher-level motif by applying a pattern rule (Sec. 9.1.3.1) to sub-motif indices, or by applying a pattern rule to the sequence of the maximum pitches of each motif.

9.2. Programming Language Issues

Strasheela makes use of a programming language as its user interface. Whereas the present text uses mathematical notation for this purpose, the prototype employs the Oz programming language [Smolka, 1995; van Roy and Haridi, 2004] as user interface language (and as implementation language as well).

Oz has been chosen because it meets the requirements of Strasheela very well. Strasheela applies a multi-paradigm programming approach which is founded on three programming paradigms: the space-based constraint model, functional programming, and object-oriented programming (Chap. 5 – 7). Only a few languages offer this combination of programming paradigms. The Oz language natively supports all these paradigms in a coherent whole.

There are a number of differences between the mathematical notation used in this text and Oz notation. Firstly, mathematical notation has solely declarative semantics which serves its purpose as a means to communicate the main ideas of the Strasheela design. Oz, on the other hand, also has clear operational semantics, which is essential for an actual software implementation. Therefore, Oz notation is often more specific. For example, the equality operator of mathematical notation (=) has a number of different counterparts in Oz notation: the boolean predicate (`==`), unification (`=`), the equality constraint (`=:`), and stateful binding (`:=`).

In addition, the Oz programming language supports a number of concepts for which no counterpart was introduced in the mathematical notation of this thesis. For example, this text introduced no mathematical notation for first-class procedures, concurrent threads, nor classes or methods. Still, all these concepts play an important role in the implementation and usage of Strasheela. For example, Chap. 6 introduced compositional rules as (sequential) first-class functions. Instead, rules in the prototype are actually expressed by (possibly concurrent) first-class procedures (cf. Sec. 7.2.3.4).

[1] To eliminate symmetries (i.e. different solutions which are equivalent), this non-motivic motif constraint should determine the motif index as well.

Oz notation has its own syntax which clearly differs from mathematical notation. For example, a function definition in Oz differs clearly from the notation used in this text. Also, similar syntactic constructs often differ in small details. For instance, the notation of a list is similar in both notations, but Oz omits the commas (e.g. $[1, 2, 3]$ vs. `[1 2 3]`).

Although Oz constitutes a highly suitable language for implementing the Strasheela model, another language could have been chosen. In principle, the Strasheela model can be reproduced in any language which supports the programming paradigms required by its design (i.e. support for the space-based constraint model, functional programming and object-oriented programming). For instance, Strasheela can be implemented in a language which supports functional programming and object-oriented programming plus an interface to C++. This interface would then allow the use of the Gecode library [Gecode authors, 2006], which implements the space-based constraint model (interfaces to Gecode are in progress for a number of programming languages including Oz [Avispa Research Group, 2006] and Common Lisp[2]). For example, such an approach makes it possible to implement Strasheela in Common Lisp.

9.3. Limitations of the Prototype

This section discusses limitations of the prototype with respect to its generality. These shortcomings are discussed in the present chapter, because they are shortcomings of the current implementation and not of Strasheela's actual design.

Strasheela also has one principle limitation which restricts its generality compared with existing music constraint systems: Strasheela features domain specific variables (Sec. 7.2.3.1) instead of variables with universal domain. The advantages and disadvantages of this approach are explained in Sec. 10.1.2

Hard-Wired Variable Domains in Implicit Constraints

Section 6.2.8 explained that specific Strasheela objects apply constraints implicitly for convenience when an object is created. The only implicitly applied constraints of the Strasheela core are the temporal constraints affecting the temporal items (Sec. 5.4.4.1). The Strasheela extensions presented in the present chapter – the harmony model and the motif model – apply further constraints implicitly.

In the present prototype implementation, implicit constraints are hard-wired to specific variable domains. For instance, all implicit temporal constraints are hard-wired to finite domain integers.

In principle, the Strasheela user can freely control the domain for every variable in the music representation. For instance, a numeric variable may be either a finite domain integer or a real number from the XRI constraint system [Avispa Research Group, 2005].

[2] Personal communication with Camilo Rueda, 26 July 2006 at IRCAM, Paris.

Variables to which constraints are applied implicitly, however, are restricted to the domain of their implicit constraints. This means that temporal parameters are limited to finite domain integers.

The consequences of this limitation depend on the parameters. In case of the temporal parameters, for example, this limitation makes complex rhythms (e.g. nested tuplets) harder to realise (although setting the temporal parameter attribute *timeUnit* to *beats*(N) where N is reasonably large can still make complex nested tuplets possible, Sec. 5.4.1.1). Moreover, the *offsetTime* of temporal items cannot be negative (e.g. items in a sequential container cannot overlap).

The limitation could be addressed by refactoring the implicit constraints. The refactoring would introduce an abstraction layer which substitutes the domain specific constraints by more general constraints. Which domain specific constraint is called by such a more general constraint would be depend either on a global user setting or on the domain of the variable arguments supplied to the more general constraint.

Random Variable Ordering without Recomputation

Strasheela predefines a number of distribution strategy (Sec. 7.2.3.2) building blocks, including a random value ordering. For musical applications, such a value ordering is highly important.

The present implementation of this value ordering, however, is indeterministic (i.e. repeating a distribution during the search process does not necessarily result in the same decision). This implementation violates a fundamental requirement of a distribution strategy [Schulte, 2002]. Searching using pure copying does work, but recomputation (a technique for saving memory which is important for solving complex CSPs, Sec. 7.2.3.5) is not possible with this value ordering.

The limitation could be addressed by making the value ordering deterministic even if it appears to be random. For example, random numbers could be stored inside the search script with an unambiguous mapping between a node in the search tree and a specific random number. Alternatively, the limitation could be addressed by using the constraint library Gecode (see above). Gecode supports batch recomputation which records all decisions of the distribution strategy (Sec. 7.2.3.5).

Memory Consumption

Strasheela's music representation can store a considerable amount of information. However, this data requires space (i.e. RAM). Only a small part of this data actually changes during the search process. The search process only determines the values of the variables in the representation (the value of parameters such as note durations and pitches) – all other data is static.

The space-based constraint model uses copying (cloning of spaces) to make the search process programmable. In the current implementation, the whole music representation is copied during the search process, although only a small part of this data only changes. For

complex CSPs with deep search trees, this results in considerable memory requirements. Recomputation makes it still possible to solve arbitrarily complex problems.

A better solution would be to avoid wasting memory in the first place. In an improved version of the Strasheela prototype, the music representation would be stored outside the computational space and only the constrained variables would be stored within the space and would be copied. An abstraction layer would provide for an unambiguous and bidirectional mapping between the parameters in the music representation outside and the variables inside the computational space. This mapping between the music representation and the variables would make it possible to utilise all information provided by the representation and the variables together in order (i) to apply constraints to the variables and (ii) to conduct the score distribution strategies presented in Sec. 7.3.2. Such an approach would drastically reduce the required amount of memory and also slightly improve runtime efficiency by reducing the amount of copying and garbage collection.

10. Comparison and Evaluation

Section 4.2 stated the goal of the present research: this research aims at creating a system which facilitates the definition of musical CSPs (when compared with a general constraint system) and at the same time is highly generic (i.e. allows for the definition of a large set of musical CSPs). A most generic music constraint system would support the implementation of any music theory model conceivable, but such a system would possibly be highly inefficient. For usability, the present research aims at creating a system which is highly generic but still reasonably efficient.

This chapter evaluates the present research. The chapter asks (i) whether Strasheela indeed facilitates the definition of musical CSPs, (ii) whether it is generic and (iii) whether is it reasonably efficient.

The evaluation of the first aspect of this goal can be brief. Experience with existing generic music constraint systems shows that such systems indeed considerably facilitate the definition of musical CSPs. This experience is confirmed by the present research. For instance, the definition of the presented counterpoint examples (Chap. 8) in a general constraint system without any special support for musical CSPs would amount to reproducing considerable parts of Strasheela in such systems. These examples require a music representation which expresses the temporal structure as well as the pitch structure of polyphonic music and which allows the application of compositional rules to a large set of score contexts.

The generality of Strasheela is evaluated by comparing its degree of generality with the generality of existing systems. Only the most important generic music constraint systems are taken into account in this comparison: this chapter compares PMC, score-PMC, Situation, and the combination of MusES and BackTalk with Strasheela in terms of their generality. All these systems have been discussed before, but from a different perspective. Section 3.3 introduced the fundamental idea of each system. Section 4.1 analysed fundamental limitations of these systems with respect to generality before Strasheela was introduced. In contrast, this chapter evaluates the design of Strasheela by comparing it with existing systems. Further systems are left out of the discussion for brevity. Section 3.3.3 already showed that OMClouds is less generic than PWConstraints and Situation. Also the other systems presented in Sec. 3.3.4 are either by order of magnitudes less efficient (CHARM and Arno), mainly theoretical (PiCO), or much less developed (OMBT).

The efficiency of Strasheela is evaluated by examining Strasheela's efficiency for the examples shown previously and by comparing how different systems allow the user to control the search process. Nevertheless, this text does not compare the performance of

existing music constraint systems with Strasheela. The main reason is the claim of the present research that Strasheela allows its user to solve CSPs which are hard or even impossible to solve with existing systems – hence, a comparison is not possible for such CSPs. In addition, there exist no standard musical benchmark CSPs so far.

Chapter Overview

This chapter discusses the three important design aspects of a music constraint system in turn. Section 10.1 compares the music representation of the different systems and examines what information can be represented and constrained in each system. The rule formalism and in particular the rule application mechanisms of different systems are investigated in Sec. 10.2. Section 10.3 compares how the search can be adopted to specific musical CSPs in these systems. Section 10.4 then asks how Strasheela could be further generalised. A brief conclusion summarises this chapter (Sec. 10.5).

10.1. Music Representation

It can be argued that research into music representation and research into music constraint systems have only marginally influenced each other so far. On the one hand, few genuine music representations incorporate the notion of variables to represent partial information and to express relations between unknowns. Examples for such representations include Music Structures [Balaban, 1996] and the Prolog implementation of CHARM [Harris et al., 1991]. These representations, however, introduce variables mainly as a theoretical device, for example, to express that some information can be missing [Balaban, 1996]. The authors do not state actual CSPs which require search for solving.

Music constraint systems, on the other hand, have made little use so far of the rich means developed by music representation research to represent information about music. For example, music representation research identified hierarchic representations as an important means to express score information (cf. Sec. 2.4). However, the following systems represent music only by a sequence of objects: PMC (Sec. 3.3.1.1, [Laurson, 1996]), Situation (Sec. 3.3.2, [Rueda et al., 1998]), and OMClouds (Sec. 3.3.3, [Truchet et al., 2003]). Similarly, music representation research identified the data abstraction concept as an expressive means, for instance, of allowing for complex derived score information (cf. Sec. 2.6). Still, only a few music constraint systems provide any data abstraction layer.

The present research proposes an integration of the notion of constrained variables (solved by search) and several established music representation principles. Doing so results in both a more expressive general music representation as well as a more expressive music constraint system (which makes use of this representation).

10.1.1. Representation Format

The definition of a musical CSP makes use of various pieces of musical information. There is the information which expresses the actual solution (e.g. a number of notes with their parameters start time, duration and pitch), and there is the information required to express the score contexts affected by a compositional rule (cf. Chap. 4). The designer of a generic music constraint system cannot foresee the CSPs to which the system will be applied. It is therefore not possible to foresee which information – and in particular which score contexts – will be required from the music representation. For a generic system, it is therefore crucial that the user is able to define which information is represented and which score contexts can be accessed. This section compares Strasheela to existing music constraint systems concerning these requirements.

10.1.1.1. Strasheela

Strasheela supports several basic principles to store score information. These principles were explained in Sec. 5.6.1 and are only briefly summarised here. Explicitly stored pieces of information are the type of each score object, its attributes, the nesting of score objects, and the order of score objects at the same nesting level. Derived information can be deduced by utilising any piece of explicit information or of other derived information about score objects. Any score object provides access to all information in the score due to bidirectional links between all objects. Additional principles make the representation more convenient to use and extend: Strasheela score objects are abstract data types which are defined incrementally by class inheritance. The textual representation facilitates the specification of score topologies.

10.1.1.2. Comparison

The fundamental principles of explicitly stated and derived musical information (see above) are also supported by other music constraint systems, but these systems support only a subset of these principles or support them in a less general way. For example, existing systems store less explicit information and limit access to derived information.

As a first overview, Tab. 10.1 compares how different systems support the principles of explicitly stated information. Explicitly represented nesting and order are subsumed in the score topology column. The table distinguishes between 'user defined' and 'user definable', where 'user defined' means that there is no default setting for this principle (e.g. Situation does not predefine any score object attributes) and 'user definable' means that the user can extend what a system predefines (e.g. the MusES user can extend the predefined attributes of a note). Each system is discussed in more detail below.

Table 10.2 continues this overview for derived information. The table shows which score contexts different systems can access in order to deduce information from these contexts. Please note that this table implicitly shows which score contexts can be constrained. The

	Attributes	Score Topology	Type
PMC	none	flat sequence	n/a
score-PMC	fixed	tree (fixed nesting)	fixed
Situation	user defined	flat sequence	n/a
MusES and BackTalk	user definable	one or more temporal sequences[a]	user definable, hierarchical
Strasheela	user definable	user defined[b]	user definable, hierarchical

[a] Each container (`TemporalCollection`) stores temporal objects (e.g. notes, chords) sorted by their start time and duration, i.e. no explicit non-temporal order is represented.

[b] Multiple topologies supported (e.g. nested event lists, tree of generic temporal containers, acyclic graph of arbitrary containers, see Sec. 5.4.3.3).

Table 10.1.: The explicitly represented information supported by the music representation of various music constraint systems. The information represented by variables is not taken into account. See Sec. 10.1.2 below.

supported derived information is also explained in more detail below in the discussion of each system.

	Derived Information Accessible About
PMC	positionally related variables (only via rule application mechanism)
score-PMC	fixed set of predefined contexts
Situation	(i) attributes of a single object,[a] (ii) positionally related objects (only via rule application mechanism)
MusES and BackTalk	any set of temporally related score objects
Strasheela	any set of score objects

[a] The attributes of Situation objects include explicitly represented distances between attributes (see Sec. 3.3.2.1).

Table 10.2.: The derived information supported by the music representation of various music constraint systems

PMC

The PWConstraints subsystem PMC constitutes primarily a general constraint programming language and does not provide much support for the special needs of musical CSPs. The music representation of PMC consists of a flat sequence of variables and supports neither score objects, attributes, nor hierarchic nesting of objects. However, the domain of a variable can consist of arbitrary values – including composite data. This feature allows for a hierarchically nested score section as a solution for a single variable (Sec. 10.1.2 discusses this aspect in more detail).

An object in the music representation (i.e. a variable in the sequence of variables) only knows its own value and has no further information about the partial solution. For example, a variable does not know its position in the sequence nor the values of its neighbours. Only the rule application mechanism has access to the list of all already-determined variables and allows the user to access the position of the current variable and its predecessors.

Consequently, derived information can only deduce information from positionally related variables which are already determined. For example, it is possible to access the current variable and its predecessor (e.g. in order to constrain the distance between these variables). However, no information about variables after the current variable (i.e. about undetermined variables) is accessible.

score-PMC

In contrast to the more general PMC, the seminal PWConstraints subsystem score-PMC (Sec. 3.3.1.2, [Laurson, 1996]) specifically addresses musical CSPs, in particular polyphonic CSPs. The system introduces special compound score data types (e.g. note, beat, harmonic slice) which form a hierarchically nested score. Thus, score-PMC supports all four principles of explicitly represented information (object attributes, hierarchic nesting, order, and type information).

However, the user can hardly adapt this representation. Score data types have a fixed set of attributes, the user cannot introduce new types, and the score topology is fixed. The user is limited to minor adaptations such as specifying the size of the score (e.g. the number of notes in a part). The system score-PMC supports more derived information than most other music constraint systems, yet the set of supported score contexts is also fixed and cannot be extended by the user.

Situation

Situation constitutes a highly generic music constraint system. When compared with score-PMC, the representation of Situation (Sec. 3.3.2.1) is kept more abstract for generality. Whereas score-PMC fixes its music representation by modelling distinct musical concepts (e.g. note, beat, measure, part), the Situation user defines the attributes for a Situation object and freely interprets what an object and its attributes stand for (e.g. an event with a set of parameters or a chord with a number of pitches).

Situation facilitates the definition of a music representation by a template-like specification. The solver (which implicitly creates the music representation) features several arguments by which the user controls the music representation creation: for example, the user specifies the total number of objects, the number of attributes for each object (specified as points and distances, see Sec. 3.3.2.1), the domain for each attribute, etc.

However, Situation's music representation limits what information can be stored in the score. For example, the score topology of Situation is limited to a sequence of objects which feature a number of attributes. However, Situation does not support hierarchic

nesting (e.g. note x is contained voice *alto*). Also, a Situation object lacks explicit type information. It is therefore hard to define CSPs which relate objects of different types where each type features its own set of attributes and possibly even implicit constraints (e.g. a note object and a voice object type, where the note type features the attributes start time, duration and pitch, and a voice object type features the attributes start time, duration, *no* attribute pitch, but implicitly constrains its contained notes to follow each other sequentially in time).

The Situation user can interpret the meaning of objects and their attributes at will, but Situation does not assist any interpretation with specific data abstractions (i.e. there are no accessors such as *getPitch* or *getDuration*). Moreover, Situation does not arrange for user-defined data abstractions to ease the definition of complex CSPs (e.g. predefined Situation constraints cannot be used within user-defined data abstractions).

Moreover, Situation objects only know their attributes. For instance, positional information is not accessible in music representation. This information is only provided for Situation's rule application mechanism (cf. PMC above).

Finally, Situation provides only limited means to access derived information. For example, Situation's rule application mechanism for user-defined constraints provides means to access already determined score objects which allows the user to freely derive information from their attribute values. Like in PMC, however, no information on undetermined objects is accessible. Also, Situation does not predefine any accessors for specific score contexts. For example, Situation lacks the polyphonic contexts accessible in score-PMC. In Situation, only positionally related score contexts (e.g. neighbouring object pairs in the sequence) are supported by Situation's rule application mechanism (Sec. 3.3.2.2 and Sec. 6.2.5).

MusES and BackTalk

The music representation MusES together with the constraint solver BackTalk forms a highly expressive music constraint system (Sec. 3.3.4). MusES supports most of the previously mentioned four principles of explicitly represented information. MusES score objects feature a type, a set of attributes, and allow objects to be nested (e.g. temporal objects such as notes can be contained in a temporal collection). Moreover, MusES is also highly extendable. MusES consists of a set of Smalltalk classes and that way implements a set of incrementally defined abstract data types which allows for easy extension by the user.

Some representational aspects are more convenient in MusES than in Strasheela: MusES inherited the reflective capabilities of its implementation language Smalltalk. For example, for any given object instance the system provides access to the set of its attributes or to its superclass (for security reasons, Oz requires the programmer to implement access to such information explicitly). MusES also features a particularly convenient notation of some concepts: MusES represents many musical concepts by special classes in place of instances of more generic classes. For example, MusES features a special class for each pitch class and for each interval. For instance, MusES represents the pitch class $c\sharp$

as well as the interval "augmented fourth" by their own classes with clear class names. Together with the Smalltalk syntax, this design results in a highly legible pitch class algebra notation.

Yet, MusES is less expressive than Strasheela's music representation in two respects. MusES supports hierarchic nesting only in a limited way and its pitch representation is restricted to western tonal music.

The limitation concerning hierarchic nesting is related to MusES' excellent support for temporal information. Every temporal object in MusES (e.g. a playable note or chord) stores its start time and duration. Temporal objects can be grouped in a temporal collection (a container). A temporal collection ensures that its contained objects are always sorted according to their start time and duration [Pachet et al., 1996]. This design allows for an easy and efficient access to, for example, all objects in a temporal collection within a certain time span. Based on this ability, MusES defines an exhaustive set of temporal relations between temporal objects.

However, the fact that temporal collections implicitly sort their contained objects restricts the expressiveness of these collections. Firstly, temporal collections cannot be nested in MusES (it would disturb the automatic temporal ordering). Secondly, temporal collections always order their contained objects temporally. However, not all ordering information required in a score is of a temporal nature (e.g. the order of parts in a score is non-temporal).

Pachet [1994] motivates the design of MusES by comparing it with MODE (i.e. with MODE's music representation, called SmOKe [Pope, 1992]). SmOKe supports nested event lists (a SmOKe event list is also an event, see Sec. 2.4.3) and the events in an event list can be arranged in a non-temporal order. However, a SmOKe event does not provide access to its start time: this information is only known by its event list. This representation scheme is sufficient for the sound synthesis purposes of SmOKe, but proved unsatisfactory for the analytical purposes of MusES which required access to more information. This led to the different design of MusES.

Strasheela's music representation combines the benefits of both SmOKe and MusES. Like SmOKe, Strasheela supports hierarchic nesting and any ordering of score objects (including a non-temporal order). Beyond this, Strasheela provides access to any information available within a score object (e.g. the start time of an event). This augmented expressivity is obtained by two design principles: bidirectional links and constraint propagation. Strasheela allows for hierarchic nesting of containers (SmOKe's score topology is only a special case of the topologies supported by Strasheela). However, the hierarchically nested score objects in Strasheela are bidirectionally linked: every container can access all information stored in its contained items and vice versa. Furthermore, parameters exchange knowledge about their values (which are constrained variables), due to constraint propagation. For instance, information about temporal parameters such as start-time, duration and end-time is propagated between temporal containers and events (Sec. 5.4.4).

Based on this design, MusES' exhaustive set of temporal relations can be reproduced in Strasheela (either as plain accessors or as constraints), as was demonstrated for the constraint *isSimultaneous* (Fig. 5.32) and the constraint *getSimultaneousItems* (Fig. 5.31). Their implementation can be less efficient in Strasheela than in MusES, but it is still sufficiently fast. For example, accessing specific score contexts may require a traversal of the full score, but usually a score context must be accessed only once (e.g. objects are accessed once by filtering to apply a constraint, but the search process does not need to filter these objects again).

An important contribution of MusES is its support of enharmonic spelling [Pachet, 1993]. MusES defines an algebra of pitch classes (e.g. the pitch class c raised by an augmented fourth results in $f\sharp$ which differs from $g\flat$). However, this spelling is hard-coded into system and alternative pitch representations are not supported. For example, MusES cannot represent music which does not fit into western enharmonic spelling such as music in the gamelan pelog scale or music using Partch's 43-tone scale [Partch, 1974].

The Strasheela user, however, can choose a pitch representation according to their needs. The core music representation of Strasheela introduces a single-value pitch representation which can be measured in various units such as key-number, cent, or Hz (Sec. 5.4.1.1). It can thus express conventional western music as well as music using alternative tunings, although this core representation does not support enharmonic spelling.

This single-valued representation can be extended by the user. For example, the harmony model defined by the Strasheela prototype complements this representation by composite pitch representations (see Sec. 9.1.3.2). This model defines a pitch representation which consists of a degree, an accidental and an octave and which thus allows for the expression of enharmonic spelling. Moreover, this extended representation is not limited to the western set of pitch classes. For instance, in this representation a degree can denote a pitch class measured in cents. The multiple components of the pitch representations (e.g. the single-valued pitch of Strasheela's core representation as well as the degree, accidental and octave) are represented by individual constrained variables, the relationship between them is maintained by constraints, and the user can apply further constraints to each of the components independently.

10.1.2. Constrainable Aspects

System Comparison

Music constraint systems differ not only in their music representation format, but also in which aspects of this representation can be constrained. In this regard, two factors must be taken into account. Systems differ in (i) what values a variable domain can consist of and (ii) where a variable can occur in the music representation. For example, each score-PMC variable is (i) an integer and (ii) denotes a pitch (a key-number).

The systems PMC and MusES plus Backtalk support variables with universal domains, that is, their domain can consist of arbitrary values. The domain of a PMC variable

consists of arbitrary Lisp values and likewise the domain of a Backtalk variable consists of arbitrary Smalltalk objects.

Situation and Strasheela, on the other hand, support constraint propagation and therefore CSPs in these systems usually only make use of variable domains which are supported by their constraint propagation algorithms. The actual domains supported depend on the system. Situation supports numbers (e.g. a mix of integers, floats and ratios) whereas Strasheela supports, for instance, finite domain integers and finite sets of integers (cf. Sec. 7.2.3.1).

Situation and Strasheela both support variables with universal domains as well. Situation comes with an example where variable domains consist of chord objects (i.e. CLOS instances). Similarly, the Strasheela prototype provides a proof-of-concept implementation of a constraint system with universal domain variables.[1] This example is defined in terms of the space-based constraint model and supports the features of this model (e.g. a user-defined distribution strategy is supported) – except propagation. This exception highlights the fundamental drawback of the universal domain. Constraint propagation greatly reduces the size of the search space and thus improves efficiency enormously (Sec. 7.2.3.1). Universal domain variables, however, cannot benefit from constraint propagation, because the propagation algorithms have been optimised for specific domains. The respective advantages and disadvantages of universal domains and specific variable domains are discussed further below.

The variable domains supported by a system and its music representation both influence where a variable can occur in this representation. Naturally, a variable can only substitute a value which corresponds to its domain and the occurrence of a variable is restricted by the format of the music representation. For instance, a Situation variable is a number (when constraint propagation is used) and numbers only occur as an attribute (a point or a distance) of an object within Situation's sequence of objects. Consequently, in Situation a variable can only occur as an attribute of such an object. Table 10.3 lists the respective variable domains supported by the different systems and where variables can occur in the music representation.

Consequences of Specific Variable Domains

Strasheela (i.e. Oz) is restricted to specific variable domains in order to support constraint propagation (see above). This restriction to specific variable domains is the only aspect in which Strasheela is less general than any other generic music constraint system. The restriction is caused by a decision for performance instead of generality. Nowadays, virtually every major general constraint system (e.g. SICStus [SICS], ECLiPSe [Cheadle et al., 2003], GNU Prolog [Diaz, 2003], ILOG Solver [ILOG], Gecode [Gecode authors, 2006]) supports constraint propagation for an efficient search, and consequently supports constraints on variables with specific domains. In the space-based constraint model, the set of natively supported variable domains is not fixed. A number of domains have been

[1] The example can be found in the test directory of the Strasheela release available from `https://sourceforge.net/projects/strasheela/`.

	Variable Domain	Variable Occurrence
PMC	any value	element in sequence
score-PMC	integer	note pitch (key-number)
Situation	number[a]	attribute of object in sequence
MusES and BackTalk	any value	anywhere
Strasheela	specific domains[b]	any value suiting the variable's domain[c]

[a] A domain consists, for example, of integers, floats or ratios (possibly mixed). Alternatively, Situation supports variables of universal domains. However, choosing this option simplifies the music representation into a flat list and disables constraint propagation.

[b] In principle, universal domains are possible. However, constraint propagation is only supported for specific domains (e.g. finite domain integers, and finite sets). The set of supported domains can be extended.

[c] For example, constrained variables can occur as attributes of score objects or locally in rules.

Table 10.3.: Constrainable music representation aspects in various music constraint systems

developed by the Oz community (see Sec. 7.2.3.1) and development is still ongoing. All domains available in Oz are also available in the Strasheela prototype.

It turns out that the restriction to specific variable domains narrows expressivity less than it might first appear. The consequences of this restriction are best understood by distinguishing between two cases. Firstly, a single variable whose domain consists of composite values can often be translated into a composite value which contains multiple variables. For example, a single list variable expressing a list of n integers can be substituted by n integer variables grouped in a list. Secondly, there exist genuine variable domains which cannot be directly substituted. Such a genuine domain often corresponds to some common mathematical domain such as the integer, real, or set domain. For example, a constraint on real numbers (e.g. a logarithm constraint) cannot be adequately substituted by a constraint on integer variables. Likewise, a variable for a set (or list) of integers whose cardinality (or length) is undetermined and constrained in the problem definition cannot be substituted by a set (or list) containing multiple integer variables.

There are variable domains which can be substituted by multiple variables of other domains. This case occurs relatively often in music constraint programming. For example, OMRC [Sandred, 2000a] expresses domains of rhythmic motifs by variables whose domain consist of lists of integers. Likewise, Pachet and Roy [1995] advocate the use variables whose domains consist of composite Smalltalk objects (e.g. note and chord objects) and criticise the harmonisation system by Ballesta [1994] for using multiple variables to express the multiple features of a single note or interval.

Substituting a composite variable with multiple variables not only preserves the same semantics, but can result in a manifestly more efficient search. Finding a value for a variable whose domain consists of instances of composite data boils down to a naïve

generate-and-test search for this variable. In contrast, splitting this composite variable into multiple variables allows the search process to proceed incrementally and that way inconsistent solution candidates are excluded earlier (this advantage is similar to the advantage of backtracking over generate-and-test). Furthermore, multiple variables allow for constraint propagation between these variables. For example, the all-interval series CSP is solved more efficiently when the series is represented by a sequence of multiple variables instead of a single variable whose domain consists of fully determined sequence solution candidates. Also the poor performance of Ballesta's system is not necessarily caused by too many variables and constrains (as criticised by Pachet, see above), but primarily by an inadequate variable ordering (e.g. harmonic decisions on the chord level should be made first, after constraint propagation has reduced the domain for chord variables, cf. Sec. 7.3.1).

A genuine variable domain cannot be directly substituted by variables of other domains. Still, even such genuine domains can often be encoded by variables of other domains. For instance, the example presented in Sec. 8.3 effectively constrains the lengths of lists of items in a container without resorting to a variable with a list domain. Instead, this example introduced the notion of an existence attribute (represented by the duration of a temporal item). Similarly, Dooms et al. [2005a] propose the implementation of a graph domain by boolean variables (0/1-variables) plus logical connective and summation constraints.

10.2. Rule Formalism

A musical CSP consists of variables and constraints on these variables – like any CSP. The meaning of the variables is defined by the music representation (e.g. a certain variable represents the pitch of the first note of a certain voice). Constraints on variables express compositional rules applied to the music representation.

Constraints are applied to variable sets which constitute various score contexts. Section 3.3.1.2 discussed this notion for score-PMC. A constraint (or rule) is either applied directly to a score context (cf. Fig. 6.2) or by some convenient application mechanism which implicitly accesses the score contexts and applies the constraint to it (e.g. the pattern-matching based mechanism of PWConstraints, Sec. 6.2.6).

Direct application requires direct access to the music representation and its variables. This is provided, for example, by the combination of MusES and BackTalk. For instance, with MusES and BackTalk a rule may be applied by an iteration whose running index is used to access neighbouring notes in a melody as follows (cf. [Roy and Pachet, 1997])

$$myConstraint(nth(myMelody, i), nth(myMelody, i + 1))$$

Systems which do not allow direct access to the music representation and its variables (e.g. PWConstraints, Situation, OMClouds) require special rule application mechanisms.

Different systems offer different application mechanisms. For example, Situation offers an index-based mechanism (Sec. 3.3.2.2) and PWConstraints a matching-based mechanism (Sec. 3.3.1.1).

Strasheela provides both direct constraint application as well as convenient rule application mechanisms. The fact that a Strasheela rule is a first-class function[2] forms the foundation for its rule formalism. This allows the user to program complex rule application mechanisms as higher-order functions (Sec. 6.2).

Arbitrary control structures can be defined in terms of higher-order functions, including the rule application mechanisms of previous systems. As a demonstration, the index-based rule application mechanisms of Situation and the pattern-matching based mechanism of PWConstraints have been reproduced in Strasheela (Sec. 6.2.5 and Sec. 6.2.6).

On the other hand, first-class functions can also express rule application mechanisms which cannot be adequately reproduced by the mechanisms of Situation and PWConstraints. The mechanisms of both systems are highly suited to applying a rule to elements in a sequence, because these mechanisms rely on positional relations (such as neighbouring elements). First-class functions, on the other hand, can process arbitrary data structures (e.g. trees or graphs, besides sequences). For example, a rule can be applied to all score objects in a hierarchic score representation for which some test function returns true (Sec. 6.2.7).

Hence, a rule application mechanism based on the notion of higher-order functions is more generic than the mechanisms of Situation and PWConstraints.

Experience with Strasheela shows that a CSP which makes use of suitable higher-order functions is often distinctly more concise than an equivalent CSP which applies constraints directly (e.g. by explicit nested loops). Higher-order functions can abstract away complex control structures which implement mathematical concepts (e.g. the Cartesian product, Fig. 6.16) or knowledge about the music representation (e.g. implicit traversing of the score hierarchy, Sec. 6.2.7).

Because Strasheela's rule application mechanisms allow for a concise and expressive CSP definition, other music constraint systems may be interested in adopting these mechanisms. These rule application mechanisms can be easily reproduced in the combination of MusES and BackTalk. This system is implemented in Smalltalk and Smalltalk provides code blocks, which are essentially first-class functions. Realising Strasheela's rule application mechanisms in systems which don't support direct access to their music representation and its variables (i.e. Situation, PWConstraints and OMClouds) requires internal changes to these systems.

[2] A Strasheela rule is actually a first-class procedure (Sec. 6.1 and Sec. 7.2.3.4).

10.3. Search Strategy

The operational foundation of Strasheela is the space-based constraint programming model (Sec. 7.2) which provides important features for defining and solving CSPs including constraint propagation, user-defined distribution strategies, user-defined exploration strategies, reified constraints and recomputation.

Particularly apparent features are propagation and distribution. Constraint propagation is an essential constraint programming technique used to reduce the size of the search space (cf. Dechter [2003]) and nowadays is supported by virtually all major general constraint programming systems. Constraint distribution allows for user-defined and dynamic variable and value ordering. The choice of a variable ordering has an immense influence on the size of the search space, in particular, if the ordering can exploit knowledge about the structure of the CSP (Sec. 7.3).

The space-based constraint model and its implementation in the Mozart system is competitive in performance with other state-of-the-art constraint systems and superior for large problems. Müller [2001] compared the performance of constraint propagation in Mozart with ILOG Solver, ECLiPSe, SICStus, and GNU Prolog. Similarly, [Schulte, 2002] compared the performance of space-based copying and recomputation with the trailing-based search of these other systems.

Strasheela makes the space-based constraint programming model and its features available for the first time in a generic music constraint system (to the knowledge of the author). The space-based constraint model has been applied before for music composition [Henz et al., 1996], but this research only implemented a very specific CSP (see Sec. 3.2.3). Instead, the present research analysed the particular complexities of musical CSPs in general (such as inaccessible score contexts, Sec. 6.3.1) and identified the need for user-definable dynamic variable orderings. To address this need, the design of Strasheela allows a score distribution strategy to exploit any information available in the score at the time the distribution happens. Because constraint distribution is orthogonal in the space-based constraint model, Strasheela preserves all other features of this model.

The present research proposes special score distribution strategies tailored for musical CSPs which define dynamic variable orderings for typical musical CSPs. The performance figures provided in Sec. 8.2.2 clearly indicate the importance of a well-chosen distribution strategy in solving musical CSPs. For example, the left-to-right strategy (Sec. 7.3.2.3) allows the user to efficiently solve polyphonic CSPs which existing systems (e.g. score-PMC) explicitly made impossible to define due to their computational complexity.

Consequently, the features of Strasheela's constraint solver are only partially (if at all) supported by existing music constraint systems. Constraint propagation is supported in some form by both Situation and BackTalk (but not by PWConstraints). BackTalk supports a number of consistency enforcing techniques (see Sec. 3.3.4), but these are performed only once before the actual search process starts. Situation implements a variant of minimal forward checking for hierarchical domains (see Sec. 3.3.2.3), but its

propagation algorithm is only suitable for its specific data structure (where all distances of an object are represented by a single variable, cf. Sec. 3.3.2.3).

No existing music constraint system allows for user-defined dynamic variable and value ordering as supported by Strasheela (to the knowledge of the author). BackTalk supports dynamic variable orderings. Yet, MusES does not provide the required information to define score distribution strategies as proposed in Sec. 7.3.2.

Some existing systems allow the user to choose a search strategy (i.e. an exploration strategy). For example, BackTalk supports a number of enumeration algorithms. BackTalk even supports *lookback schemes* (e.g. backmarking, [Dechter, 2003]). In contrast, lookback schemes have not yet been proposed for the space-based constraint programming model. Due to the programmable nature of this model, this feature could be provided in principle [Schulte, 2002]. However, BackTalk does not distinguish between the creation (distribution) and the exploration of the search tree, and special exploration strategies such as a best-solution search (which does not need to explore the full search tree to find an optimal solution, Sec. 7.2.3.3) are not provided.

10.4. Can Strasheela be Further Generalised?

Section 5.6.1 discussed the basic principles that Strasheela uses to represent explicit information: the type of a score object, its attributes, the hierarchic nesting of objects, and the order of objects on the same hierarchical level. The value of object attributes is the only explicitly represented information that can be unknown and constrained in Strasheela. The explicit information represented by the other principles must be determined in the CSP definition. An interesting research question is how the other principles can support unknown and constrained information.

Section 8.3 proposed an approach which effectively constrains the hierarchic nesting of objects, although only in a limited form. When comparing different solutions, the actual shape of the hierarchy cannot dramatically change, subtrees either 'exist' or 'do not exist' (see Sec. 8.3.4).

A generalisation of this approach allows the user to freely constrain the score topology. Such a generalisation makes it possible to freely constrain the musical texture (e.g. whether a solution is homophonic, polyphonic, or a complex piano-like texture which is constantly changing). The generalisation can be based on available research into graph domain constraint programming such as Dooms et al. [2005b]. Graph domains have already been applied in fields such as bio-informatics [Dooms et al., 2005a] and computational linguistics [Debusmann et al., 2005]. Tree domains have also been proposed for musical applications. Curry et al. [2000] show how score trees are represented by finite domain constraints.

Nevertheless, a constrainable representation of the hierarchic nesting based on graph domains is still not fully generic, as the nesting of a solution must also correspond with the type of the nested objects (e.g. a note object cannot contain another item).

A further generalisation of Strasheela would make the type information of score objects constrainable. Constraining the type of an object implicitly constrains the set of its attributes and its ADT interface. However, such a generalisation is not compatible with the object-oriented programming paradigm. This means that such a generalisation would not only require a fundamental redesign of Strasheela, but that the redesigned music representation would have to do without the benefits of object-oriented programming such as concision due to incremental ADT definitions. Moreover, it can be difficult to realise such a generalisation with specific domain variables which support constraint propagation. In order to make the object type constrainable, the set of all its attributes and its whole interface must be represented by variables as well. The values for these data can have diverse formats and are therefore difficult to represent with specific domain variables. This data is more easily represented by universal domain variables, but such variables do not support constraint propagation and thus such a representation would slow down the search process considerably.

In general, CSPs become more complex (to define and to solve) when more principles of explicit information support unknown and constrained information. Increasing the amount of unknown and constrained information increases the number of inaccessible score contexts. Inaccessible score contexts, however, complicate the application of compositional rules (cf. Sec. 6.3 and Sec. 8.3.4). For example, in cases where the hierarchic structure is unknown, information such as "which note belongs to which voice" and "which note follows which other note" is unknown and consequently, all rules constraining these contexts must be applied by techniques suitable for inaccessible score contexts. In cases where even the type of an object is unknown, the interface of this object is also unknown and no type-specific function can be applied to the object before enough information about its type becomes available. In addition, inaccessible score contexts also result in efficiency problems. Constraints applied to inaccessible score contexts can only weakly propagate. If the constraint application is delayed (e.g. because type information is missing) no propagation is possible at all. In such a situation, the search process detects a conflict too late (when a delayed rule is applied) which results in severe performance problems.

In summary, it is in principle possible to further generalise Strasheela. However, such a generalisation would complicate the definition of musical CSPs. Moreover, such a generalisation would considerably impair the efficiency of the search process.

10.5. Summary

This chapter compared Strasheela with existing systems in terms of generality. The chapter showed that Strasheela's basic principles for representing score information are only partially supported by existing systems. Consequently, all information stored in the music representation of any existing system can be reproduced in Strasheela, but these

systems can only partly reproduce the information supported by Strasheela.[3] In practice, this has the effect that musical CSPs in existing systems usually express less information about the music because information for which the adequate principle is missing is often omitted. Moreover, the limited support of existing systems for derived information also restricts which score contexts can be constrained. Strasheela's support for modular and concise user extensions of the music representation are also missing in most existing systems. All these factors restrict the definition of musical CSPs in existing systems and in particular these factors make the definition of complex CSPs more difficult in existing systems than in Strasheela.

On the other hand, Strasheela also has a drawback in terms of generality when compared with most existing systems (the exception is Situation). Most of these systems support variables with a universal domain whereas Strasheela supports variables with specific domains. However, specific variable domains allow for constraint propagation which greatly improves efficiency. Moreover, this chapter showed that specific variable domains limit the generality only in certain cases. CSPs where universal variable domains have been most successful in music constraint programming in the past can often be reformulated into a CSP on specific domain variables. In addition, Strasheela is not limited to specific variable domains in principle.

The chapter also compared the rule formalisms of these systems. It was shown that the existing rule application mechanisms (which are featured by the systems Situation and PWConstraints) can be reproduced in Strasheela but not the other way round. Strasheela facilitates the definition of complex CSPs because it allows the user to program rule application mechanisms according to their needs.

Moreover, this chapter compared the search strategy of the systems. It was shown that Strasheela's search process is more flexible than other systems as it can be customised for the needs of the musical CSP at hand. In this matter, the support of user-defined dynamic variable and value orderings are particularly important. Existing systems, on the other hand, are optimised for specific CSP classes and their customisation capabilities are limited. Consequently, these systems are less suited for CSPs for which they are not optimised whereas Strasheela has no such bias towards specific CSP classes.

After showing that Strasheela is in many respects more generic than existing music constraint systems, the chapter asked how Strasheela could be further generalised. Approaches which would make the hierarchic nesting and the typing of score objects constrainable were presented. In addition, the consequences of such generalisations – regarding the ease of CSP definitions and the efficiency of the search process – were discussed.

[3] This claim is valid only when implementation language differences are ignored. For example, Common Lisp supports ratios and complex numbers as built-in data types and thus PWConstraints and Situation can incorporate such data types in their representation. Such data types are missing in Oz and thus Strasheela cannot easily reproduce a representation containing such types. Nevertheless, Oz can be extended by libraries for these types. Alternatively, Strasheela can be re-implemented in Lisp.

11. Conclusion

In this chapter, the main contributions of the present research are summarised (Sec. 11.1) and an outline of future work is presented (Sec. 11.2).

11.1. Main Contributions

This thesis presents a highly generic music constraint system, called Strasheela. Strasheela allows its user to define a large set of musical CSPs and solves them in a reasonably efficient way. When compared with general constraint systems (e.g. general programming languages supporting constraint programming), Strasheela greatly simplifies the definition of musical CSPs by predefining the building blocks required for this class of CSPs. When compared with existing generic music constraint systems (presented in Sec. 3.3), Strasheela allows its user to define and solve a considerably larger set of musical CSPs or makes their definition more modular and concise and that way allows for more complex CSPs to be implemented (see Chap. 10).

Strasheela is more generic than existing music constraint systems primarily because Strasheela is more programmable and more extendable. The Strasheela user controls the information expressed by the music representation, programs how rules are applied to the music, and defines the decision-making of the search process.

Music Representation

The present research proposes a highly expressive music representation, well-suited to defining complex musical CSPs (Chap. 5). When compared with general music representations (presented in Chap. 2), Strasheela adds constrained variables to the representation together with efficient means for solving. When compared with the representations of existing generic music constraint systems, Strasheela's basic principles for representing information (see Sec. 5.6.1) are only partly supported by existing systems (see Sec. 10.1). As a result, Strasheela's representation can contain more information and provides full user control over the contained information. Moreover, Strasheela allows its user to access any score context which can be isolated with the information provided by the score from any object in the score (Sec. 5.6).

Rule Formalism

Strasheela's rule formalism allows the user to constrain complex variable combinations in a convenient way (Chap. 6). The user controls the scope of compositional rules with

rule application mechanisms which can be defined by the user according to needs. In Strasheela, a compositional rule is encapsulated by a first-class function (Sec. 6.1). A rule application mechanism is a higher-order function which expects a rule and a score (or some part thereof) and applies the rule to sets of variables in the score. This thesis presents several rule application mechanisms, including the mechanisms of PWConstraints and Situation (Sec. 6.2). Moreover, Strasheela supports constraining inaccessible score contexts, for example, by implication constraints (Sec. 6.3).

Search Approach

Strasheela solves a wide range of musical CSPs in an efficient way by making the search process adaptable to these problems (Chap. 7). Strasheela employs the space-based constraint model, because this model makes the search process programmable at a high-level (Sec. 7.2). For example, by defining a distribution strategy, the Strasheela user defines a dynamic variable and value ordering which is independent of the CSP definition. Strasheela customises this model for music constraint programming: Strasheela allows a score distribution strategy to exploit any information available in the score whenever it performs a distribution step (Sec. 7.3). Strasheela preserves all other features of the model such as constraint propagation, user-defined exploration strategies, reified constraints, and recomputation (Sec. 7.2.3). This thesis presents special score distribution strategies suitable for a large range of musical CSPs. For example, the text revises the left-to-right search strategy of score-PMC such that the revised strategy efficiently solves polyphonic CSPs including problems which score-PMC explicitly made impossible to define due to their computational complexity (Sec. 7.3.2.3 and Sec. 8.2.2).

Prototype

The present research provides a prototype implementation of the presented model (Chap. 9). In addition, the prototype features enhancements such as style-specific extensions of the music representation (e.g. a harmony model and a motif model) and output into several music representation formats. The prototype comes with a set of example applications and documentation.

11.2. Future Work

User-Guided Search Process

Strasheela solves a CSP autonomously, that is, without any user input besides the CSP itself. This approach requires a full formalisation of all compositional knowledge relevant to solve the CSP. However, a music theory is often presented partly in an informal way. For instance, composition textbooks teach a significant amount of professional skill by musical examples instead of explicit rules.

It would therefore be interesting to investigate how a music constraint system can support both forms of user knowledge: explicit compositional knowledge (formalised in a set

of rules) and knowledge which is learnt from examples but has not been explicated. For example, Ovans and Davison [1992] propose a system which closely integrates formalised compositional rules followed automatically by the systems with a manual composition process. The rules only perform constraint propagation, but every decision during the search process is made by the user. The system implements first species Fuxian counterpoint, and features a graphical user interface which shows the pitch domain of each note (this domain is shown like a chord). The user determines the pitch of a note by clicking on one of its domain pitches. Constraint propagation then reduces the domains of the other notes according to the explicit rules. It would be interesting to investigate how such an approach can be generalised for arbitrary musical CSPs. A generalisation requires a very different user interface, because the approach of Ovans and Davison is limited to CSPs which only constrain the pitches of monophonic melodies. Moreover, it is desirable to allow for an optional automatic completion of a hand-edited partial solution, as well as for manual changes to a solution. For example, the user may enter important parts by hand and then ask the system to fill in the details. Also, the user may change details of an unsatisfying solution and ask for a new solution which fits the changes.

Integrating Soft Constraints

Music theories are usually full of exceptions which are very hard to fully formalise. Moreover, most compositional rules only express preferences instead of strict rules. Also, composers wish to grade the importance of compositional rules such that less important rules may be neglected in an over-constraint situation.

Preferences can be expressed by two techniques in Strasheela: reified constraints and best-solution search. For example, reified constraints can express the probability of how often a rule must be fulfilled (Sec. 7.2.3.4). Best solution search (Sec. 7.2.3.3) finds multiple solutions and always constrains the next solution to be better according to a user-defined criterion. This criterion can express a conjunction of preferences, possibly graded by importance (this technique was applied by Henz et al. [1996] to express soft compositional rules). Still, both techniques express preferences only indirectly by a combination of strict constraints. In effect, these techniques are relatively inflexible and inefficient.

Therefore, it would be interesting to study alternative approaches which introduce *soft constraints* as single propagators which can be relaxed. For example, [Bistarelli et al., 2002] propose a highly generic way to express concurrent soft constraints based on the notion of semirings (i.e. an algebraic structure similar to a ring which defines a set of values and the two binary operators + and $\times$). These constraints can express well-known soft CSP classes including fuzzy CSPs, probabilistic CSPs and weighted CSPs (these cases differ in the semiring of truth values they are based on). Delgado et al. [2005] present an implementation of semiring-based constraints for Oz which can combine hard and soft constraints in a single CSP and which could be applied directly in Strasheela.

User Interface

The current Strasheela prototype employs the Oz language as user interface because of its unique features as a multi-paradigm programming language. However, Oz has hardly been used in the field of computer music so far and consequently a new user needs some effort to learn Strasheela. In particular composers and music theorists without much programming experience are likely to shy away from a system which requires learning a new programming language.

Porting Strasheela to a language which is more widely-used in the computer-aided composition community or which offers special support for unexperienced programmers makes the system more easily accessible. An example of a widely-used language is Common Lisp, because there exist a considerable number of Lisp-based composition systems (e.g. Common Music [Taube, 2004], PatchWork [Laurson, 1996] with its successors OpenMusic [Assayag et al., 1997] and PWGL [Laurson and Kuuskankare, 2002], ACToolbox [Berg]). Visual languages enjoy much popularity among musicians (especially data-flow language such as Max and its relatives [Puckette, 2002]). Section 3.3.1 discussed the benefits and disadvantages of visual programming. It would be interesting to investigate how visual programming can facilitate the use of a generic music constraint system like Strasheela in a way that scales well even for complex CSPs. For instance, it may be suitable that the user defines in a visual way the top-level of a CSP (which usually only instantiates the score and applies rules to it), whereas the constructs used by this top-level (e.g. the actual rules) are defined textually. An alternative approach offering special support for unexperienced programmers is the use of a programming environment for a purely textual language, which is especially targeting programming beginners as exemplified by DrScheme [Findler et al., 2002].

Learning-Based Rule Generation

The Strasheela user defines a music theory model completely manually. In the context of computer-aided composition, this approach has the advantage that the user has full control and can even define a theory for a musical style which does not exist yet. Composers are often primarily interested in developing their own distinct musical language instead of replicating an existing musical style. Nevertheless, this freedom is coupled with the effort of creating such a music theory definition.

Research on computational models of music composition often also applies a learning-based approach where compositional knowledge is deduced from existing pieces. For example, Hild et al. [1992] propose HARMONET, which uses a connectionist approach in order to learn choral harmonisation from Bach choral examples. Cope [1991, 1996] presents EMI, a system which analyses given pieces of music and uses the result of this analysis in order to create similar pieces. For example, EMI extracts reappearing phrases typical for a certain style (called signatures) and analyses the function of musical fragments (e.g. whether they raise an expectation or express a conclusion, using a grammar called SPEAC).

It would be interesting to integrate learning techniques into Strasheela in such a way

that the user would mix manually-defined compositional knowledge with explicitly represented knowledge deduced from existing pieces. If the knowledge learnt is represented in the form of rules, then this knowledge can be freely mixed with handwritten rules. For example, [Morales and Morales, 1995] propose a system which automatically creates rules in first-order logic (horn clauses) given a musical example and rule templates. The textbook [Mitchell, 1997] introduces learning techniques including the learning of rules.

Real-Time Constraint Programming

Many computer musicians are interested in creating interactive music performance software, as exemplified by the large user community of systems like Max/MSP and SuperCollider. Programs created by these systems can show a very complex behaviour (e.g. demonstrated by the interactive real-time composition Lexikon-Sonate [Essl, 2004]). Nevertheless, due to the procedural nature of these systems it is hard to create software whose output complies with a complex music theory (e.g. some theory of harmony conceived by the user).

It would be interesting to explore how constraint programming can facilitate the creation of interactive music software which implements complex music theories. SuperCollider already supports constraint programming, but only in a very limited form: constraints can filter undesirable values out of a SuperCollider pattern.[1] Oz, on the other hand, provides several features which makes it a promising platform for real-time constraint programming. For instance, soft real-time programming capabilities are provided by its scheduling facilities. Also, Oz can emulate Max' dataflow programming approach: Oz data (e.g. numbers or score fragments) can flow through streams between concurrent processes which synchronise on the availability of data via logic variables (cf. Sec. 7.1). A process in such a dataflow program can conduct any computation, including solving a CSP. For example, the full Fuxian example (Sec. 8.1) takes only about 50 msec to solve, which may be fast enough for a real-time context. Oz supports techniques which can ensure that a process takes only a limited amount of time in an interactive program. Oz supports both "don't know" nondeterminism (where a CSP is solved by a complete search; this was the only approach discussed so far) and also "don't care" nondeterminism (where a decision may require fulfilling complex constraints but is never taken back once made; this is also known as committed choice constraint programming). While "don't know" nondeterminism is well-suited for creating composition software, "don't care" nondeterminism is well-suited for creating interactive improvisation software. Moreover, search is encapsulated in Oz: a search process runs concurrently in its own thread (i.e. it does not block the rest of the program) and it can be cancelled in cases where it does not find a solution within a limited amount of time. A complete search for a sub-CSP can even be encapsulated within a committed choice CSP. Finally, Oz' networking support (e.g. support for sockets) allows for a coupling with other systems such as Max or SuperCollider.

[1] This feature is implemented and documented by the so-called crucial library which is part of SuperCollider.

Appendix A.

Notational Conventions

Throughout this thesis, mathematical notation is used as a kind of pseudo-code to communicate examples. For this purpose, decisions for notational variants aimed for legibility instead of rigour.

The notation complements first-order logic notation by functions as first-class values plus a number of data types with a literal representation.

Great care has been taken to choose a notation which follows common conventions. Still, the present appendix provides a brief summary of this notation.

The notation discussed here is the notation used in this thesis. The user-interface of the Strasheela prototype, on the other hand, uses the Oz programming language (see Sec. 9). The mathematical notation introduced here has solely been chosen to simplify the discussion in this text: it is not required to know any particular programming language for reading the examples given here.

A.1. Variables

The notation of a variable reflects the nature of its value to make examples more readable (see Tab. A.1[1]). In general, a variable is notated starting with a lower case letter. Such a variable can be bound to, for instance, a number, a function, or an ADT.

A constrained variable (i.e. a partial value with a domain) is usually notated starting with an upper case letter to stand out more clearly. Such a variable can be understood as a logically quantified variable. In case it is not decided whether a variable is constrained (e.g. if a variable is an argument in a function definition) the lower case notation may be used.

It should be noted that a variable which binds a composite value or an ADT is always notated with a lower case letter (e.g. *myNote*) even if it contains a constrained variable. For example, the pitch value of *myNote* may be a constrained variable (e.g. $MyPitch = getPitch(myNote)$), but *myNote* itself is written starting with a lower case letter.

[1] For comprehensibility, instead of an abstract syntax specification (e.g. in BNF notation) an informal mix of simple examples and BNF constructs are used.

Concept	Notation
variable	x
constrained variable (partial value)	X
set variable	$\mathbf{S}$
(variable) definition	$x \stackrel{def}{=} \langle expression \rangle$
introducing a local variable (first form)	**let** $\langle definitions \rangle$ **in** $\langle expression \rangle$
introducing a local variable (second form)	$\langle expression \rangle$ **where** $\langle definitions \rangle$
notation short hand (second form)	$\langle expression \rangle : \langle definitions \rangle$

Table A.1.: Variable notation

A set is notated in bold face.

For convenience, the notation does not necessarily require to explicitly declare a variable. The scope of a variable is its surrounding block, for instance, a function definition. Many examples explicitly define a local scope for variables with either the **let** or the **where** form. The where-notation can also be used to express a number of conditions on a value as in $\langle expression \rangle : \langle conditions \rangle$.

A.2. Functional Abstraction

In this thesis, a function is a first-class value bound to a variable (i.e. a λ-expression, see discussion in Sec. 2.8). Nevertheless, the common notation for a function definition and function application is used (see Tab. A.2), making examples most easy to read by avoiding the less widespread lambda calculus notation. Instead, the present text uses the *where-notation* [Landin, 1966] as syntactic sugar for an anonymous function. For example, the function f serves the purpose of an anonymous function in $g(f \,\mathbf{where}\, f(x) \stackrel{def}{=} x^2)$ or shorter $g(f : f(x) \stackrel{def}{=} x^2)$.

For convenience, the notation allows for functions with keyword arguments (named arguments), which are also optional arguments.

A.3. Control Structures

The if-then-else conditional is written in a common way. This expression always returns a value (see Tab. A.3).

Concept	Notation Example
(function) definition	$square(x) \stackrel{def}{=} x^2$
function application	$square(3) = 9$
"inline" function definition (serves purpose of anonymous function)	$map([1,2,3], f : f(x) \stackrel{def}{=} x^2) = [1,4,9]$
function with optional keyword arguments (definition and applications without and with optional argument)	$addN(x, n{:}\,1) \stackrel{def}{=} x + n$ $addN(3) = 4$ $addN(3, n{:}\,2) = 5$

Table A.2.: Functional abstraction

Concept	Notation Example
conditional	**if** ⟨*boolean expression*⟩ **then** ⟨*expression*⟩ **else** ⟨*expression*⟩

Table A.3.: Control structures

A.4. Values and Data Structures

A number of data types is used in the present text. Not all these data types are used in common mathematical notation, therefore the notation is also inspired by the syntax of various programming languages and Backus-Naur form (see Tab. A.4).

A *symbol* (which evaluates to itself) is notated starting with a lower case letter, much like a variable (which evaluates to its value). Usually, the context will tell what is meant.

In mathematical notation, a *list* is often notated simple as $1, 2, 3$. Instead, the notation used here is slightly more explicit and encloses a list in brackets. The empty list is denoted with the symbol *nil*. A *set* is written in the common way (i.e. enclosed by curly braces). A *record* is a composite data structure with a label (in Tab. A.4, the label is *myLabel*) and values (x_1 to x_3) at named features (a to c). Although the record notation is similar to the notation of a function call, the context will tell what is meant. Following conventions of mathematical notation, the elements in a list, set, and record are separated by colons.

A.5. Operations

The notation of operations on values (e.g. numerical operations, set operations, and logical connectives) follows the usual conventions (see Tab. A.5).

The operational semantics of these operations is overloaded for simplicity. For example, the notation of this thesis only uses a single equality operator ($x = y$). This operator can have a number of operational meanings. These meanings are clearly distinguished in

Concept	Notation Example
two truth values	$true$, $false$ or 1, 0
number	1
symbol	$mySymbol$
list (sequence) and empty list	$[1, 2, 3]$ and nil
set	$\{1, 2, 3\}$
record with a number of features and their values	$myLabel(a\colon x_1, b\colon x_2, c\colon x_3)$
ADT (i.e. a value without literal representation)	$\langle myValue \rangle$

Table A.4.: Values and data structures (which evaluate to themselves)

the Oz notation (and thus the Strasheela prototype examples), for instance, unification (`X = Y`), the equality test returning a boolean (`X == Y`), and the equality constraint (`X =: Y`).

Similarly, the examples in Tab. A.5 show the overloading of numerical and set operators. These operators can be applied as functions with determined arguments returning a determined value but also as constraints to constrained variables.

Concept	Notation Example
numerical operation example (deterministic addition and addition constraint)	$x = y + z$ $X = Y + Z$
set operation example (boolean membership predicate and membership constraint)	$x \in \mathbf{Set}$ $X \in Set$
logical connectives (conjunction, disjunction, negation, implication and equivalence)	$B_1 \wedge B_2$ $B_1 \vee B_2$ $\neg B$ $B_1 \Rightarrow B_2$ $B_1 \Leftrightarrow B_2$
cumulative conjunction	$\bigwedge[B_1, B_2, B_3]$ or $\bigwedge_{i=1}^{n} B_i$

Table A.5.: Operations

Appendix B.

Additional Definitions

The present appendix provides definitions which were released from the body of this thesis for brevity. The main text motives these functions, explains their behaviour and points here into the appendix for their definition. In turn, each definition shown here references the section in which it was discussed.

$$
\begin{array}{lll}
\mathit{getTemporalTopLevel}(x) \stackrel{def}{=} & \textbf{let} & c \stackrel{def}{=} \mathit{find}\,(\mathit{getContainers}(x), \\
 & & \qquad \mathit{isTemporalAspect}) \\
 & \textbf{in} & \textbf{if} \quad c = \mathit{nil} \\
 & & \textbf{then} \; x \\
 & & \textbf{else} \quad \mathit{getTemporalTopLevel}(c)
\end{array}
$$

Figure B.1.: The function *getTemporalTopLevel* is discussed in Sec. 5.6.2.1

$$
\begin{array}{ll}
\mathit{zip}(xs, ys, fn) \stackrel{def}{=} & \textbf{if}\; xs = \mathit{nil} \\
 & \textbf{then}\; \mathit{nil} \\
 & \textbf{else}\; \mathit{cons}\,(\mathit{fn}(\mathit{head}(xs), \mathit{head}(ys)), \\
 & \qquad \mathit{zip}(\mathit{tail}(xs), \mathit{tail}(ys), \mathit{fn}))
\end{array}
$$

Figure B.2.: The function *zip* is discussed in Sec. 6.2.4

```
let /* The function ranges returns a list of elements in xs which match decls. decls is a list consisting
       of single indices and startIndex#endIndex pairs.                                          */
    ranges(xs, decls) ≝
        /* mappend(xs, fn) is a variant of map(xs, fn). The mappend argument fn must return a list.
           mappend appends all collected sublists (hence the name: map-append or mappend).      */
        mappend(decls,
                f : f(decl) ≝ if isPair(decl)
                              /* decl is a start#end pair.
                                 The expression sublist(xs, start, end) accesses the sublist of xs
                                 that consists of the start-th to end-th elements (including).  */
                              then let start#end ≝ decl
                                   in  sublist(xs, start, end)
                              /* decl is a single index.                                        */
                              else [nth(xs, decl)])
in  mapIndex(xs, decls, fn) ≝ let matchingXs ≝ ranges(xs, decls)
                              in  map(matchingXs, fn)
```

Figure B.3.: The function *mapIndex* is discussed in Sec. 6.2.5

```
mapPM(xs, patternExpr, fn) ≝
  let /* collectPM is a highly recursive auxiliary function which returns in a list any list of elements from
         xs (a list) which match the patternExpr (a list of pattern symbols). The argument matchingXs
         (initially always nil) is used to pass matching elements accumulated so far to the next recursive
         call of collectPM.                                                                          */
      collectPM(xs, patternExpr, matchingXs) ≝
        if   patternExpr = nil ∨ xs = nil
        then [reverse(matchingXs)]
        else /* Recursively call collectPM – depending on the first symbol in patternExpr.          */
             let symbol ≝ head(patternExpr)
                 xsTail ≝ tail(xs)
                 patternTail ≝ tail(patternExpr)
             in  result where
                                  ⎧ collectPM(xsTail, patternTail, matchingXs)                  if symbol = ?
                                  ⎪ collectPM(xsTail,                                           if symbol = x
                                  ⎪           patternTail,
                                  ⎪           cons(head(xs), matchingXs))
                                  ⎪ let /* minimal length of remaining pattern is length of     if symbol = ∗
                                  ⎪     patternTail without any occurence of * (Kleene star)
                                  ⎪     */
                                  ⎪     minLength ≝ length( filter(patternTail,
                                  ⎪                                f : f(x) ≝ x ≠ ∗))
                      result =    ⎪ in  /* mappendTail recursively applies f to every non-nil
                                  ⎪     tail of xs and appends the resulting lists.          */
                                  ⎪      mappendTail(xs,
                                  ⎪                  f : f(sublist) ≝
                                  ⎪                    if   length(sublist) ≥ minLength
                                  ⎪                    then collectPM(sublist,
                                  ⎪                                   patternTail,
                                  ⎪                                   matchingXs)
                                  ⎪                    else nil)
                                  ⎩
  in  map(collectPM(xs, patternExpr, nil), fn)
```

Figure B.4.: The function *mapPM* is discussed in Sec. 6.2.6

```
getTestIndex(x, tests) ≝

  /* The function findPosition expects a list and a unary boolean function and returns the position
     of the first list element for which the function returns true. In getTestIndex, this list consists of
     boolean functions.                                                                              */
  findPosition(tests, f : f(test) ≝ test(x))
```

Figure B.5.: The function *getTestIndex* is discussed in Sec. 7.3.2.2

$direction(X_1, X_2, Dir) \stackrel{def}{=}$

/ Dir is constrained to the direction of the interval between X_1 and X_2. An interval 'upwards' (the predecessor is smaller than the successor) is represented by 2, an 'horizontal' interval (the predecessor and the successor are equal) is represented by 1, and an interval 'downwards' by 0. X_1, X_2, and Dir are all finite domain integers. */*

let $IsUp \stackrel{def}{=} fdInt(\{0, 1\})$
$IsEq \stackrel{def}{=} fdInt(\{0, 1\})$
in $0 \leq Dir \leq 2$
$IsUp = (X_1 < X_2)$
$IsEq = (X_1 = X_2)$
$Dir = 2IsUp + IsEq$

Figure B.6.: The function *direction* is discussed in Sec. 8.1.2.2

Appendix C.

The Strasheela Website

Chapter 9 introduced the Strasheela prototype which implements the Strasheela model presented in this text and also provides several extensions of this model. The prototype be obtained from Sourceforge at `https://sourceforge.net/projects/strasheela/`. The Strasheela website (`http://strasheela.sourceforge.net`) provides documentation for this prototype which includes installation instructions, links to relevant Oz documentation, as well as reference documentation for the Strasheela core and its extensions. In addition, the website presents and explains several Strasheela application examples (`http://strasheela.sourceforge.net/strasheela/doc/StrasheelaExamples.html`). These include not only the examples presented in chapter 8, but also other examples as well.

Appendix D.

Source Material

The material presented in this text has been partly published before in the following articles.

- Anders, T. (2000). Arno: Constraints Programming in Common Music. In *Proceedings of the 2000 International Computer Music Conference* [Anders, 2000].
- Anders, T. (2002). A wizard's aid: efficient music constraint programming with Oz. In *Proceedings of the 2002 International Computer Music Conference* [Anders, 2002].
- Anders, T., C. Anagnostopoulou, and M. Alcorn (2005). Strasheela: Design and Usage of a Music Composition Environment Based on the Oz Programming Model. In P. v. Roy (Ed.), *Multiparadigm Programming in Mozart/OZ: Second International Conference, MOZ 2004*, LNCS 3389. Springer-Verlag [Anders et al., 2005].

In addition, material was presented in the following talks.

- Anders, T. (2004). The Score Description Language: A Tutorial. PRISMA Meeting,[1] Centro Tempo Reale, Florence, January 2004.
- Anders, T. (2004). Comparing Constraint Based Composition Systems: A Survey. PRISMA meeting, IRCAM, Paris, June 2004.
- Anders, T. (2005). Composing Music by Composing Rules. Invited lecture, Music Informatics Research Group, Edinburgh University, March 2005.
- Anders, T. (2006). Why is Strasheela a Generic Music Constraint System? PRISMA meeting, IRCAM, Paris, June 2006.

[1] PRISMA (Pedagogia e Ricerca Internazionale sui Sistemi Musicali Assistiti) is an international group of about twenty composers and researchers, where many members have expertise in music constraint programming.

Bibliography

Abelson, H., G. J. Sussman, and J. Sussman (1985). *Structure and Interpretation of Computer Programs*. MIT Press.

Agon, C., G. Assayag, J. Baboni, and K.Haddad (2001). *OpenMusic (OM) 4.0. User's Manual Reference & Tutorial*. Paris: IRCAM.

Agon, C., G. Assayag, O. Delerue, and C. Rueda (1998). Objects, Time and Constraints in OpenMusic. In *Proceedings of the 1998 International Computer Music Conference*, Ann Arbor, Michigan.

Alvarez, G., J. F. Diaz, L. O. Quesada, F. D. Valencia, G. Assayag, and C. Rueda (1998). PiCO: A Calculus of Concurrent Constraint Objects for Musical Applications. In *ECAI 98 Workshop on Constraints for Artistic Applications*, Brighton.

Anders, T. (2000). Arno: Constraints Programming in Common Music. In *Proceedings of the 2000 International Computer Music Conference*, Berlin, Germany.

Anders, T. (2002). A wizard's aid: efficient music constraint programming with Oz. In *Proceedings of the 2002 International Computer Music Conference*, Göteborg, Sweden.

Anders, T. (2003). Composing Music by Composing Rules: Computer Aided Composition employing Constraint Logic Programming. Technical report, Sonic Arts Research Centre, Queen's University Belfast.

Anders, T. (2006). Strasheela. `http://strasheela.sourceforge.net/` (accessed 29 May 2006).

Anders, T., C. Anagnostopoulou, and M. Alcorn (2005). Strasheela: Design and Usage of a Music Composition Environment Based on the Oz Programming Model. In P. V. Roy (Ed.), *Multiparadigm Programming in Mozart/OZ: Second International Conference, MOZ 2004*, Number 3389 in Lecture Notes in Computer Science. Springer-Verlag.

Apt, K. R. (2003). *Principles of Constraint Programming*. Cambridge University Press.

Ariza, C. (2005). Navigating the Landscape of Computer-Aided Algorithmic Composition Systems: A Definition, Seven Descriptors, and a Lexicon of Systems and Research. In *Proceedings of the International Computer Music Conference*.

Ariza, C. (2006). algorithmic.net: A Lexicon of Systems and Research in Computer-Aided Algorithmic Music Composition. `http://www.flexatone.net/algoNet/index.html` (accessed 15 May 2006).

Assayag, G. (1998). Computer Assisted Composition Today. In *1st Symposium on Music and Computers*, Corfu.

Assayag, G. and C. Agon (1996). OpenMusic Architecture. In *Proceedings of ICMC 96*, Hong Kong, China.

Assayag, G., C. Agon, J. Fineberg, and P. Hanappe (1997). An Object Oriented Visual Environment For Musical Composition. In *Proceedings of the International Computer Music Conference*, Thessaloniki.

Assayag, G., C. Rueda, M. Laurson, C. Agon, and O. Delerue (1999). Computer Assisted Composition at IRCAM: From PatchWork to Open Music. *Computer Music Journal 23*(3).

Avispa Research Group (2005). XRI and LP for Mozart. `https://gna.org/projects/xrilpoz/` (accessed 4 August 2006).

Avispa Research Group (2006). GEOZ - Improving Mozart Constraints Serv. `http://home.gna.org/xrilpoz/index.html` (accessed 4 August 2006).

Balaban, M. (1996). The Music Structures Approach to Knowledge Representation for Music Processing. *Computer Music Journal 20*(2).

Ballesta, P. (1994). *Contraintes et objets: clefs de voûte d'un outil d'aide á la composition.* Ph. D. thesis, Université du Maine.

Barendregt, H. and E. Barendsen (1991). Introduction to Lambda Calculus. Technical report, Department of Computer Science, Catholic University of Nijmegen.

Bartak, R. (1998). On-Line Guide to Constraints Programming. `http://kti.mff.cuni.cz/~bartak/constraints/` (accessed 25 May 2006).

Baum, L. F. (1993, orig. 1900). *The Wonderful Wizard of Oz.* Wordsworth.

Berg, P. AC Toolbox. `http://www.koncon.nl/ACToolbox/` (accessed 24 May 2006).

Berlioz, H. and R. Strauss (1904). *Instrumentationslehre.* Edition Peters.

Bistarelli, S., U. Montanari, and F. Rossi (2002). Soft concurrent constraint programming. In *European Symposium on Programming*, Grenoble, France.

Bonnet, A. and C. Rueda (1999). *OpenMusic. Situation. version 3* (3rd ed.). Paris: IRCAM.

Booch, G. (1991). *Object Oriented Design with Applications.* Benjamin-Cummings.

Booch, G., J. Rumbaugh, and I. Jacobson (1998). *The Unified Modeling Language User Guide.* Addison Wesley.

Boshernitsan, M. and M. Downes (2004). Visual Programming Languages: A Survey. Technical Report UCB/CSD-04-1368, Computer Science Division (EECS), University of California, Berkeley.

Boulanger, R. (Ed.) (2000). *The Csound Book. Perspectives in Software Synthesis, Sound Desing, Signal Processing, and Programming.* The MIT Press.

Bratko, I. (2001). *Prolog. Programming for Artificial Intelligence* (3rd ed.). Addison-Wesley.

Bresson, J., C. Agon, and G. Assayag (2005). OpenMusic 5: A Cross-Platform Release of the Computer-Assisted Composition Environment. In *10th Brazilian Symposium on Computer Music*, Belo Horizonte, Brazil.

Burnett, M. (2006). Visual Language Research Bibliography. `http://web.engr.oregonstate.edu/~burnett/vpl.html`, (accessed 27 April 2006).

Cheadle, A. M., W. Harvey, A. J. Sadler, J. Schimpf, K. Shen, and M. G. Wallace (2003). Eclipse: An introduction. Technical Report IC-PARC-03-1, IC-Parc, Imperial College London.

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

Choi, C. W., M. Henz, and K. B. Ng (2001). Components for State Restoration in Tree Search. In T. Walsh (Ed.), *Principles and Practice of Constraint Programming – CP 2001 : 7th International Conference*, Volume 2239 of *Lecture Notes in Computer Science*. Springer-Verlag.

Codognet, P. and D. Diaz (2001). Yet Another Local Search Method for Constraint Solving. In T. Walsh and C. Gomes (Eds.), *Proceedings of the AAAI Symposium "Using Uncertainty in Computation"*. AAAI Press.

Codognet, P., D. Diaz, and C. Truchet (2002). The Adaptive Search Method for Constraint Solving and its Application to musical CSPs. In *IWH02, International Workshop on Heuristics*, Beijing, China.

Conklin, D. and I. H. Witten (1995). Multiple Viewpoint Systems for Music Prediction. *Journal of New Music Research 24*(1).

Cooper, G. and L. B. Meyer (1960). *The Rhythmic Structure of Music.* The University of Chicago Press.

Cope, D. (1991). *Computers and Musical Style.* Madison, WI: A-R Editions.

Cope, D. (1996). *Experiments in Musical Intelligence.* Madison, WI: A-R Editions.

Cormen, T. H., C. E. Leiserson, and R. L. Rivest (2001). *Introduction to Algorithms* (Second ed.). MIT Press.

Courtot, F. (1990). A Constraint Based Logic Program for Generating Polyphonies. In *Proceedings of the International Computer Music Conference*, Glasgow.

Curry, B., G. A. Wiggins, and G. Hayes (2000). Representing Trees with Constraints. In *Proceedings of the First International Conference on Computational Logic*, Lecture Notes in Computer Science 1861. Springer-Verlag.

Dannenberg, R. B. (1984). Arctic: A Functional Language for Real-Time Control. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. ACM Press New York, NY, USA.

Dannenberg, R. B. (1989). The Canon Score Language. *Computer Music Journal 13*(1).

Dannenberg, R. B. (1993). Music Representation Issues, Techniques, and Systems. *Computer Music Journal 17*(3).

Dannenberg, R. B. (1997). Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis. *Computer Music Journal 21*(3).

Debusmann, R., D. Duchier, and J. Niehren (2005). The XDG Grammar Development Kit. In P. V. Roy (Ed.), *Multiparadigm Programming in Mozart/OZ: Second International Conference, MOZ 2004*, Lecture Notes in Computer Science 3389. Springer-Verlag.

Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.

Delgado, A., C. A. Olarte, J. A. Pérez, and C. Rueda (2005). Implementing Semiring-Based Constraints Using Mozart. In P. V. Roy (Ed.), *Multiparadigm Programming in Mozart/OZ: Second International Conference, MOZ 2004*, Lecture Notes in Computer Science 3389. Springer-Verlag.

Dent, M. J. and R. E. Mercer (1994). Minimal Forward Checking. In *6th IEEE International Conference on Tools with Artifcial Intelligence*, New Orleans, Louisiana.

Desain, P. (1990). LISP as a Second Language: Functional Aspects. *Perspectives of New Music 28*(1).

Desain, P. and H. Honing (1988). LOCO: A Composition Microworld in Logo. *Computer Music Journal 12*(3).

Desain, P. and H. Honing (1997). CLOSe to the edge? Advanced object oriented techniques in the representation of musical knowledge. *Journal of New Music Research 26*(1).

Deville, Y., G. Dooms, S. Zampelli, and P. Dupont (2005). CP(Graph+Map) for Approximate Graph Matching. In F. Azevedo, C. Gervet, and E. Pontelli (Eds.), *1st International Workshop on Constraint Programming Beyond Finite Integer Domains*, Sitges, Barcelona, Spain.

Diaz, D. (2003). The GNU Prolog web site. `http://gnu-prolog.inria.fr` (accessed 25 May 2006).

Díaz, J. F., G. Gutierrez, C. A. Olarte, and C. Rueda (2005). Using Constraint Programming for Reconfiguration of Electrical Power Distribution Networks. In P. V. Roy (Ed.), *Multiparadigm Programming in Mozart/OZ: Second International Conference, MOZ 2004*, Lecture Notes in Computer Science 3389. Springer-Verlag.

Dooms, G., Y. Deville, and P. Dupont (2005a). A Mozart Implementation of CP(BioNet). In P. V. Roy (Ed.), *Multiparadigm Programming in Mozart/OZ: Second International Conference, MOZ 2004*, Lecture Notes in Computer Science 3389. Springer-Verlag.

Dooms, G., Y. Deville, and P. Dupont (2005b). CP(Graph): Introducing a Graph Computation Domain in Constraint Programming. In *11th International Conference on Principles and Practice of Constraint Programming*, Number 3709 in Lecture Notes in Computer Science, Sitges, Barcelona, Spain. Springer-Verlag.

Doty, D. B. (2002). *The Just Intonation Primer. An Introduction to the Theory and Practice of Just Intonation* (Third ed.). San Francisco, CA: Just Intonation Network.

Duchier, D., C. Gardent, and J. Niehren (1998). Concurrent Constraint Programming in Oz for Natural Language Processing. `http://www.ps.uni-sb.de/~niehren/Web/Vorlesungen/Oz-NL-SS01/BookHomePage.html` (accessed 25 May 2006).

Duchier, D., L. Kornstaedt, M. Homik, T. Müller, C. Schulte, and P. V. Roy (2004). *Mozart Documentation. System Modules* (1.3.1 ed.). Mozart Consortium. `http://www.mozart-oz.org/documentation/system/index.html` (accessed 25 May 2006).

Ebcioglu, K. (1980). Computer Counterpoint. In *Proceedings of the International Computer Music Conference 1980*, Queens College, New York City, USA.

Ebcioglu, K. (1984). An expert system for schenkerian synthesis of chorales in the style of J.S.Bach. In *Proceedings of the 1984 International Computer Music Conference*, Paris.

Ebcioglu, K. (1992). An Expert System for Harmonizing Chorales in the Style of J.S. Bach. In M. Balaban, K. Ebcioglu, and O. Laske (Eds.), *Understanding Music with AI: Perspectives on Music Cognition*, Chapter 12. MIT Press.

Essl, K. (2004). Lexikon-Sonate. `http://www.essl.at/works/Lexikon-Sonate.html` (accessed 8 August 2006).

Ferrand, M., J. A. Leite, and A. Cardoso (1999). Improving Optical Music Recognition by means of Abductive Constraint Logic Programming. In *Portuguese Conference on Artificial Intelligence*, Number 1695 in Lecture Notes in Artificial Intelligence. Springer.

Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen (2002). DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming 12*(2).

Forte, A. (1973). *The Structure of Atonal Music*. Yale University Press.

Forte, A. and S. E. Gilbert (1982). *Introduction to Schenkerian Analysis*. W. W. Norton & Company.

Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd ed.). Addison Wesley.

Frühwirth, T. and S. Abdennadher (2003). *Essentials of Constraint Programming*. Springer.

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

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns*. Addison-Wesley.

Gecode authors (2006). Gecode. generic constraint development environment. `http://www.gecode.org/` (accessed 22 April 2006).

Gervink, M. (2003, orig. 1995). Die Strukturierung des Tonraums. Versuche einer Systematisierung von Zwölftonreihen in den 1920er bis 1970er Jahren. In W. Auhagen, B. Gätjen, and K. W. Niemöller (Eds.), *Systemische Musikwissenschaft. Festschrift Jobst P. Fricke zum 65. Geburtstag*. available at `http://www.uni-koeln.de/phil-fak/muwi/fricke/` (accessed 29 May 2006).

Harris, M., A. Smaill, and G. Wiggins (1991). Representing Music Symbolically. In *IX Colloquio di Informatica Musicale*, Genoa, Italy.

Harvey, W. D. and M. L. Ginsberg (1995). Limited Discrepancy Search. In C. S. Mellish (Ed.), *Fourteenth International Joint Conference on Artificial Intelligence*, Montéal, Québec, Canada. Morgan Kaufmann Publishers.

Henz, M., S. Lauer, and D. Zimmermann (1996). COMPOzE — intention-based music composition through constraint programming. In *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence*, Toulouse, France. IEEE Computer Society Press.

Hild, H., J. Feulner, and W. Menzel (1992). HARMONET: A Neural Net for Harmonizing Chorales in the Style of J.S.Bach. In R. Lippmann, J. Moody, and D. Touretzky (Eds.), *Advances in Neural Information Processing Systems 4 (NIPS 4)*, San Mateo, CA. Morgan Kaufmann Publishers.

Hiller, L. and L. Isaacson (1993, orig. 1958). Musical Composition with a High-Speed Digital Computer. In S. M. Schwanauer and D. A. Lewitt (Eds.), *Machine Models of Music*. MIT press. reprint of original articel in Journal of Audio Engineering Society, 1958.

Honing, H. (1993). Issues in the representation of time and structure in music. *Contemporary Music Review 9*.

Hudak, P. The Haskore Computer Music System. `http://haskell.org/haskore/` (accessed 24 May 2006).

Huron, D. (2002). Music Information Processing Using the Humdrum Toolkit: Concepts, Examples, and Lessons. *Computer Music Journal 26*(2).

ILOG. ILOG Solver. `http://www.ilog.com/products/solver/` (accessed 24 May 2006).

Jacobson, I. (1992). *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison Wesley,.

Jeppesen, K. (1971, orig. 1930). *Kontrapunkt* (4th ed.). Leipzig: Breitkopf & Härtel.

Kelly, J. (1997). *The Essence of Logic*. Prentice Hall.

Kiczales, G., J. des Rivieres, and D. G. Bobrow (1991). *The Art of the Metaobject Protocol*. MIT Press.

Koch, H. C. (2000, orig. 1782–1793). *Versuch einer Anleitung zur Kompositions*. in Rudolstadt. reprint: Georg Olms Verlag.

Kornstaedt, L. and D. Duchier (2004). *The Oz Programming Interface* (1.3.1 ed.). Mozart Consortium. `http://www.mozart-oz.org/documentation/system/index.html` (accessed 25 May 2006).

Kretz, J. (2003). Continuous Gestures of Structured Material. Experiences in Computer Aided Composition. In *PRISMA 01*. Milano: EuresisEdizioni.

Kumar, V. (1992). Algorithms for constraints satisfication problems: A survey. *AI Magazine 13*(1).

Křenek, E. (1952). *Zwölfton-Kontrapunkt-Studien*. Mainz: B. Schott's Söhne.

Landin, P. J. (1966). The Next 700 Programming Languages. *Communications of the ACM 9*(3).

Laske, O. (1981). Composition theory in Koenig's Project One and Project Two. *Computer Music Journal 5*(4).

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

Laurson, M. (1999). Recent Developments in PatchWork: PWConstraints – a Rule Based to Complex Musical Problems. In *Symposium on Systems Research in the Arts*, Baden-Baden. IIAS.

Laurson, M. and J. Duthen (1989). PatchWork, a graphical language in PreForm. In *Proceedings of the International Computer Music Conference*, San Francisco, pp. 172–175.

Laurson, M. and M. Kuuskankare (2000). Towards Idiomatic Instrumental Writing: A Constraint Based Approach. In *Symposium on Systems Research in the Arts*, Baden-Baden.

Laurson, M. and M. Kuuskankare (2001). A Constraint Based Approach to Musical Textures and Instrumental Writing. In *Seventh International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*, Paphos, Cyprus.

Laurson, M. and M. Kuuskankare (2002). PWGL: A Novel Visual Language based on Common Lisp, CLOS and OpenGL. In *Proceedings of International Computer Music Conference*, Göteborg, Sweden.

Laurson, M. and M. Kuuskankare (2003). From RTM-notation to ENP-score-notation. In *Journées d'Informatique Musicale (JIM)*, Montbéliard, France.

Laurson, M. and M. Kuuskankare (2005). Extensible Constraint Syntax Through Score Accessors. In *Journées d'Informatique Musicale*, Paris.

Lerdahl, F. and R. Jackendoff (1983). *A Generative Theory of Tonal Music*. MIT Press.

Löthe, M. (1999). Knowledge Based Automatic Composition and Variation of Melodies for Minuets in Early Classical Style. In W. Burgard, T. Christaller, and A. B. Cremers (Eds.), *KI-99: Advances in Artifical Intelligence: 23rd Annual German Conference on Artificial Intelligence*, Volume 1701 of *Lecture Notes in Computer Science*. Springer.

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

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

Messiaen, O. (1944). *The Technique of my Musical Language*. Paris: Alphonse Leduc. Trans. John Satterfield.

Miranda, E. R. (2001). *Composing Music with Computers*. Focal Press.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.

Monzo, J. (2005). Encyclopedia of Microtonal Music Theory. `http://tonalsoft.com/enc/encyclopedia.aspx` (accessed 24 May 2006).

Morales, E. and R. Morales (1995). Learning Musical Rules. In *Proceedings of the IJCAI-95 International Workshop on Artificial Intelligence and Music, 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, Canada.

Morris, R. and D. Starr (1974). The Structure of All-Interval Series. *Journal of Music Theory 18*(2).

Motte, D. d. l. (1976). *Harmonielehre*. Bärenreiter-Verlag.

Motte, D. d. l. (1981). *Kontrapunkt*. Bärenreiter-Verlag.

Motte, D. d. l. (1993). *Melodie*. dtv/Bärenreiter.

Müller, T. (2001). *Constraint Propagation in Mozart*. Ph. D. thesis, Naturwissenschaftlich-Technische Fakultät I der Universität des Saarlandes.

Müller, T. (2004). *Problem Solving with Finite Set Constraints in Oz. A Tutorial* (1.3.1 ed.). Mozart Consortium. `http://www.mozart-oz.org/documentation/fst` (accessed 25 May 2006).

Nienhuys, H.-W. and J. Nieuwenhuizen (2003). Lilypond, a System for Automated Music Engraving. In *Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*, Firenze, Italy.

Norvig, P. (1992). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgen Kaufmann Publishers.

Ovans, R. and R. Davison (1992). An Interactive Constraint-Based Expert Assistant for Music Composition. In *Ninth Canadian Conference on Artificial Intelligence*, University of British Columbia, Vancouver.

Pachet, F. (1993). An Object-Oriented Representation of Pitch-Classes, Intervals, Scales and Chords: The basic MusES. Technical report, LAFORIA-IBP-CNRS, Universite Paris VI.

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

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

Pachet, F., G. Ramalho, J. Carrive, and G. Cornic (1996). Representing Temporal Musical Objects and Reasoning in the MusES System. *Journal of New Music Research 5*(3).

Pachet, F. and P. Roy (1995). Mixing Constraints and Objects: a Case Study in Automatic Harmonization. In I. Graham, B. Magnusson, and J.-M. Nerson (Eds.), *Proceedings of TOOLS-Europe'95, Versailles, France*. Prentice-Hall.

Pachet, F. and P. Roy (1998). Formulating Constraint Satisfaction Problems on Part-Whole Relations: The Case of Automatic Musical Harmonization. In *ECAI 98 Workshop on Constraints for Artistic Applications*, Brighton.

Pachet, F. and P. Roy (2001). Musical Harmonization with Constraints: A Survey. *Constraints Journal 6*(1).

Pachet, F., P. Roy, and D. Cazaly (2000). A Combinatorial Approach to Content-Based Music Selection. *IEEE MultiMedia 7*(1).

Papadopoulos, G. and G. Wiggins (1999). AI Methods for Algorithmic Composition: A Survey, a Critical View and Future Prospects. In *Proceedings of the AISB'99 Symposium on Musical Creativity*.

Parrow, J. (2001). An Introduction to the π-Calculus. In J. A. Bergstra, A. Ponse, and S. A. Smolka (Eds.), *Handbook of Process Algebra*. Elsevier.

Partch, H. (1974). *Genesis of a Music: An Account of a Creative Work, Its Roots and Its Fulfillments* (2nd ed.). DaCapo Press.

Phon-Amnuaisuk, S. (2001). *An Explicitly Structured Control Model for Exploring Search Space: Chorale Harmonisation in the Style of J.S. Bach*. Ph. D. thesis, Centre for Intelligent Systems and their Application, Division of Informatics, University of Edinburgh.

Phon-Amnuaisuk, S. (2002). Control language for harmonisation process. In C. Anagnostopoulou, M. Ferrand, and A. Smaill (Eds.), *Music and Artificial Intelligence: Second International Conference, ICMAI 2002*, Volume 2445 of *Lecture Notes in Computer Science*. Springer-Verlag.

Piston, W. (1947). *Counterpoint*. Norton.

Pope, S. T. (Ed.) (1991). *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology*. MIT Press.

Pope, S. T. (1992). The Smoke Music Representation, Description Language, and Interchange Format. In *Proceedings of the International Computer Music Conference*, San Jose.

Prusinkiewicz, P. and A. Lindenmayer (1990). *The Algorithmic Beauty of Plants*. Springer-Verlag.

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

Puckette, M. (2002). Max at Seventeen. *Computer Music Journal 26*(4).

Rameau, J. P. (1984, orig. 1722). *Treatise on Harmony (Traité de l'harmonie réduite à ses principes naturels)*. Dover. translated, with and introduction and notes, by Philip Gossett.

Ramirez, R. and J. Peralta (1998). A constraint-based melody harmonizer. In *ECAI 98 Workshop on Constraints for Artistic Applications*, Brighton.

Raymond, E. S. (2003). *The Art of Unix Programming*. Addison-Wesley Professional.

Recordare LLC (2006). MusicXML Definition. `http://www.recordare.com/xml.html` (accessed 24 May 2006).

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

Roy, P. and F. Pachet (1997). Reifying Constraint Satisfaction in Smalltalk. *Journal of Object-Oriented Programming 10*(4).

Rueda, C., G. Alvarez, L. O. Quesada, G. Tamura, F. D. Valencia, J. F. Díaz, and G. Assayag (2001). Integrating Constraints and Concurrent Objects in Musical Applications: A Calculus and its Visual Language. *Constraints 6*(1).

Rueda, C., M. Lindberg, M. Laurson, G. Block, and G. Assayag (1998). Integrating Constraint Programming in Visual Musical Composition Languages. In *ECAI 98 Workshop on Constraints for Artistic Applications*, Brighton.

Rueda, C., G. Tamura, and L. O. Quesada (1997). The Visual Model of Cordial. In *XXIII Conferencia Latinoamericana de Informática (CLEI'97)*, Valparaíso, Chile.

Rueda, C. and F. D. Valencia (1997). Improving Forward Checking with Delayed Evaluation. In *XXIII Conferencia Latinoamericana de Informática (CLEI'97)*, Valparaíso, Chile.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen (1991). *Object-oriented Modelling and Design*. Prentice-Hall.

Russell, S. J. and P. Norvig (2002). *Artificial Intelligence: A Modern Approach* (2nd ed.). Prentice Hall.

Sandred, O. (2000a). *OMRC 1.1. A library for controlling rhythm by constraints* (2nd ed.). Paris: IRCAM.

Sandred, O. (2000b). *OMRC library Tutorial. version 1.1* (2nd ed.). Paris: IRCAM.

Sandred, O. (2003). Searching for a Rhythmical Language. In *PRISMA 01*. Milano: EuresisEdizioni.

Sandred, O. (2004). Interpretation of everyday gestures – composing with rules. In *Proceedings of the 2004 Music and Music Science Conference*, Stockholm.

Schenker, H. (1956, orig. 1935). *Der freie Satz*. Universal Edition.

Schoenberg, A. (1943). *Models for Beginners in Composition*. Los Angeles: Belmont. Ed. Leonhard Stein.

Schoenberg, A. (1964). *Preliminary Exercises in Counterpoint*. New York: St. Martin's Press. Ed. Leonard Stein.

Schoenberg, A. (1967). *Fundamentals of Musical Composition*. London: Faber and Faber. Ed. Gerald Strang and Leonard Stein.

Schoenberg, A. (1979). *Grundlagen der musikalischen Komposition*. Universal Edition.

Schoenberg, A. (1986, orig. 1911). *Harmonielehre* (7th ed.). Wien: Universal Edition. Rev. ed. 1922. Trans. by Roy Carter as *Theory of Harmony*. Berkeley and Los Angeles: University of California Press. 1978.

Schoenberg, A. (1995). *The Musical Idea and the Logic, Technique, and Art of Its Presentation*. New York: Columbia University Press. Ed., trans., and with a commentary by Patricia Carpenter and Severine Neff.

Schoenberg, A. (1999, orig. 1969). *Structural Functions of Harmony* (Second revised ed.). Faber and Faber. Ed. Leonard Stein.

Schottstaedt, B. CLM. `http://ccrma.stanford.edu/software/snd/snd/clm.html` (accessed 25 May 2006).

Schottstaedt, B. (1997). Common Music Notation. In E. Selfridge-Field (Ed.), *Beyond MIDI. The Handbook of Musical Codes*. MIT press.

Schottstaedt, W. (1989). Automatic Counterpoint. In M. V. Mathews and J. R. Pierce (Eds.), *Current Directions in Computer Music Research*. The MIT Press.

Schulte, C. (1997). Oz Explorer: A Visual Constraint Programming Tool. In L. Naish (Ed.), *Proceedings of the Fourteenth International Conference on Logic Programming*, Leuven, Belgium. The MIT Press.

Schulte, C. (2002). *Programming Constraint Services*, Volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

Schulte, C. and G. Smolka (2004). *Finite Domain Constraint Programming in Oz. A Tutorial* (1.3.1 ed.). Mozart Consortium. `http://www.mozart-oz.org/documentation/fdt` (accessed 25 May 2006).

Selfridge-Field, E. (Ed.) (1997). *Beyond MIDI. The Handbook of Musical Codes*. MIT press.

SICS. SICStus Prolog. `http://www.sics.se/isl/sicstuswww/site/index.html` (accessed 24 May 2006).

Siskind, J. M. (1991). Screaming yellow zonkers. Technical report. Draft of 29 September 1991.

Siskind, J. M. and D. A. McAllester (1993). Screamer: A Portable Efficient Implementation of Nondeterministic Common Lisp. Technical Report IRCS-93-03, University of Pennsylvania Insitute for Research in Cognitive Science.

Sloan, D. (1997). HyTime and Standard Music Description Language: A Document-Description Approach. In E. Selfridge-Field (Ed.), *Beyond MIDI. The Handbook of Musical Codes*. MIT press.

Smaill, A., G. Wiggins, and M. Harris (1993). Hierarchical Music Representation for Composition and Analysis. *Computing and the Humanities Journal 27*(1).

Smolka, G. (1995). The Oz Programming Model. In J. van Leeuwen (Ed.), *Computer Science Today: Recent Trends and Developments*, Volume 1000 of *Lecture Notes in Computer Science*. Springer-Verlag.

Supper, M. (2001). A Few Remarks on Algorithmic Composition. *Computer Music Journal 25*(1).

Tack, G. and D. L. Botlan (2005). Compositional Abstractions for Search Factories. In P. V. Roy (Ed.), *Multiparadigm Programming in Mozart/OZ: Second International Conference, MOZ 2004*, Lecture Notes in Computer Science 3389. Springer-Verlag.

Taube, H. (1991). Common Music: A Music Composition Language in Common Lisp and CLOS. *Computer Music Journal 15*(2).

Taube, H. (1993). Stella: Persistent Score Representation and Score Editing in Common Music. *Computer Music Journal 17*(4).

Taube, H. (1997). An Introduction to Common Music. *Computer Music Journal 21*(1).

Taube, H. (2004). *Notes from the Metalevel*. Swets & Zeitlinger Publishing.

Taube, H. (2005). Common Music Dictionary. `http://commonmusic.sourceforge.net/doc/dict/` (accessed 24 May 2006).

Truchet, C. Backtrack Modules. `http://www.ircam.fr/equipes/repmus/OpenMusic/Documentation/OMUserDocumentation/DocFiles/Reference/backtrack/` (accessed 25 May 2006).

Truchet, C. OMBacktrack Tutorial. `http://www.ircam.fr/equipes/repmus/OpenMusic/Documentation/OMUserDocumentation/DocFiles/Reference/backtracktutorial/` (accessed 25 May 2006).

Truchet, C., C. Agon, and P. Codognet (2001). 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.

Truchet, C., G. Assayag, and P. Codognet (2001). Visual and Adaptive Constraint Programming in Music. In *Proceedings of International Computer Music Conference 2001*, Havana, Cuba.

Truchet, C., G. Assayag, and P. Codognet (2003). OMClouds, a heuristic solver for musical constraints. In *MIC2003: The Fifth Metaheuristics International Conference*, Kyoto, Japan.

Tsang, C. P. and M. Aitken (1991). Harmonizing music as a discipline of constraint logic programming. In *Proceedings of then International Computer Music Conference*, Montréal.

Ungeheuer, E. (Ed.) (2002). *Elektroakustische Musik*. Handbuch der Musik im 20. Jahrhundert, Volume 5. Laaber.

van Roy, P. and S. Haridi (2004). *Concepts, Techniques, and Models of Computer Programming*. MIT Press.

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

Wiggins, G., E. Miranda, A. Smaill, and M. Harris (1993). A Framework for the Evaluation of Music Representation Systems. *Computer Music Journal 17*(3).

Wikipedia contributors (2005). Diamond problem. In *Wikipedia, the Free Encyclopedia*. `http://en.wikipedia.org/wiki/Diamond_problem` (accessed 12 December 2005).

Wolkow, A. (2005, Russian orig. 1939). *Der Zauberer der Smaragdenstadt*. Leiv Buchhandels- und Verlagsanstalt.

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

Zhong, N. and Y. Zheng (2004). Constraint-Based Melody Representation. In U. K. Wiil (Ed.), *Computer Music Modeling and Retrieval: Second International Symposium, CMMR 2004*, Volume 3310 of *Lecture Notes in Computer Science*. Springer-Verlag.

Zils, A. and F. Pachet (2001). Musical Mosaicing. In *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-01)*, Limerick, Ireland.

Zimmermann, D. (2001). Modelling Musical Structures. *Constraints 6*(1).