Algorithms and Data Structures for a Music Notation System based on GUIDO Music Notation

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades Dr.-Ing.

vorgelegt von Dipl.-Inform. Kai Renz, geb. Flade
geboren in Darmstadt

Tag der Einreichung: 26. August 2002
Tag der mündlichen Prüfung: 21. Oktober 2002

Referenten:
Prof. Dr. Hermann Walter, Darmstadt
Prof. Dr. Keith Hamel, Vancouver, Kanada

Darmstädter Dissertationen D17

Acknowledgment

First of all, I would like to thank Prof. H. K.-G. Walter for his support. Without him, research in the field of computer music would not have started at the computer science department of the Darmstadt University of Technology.
I would also like to express my thanks to Holger H. Hoos and Keith Hamel for their fundamental work on GUIDO Music Notation and for countless discussions on the subject of music representation and music notation. Special thanks also to Jürgen Kilian for many fruitful conversations and encouragement. Additionally, I would like to thank Marko Görg, Matthias Huber, Martin Friedmann, and Christian Trebing for working on different aspects of GUIDO. Parts of this thesis would not have been possible without their work.
Many thanks go to the colleagues at the "Automata Theory and Formal Languages" group of the computer science department of the Darmstadt University of Technology, who have always been extremely helpful and open to discuss the sometimes unfamiliar subjects of computer music.
Last but not least, I wish to express my love and gratitude to my wife for her trust in my work.

Abstract

Many aspects of computer-based music notation have been previously discussed. The presented thesis deals with the process of converting a textual description of music into a conventional musical score that can be either printed or viewed on a computer screen.

The chosen textual music representation language is GUIDO Music Notation, an adequate, human-readable format, which has been developed since 1996. Because GUIDO Music Notation is not solely focused on score representation but rather on being able to represent all logical aspects of *music*, the conversion of arbitrary GUIDO descriptions into a conventional score is not necessarily an easy or straightforward task.

The thesis begins by introducing the three-layered structure of GUIDO Music Notation. The most important features of GUIDO are presented and compared to other music representation languages. The thesis continues to describe how GUIDO descriptions are first converted into a suitable computer representation. Then, automatic musical typesetting algorithms work on this inner representation. The major difference to other music notation systems lies in the fact that all implemented typesetting algorithms are described as GUIDO to GUIDO transformations. Each step of the musical typesetting process adds more richness to the GUIDO description. The final description, which contains *all* typesetting information, is then used for viewing and printing the score.

In this thesis, the implemented music notation algorithms are described, some of them in detail; especially those parts of the musical typesetting process that have similarities to text setting, namely spacing and line breaking, are presented. An improved algorithm for spacing a line of music is discussed and it is shown how the resulting spacing more closely matches the spacing of a human engraver. Additionally, a new algorithm for optimally filling pages is presented for the first time.

The implemented music notation system is used within several applications including an online music notation server which is freely available on the Internet using any standard web browser.

Contents

"In general, a study of the representation of musical notation induces a profound respect for its ingeniousness. Graphical features have been used very cleverly to convey aspects of sound that must be accommodated in the realization of a single event." [SF97a], p. 11

Chapter 1

Introduction

"Sound waves are longitudinal mechanical waves. [...] There is a large range of frequencies within which longitudinal mechanical waves can be generated, sound waves being confined to the frequency range which can stimulate the human ear and brain to the sensation of hearing."[1] This physical description of sound waves also describes how musical information is usually transmitted: human music perception is always a perception of sound waves. One way to record or store musical information is therefore the recording of sound waves using microphones. Replaying the recorded sound waves re-creates the original musical information. Because recording sound waves has been technically possible only for around 125[2] years, other ways of recording musical information were invented earlier and are still used today. The earliest and easiest way to transmit and store musical information is to memorize it: a "teacher" sings or plays the music, a "pupil" learns it and can then play or sing the music by heart. This form of oral transmission of musical information is used all over the world everyday and works very well not only for simple musical structures.[3] This approach has several disadvantages: The "teacher" must be physically present in order to teach the music. As the musical structure gets more complicated (if, for example, other voices or instruments are added), the oral transmission may become impossible. Because of these drawbacks, people have attempted to *write* down music beginning in ancient times [Blu89a]. Obviously, the musical information contained in written scores is different from recorded sound waves: Any *performance* of a written score will sound slightly different depending on the interpretation of the performer. The information contained in a graphical score transports the essential musical ideas, but leaves room for interpretation and ambiguousness.

Only because of the existence of written music, the traditional roles of the *composer* and the *musician* (or *performer*) have come into existence. When recording of

[1] Quote taken from [HR78], p. 433.

[2] Edison invented the gramophone in 1877.

[3] As an example consider children learning children's songs. In some cultures, extremely complex music is all transmitted orally.

sound waves became possible, these roles were beginning to loose importance. Only because music could be recorded and therefore easily distributed, performers (for example jazz musicians) were beginning to become as popular as previously only the composers were. Today, someone can be the composer and single performer of a piece at the same time; for these musicians, the need to produce a graphical score is of little importance. Still, a large number of composers are producing scores today, which are then interpreted and performed by musicians. Therefore, transmitting musical information using graphical scores is an important subject.

Conventional music notation as it is used today has evolved over a long period of time, beginning well before the invention of printing techniques. One of the most important contributors to todays music notation is Guido d'Arezzo ($\star$ around 992, † around 1050), who was the first to place notes on and between *lines*, which form the staves of a score known today [Blu89b]. This was the birth of the two-dimensional structure for graphical music representation that is used until now.[4] As the complexity of musical compositions grew and new instruments were invented, music notation evolved continously to adequately represent the new ideas. The techniques for producing scores also changed over time. As printing techniques were invented around 1500, they were also used and adapted for music notation.

Guido d'Arezzo

The printing process for creating scores that was used from around 1600 until beyond the invention of computer driven printers is called *engraving* [Blu89c, eng98].[5] Here, the (human) engraver uses a copper plate to "engrave" the score into the soft metal plate using specialized tools. The quality and readability of the score greatly depends on the experience of the human engraver. Very often, music notation is ambiguous, as there exist multiple ways to represent a musical idea graphically: "Musical notation is not logically self-consistent. Its visual grammar is open-ended – far more so than any numerical system or formal grammar."[6] Because of the complexity of the task, some books on the art of music engraving have been written [Had48, Rea79, Ros87, Vin88, Wan88], which give detailed instructions on how to engrave music. These instructions are usually derived by studying and extracting examples from "real" scores, which were engraved by an expert human engraver. Nevertheless, for almost any typesetting rule, there seems to exist at least one score,

[4] As shown in Figure 1.1, the pitch of a note is indicated by the vertical placement on and between lines of a staff. The attack time of an event is denoted by the horizontal placement on the staff. Events with the same attack time have the same horizontal position.

[5] Some music publishers are still employing human engravers today.

[6] Quote taken from [SF97a] (p. 15)

where this rule is explicitly broken in order to obtain a good overall result. This makes automatic music typesetting a complicated task.

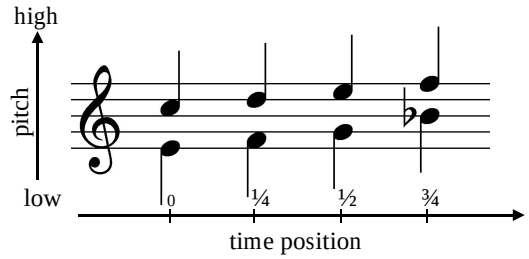


Figure 1.1: Two-dimensional structure of conventional scores

The invention of computers and high quality laser printers has dramatically changed the way musical scores are being produced today: Nowadays, almost all music typesetting[7] is done with the help of computers, but it must be noted that even though computer programs are capable of producing very fine scores, a large amount of human interaction is still required, if high quality output is desired.
The issues and problems found when automatically typesetting music are manifold. Don Byrd, who has written a dissertation on the subject, defines three main tasks that must be solved by any music notation system: *Selecting*, *Positioning*, and *Printing* the music notation symbols [Byr84]. Each of these tasks requires knowledge that was formerly held by engravers only. In order to automatically create music notation, the human knowledge must be transfered into a computer system, a task, which is always challenging and interesting. Different approaches to encode engravers knowledge into music notation systems have been tried. Most approaches use a case (or rule) based approach to automatically select and position the music notation symbols [BSN01, Gie01]. Unfortunately, there is currently little public discussion on music notation algorithms: most commercial vendors of music notation software do not publish or discuss their internal algorithms.
Even though quite a number of music notation programs are being used today, no commonly used and wide spread notation interchange format exists. Even though many proposals for such an interchange format have been made (for example NIFF or SMDL), none of them are widely used. One of the main reasons for this seems to be the complexity of any such format. Because music notation and musical structure are generally so rich, the developed formalisms are also complex. Very often, supporting such an interchange format involves enormous effort for an application developer. Additionally, some developers of commercial music notation software seem to have little interest in an unrestricted interchange of scores from one appli-

[7]Note that the word “typesetting” is borrowed from printed text; actually scores where prepared using types for a period in time, but the results were not satisfactory. Nevertheless, the term “musical typesetting” describes the process of creating a visual score (for computer display or printing).

cation to another.
The only (structured) music representation format that is widely accepted today is MIDI, which is not suited as a notation interchange format because it was never intended as such; much of the information present in a graphical score can not be represented using MIDI. Many other music representation formalism have been developed since computers are being used for music information processing. Most of these were created for a specific purpose and not for notation interchange.

Beginning in 1996, a group of developers began developing a new music representation language called GUIDO Music Notation[8] which is named after Guido d'Arezzo [HHRK98]. GUIDO is a human readable, adequate music representation language that is not primarily focused on conventional music notation; it is much more an open format, which is capable of storing musical, structural and notational information. One of the main design criteria for GUIDO was *adequateness*, meaning that a simple musical idea should have a simple GUIDO representation, whereas more complex ideas may sometimes require more complex GUIDO representations. GUIDO is also *intuitive* to read because commonly known musical names are used: it is easy to understand that the GUIDO description `[ \clef<"treble"> c d e ]` describes a treble-clef and three consecutive notes c, d, and e. Despite of the intuitive character of simple GUIDO descriptions, it can very well be used for representing complex musical ideas. It was shown in [HHR99] that GUIDO is also very well suited as a notation interchange format.

Because of its features, an increasing number of research groups and individuals began to use GUIDO Music Notation as the underlying music representation formalism in various music related projects. In this context, the need for an easy to use music notation system that converts GUIDO descriptions into conventional graphical scores became quite strong; this demand led directly to the design and implementation of a music notation system based on GUIDO, which is being described in the presented thesis. Other music notation software has been adapted to import and export GUIDO [Ham98]. Even though GUIDO descriptions may contain all necessary formatting information, the general process of converting an *arbitrary* GUIDO description into a conventional score is not trivial: Because a GUIDO description can be highly unspecified, meaning that no explicit formatting information is contained, the missing information must be generated automatically. Additionally, the GUIDO description may contain notes or rests with unconventional durations, for which no standard graphical representation is possible. These cases must be dealt with by a set of music notation algorithms, which must first be recognized, defined, and implemented. It should also be possible to describe each of these music notation algorithms as a GUIDO to GUIDO transformation. This means that each algorithm gets a GUIDO description as its input and returns an enhanced GUIDO description, where well defined formatting aspects have been automatically added.

[8] In the following, the terms GUIDO Music Notation and GUIDO are used in parallel. Both terms describe the format as well as specific files. It will be clear from the context, what is meant.

This concept greatly simplifies the exchange of music notation algorithms and is also very helpful for generally discussing the issues found in automatic music notation systems.
Because GUIDO descriptions are human-readable plain text, any application supporting GUIDO requires an internal, computer-suited representation for storing, manipulating, and traversing the musical data. As part of this thesis an object-oriented class library, called Abstract Representation, was designed and implemented as such an internal representation for GUIDO descriptions. The GUIDO parser kit [Hoo] is used to convert a GUIDO description into one instance of the Abstract Representation. Because the general structure of the Abstract Representation closely matches the internal structure of GUIDO, the conversion process is straight-forward. As the Abstract Representation is an object-oriented data structure, the required manipulation and traversal functions are an integral part of the framework. These manipulation functions, which are mostly used by music notation algorithms, include operations for deleting or splitting events or adding musical markup.
In order to create a graphical score, an additional object-oriented class library, called the Graphical Representation was developed. Each notational element found in a conventional score has a direct counterpart in the Graphical Representation. Two steps are then required to create a conventional score from an arbitrary GUIDO description: first, the GUIDO description must be converted into the Abstract Representation, which must then be converted into the Graphical Representation. Once the notational elements of the Graphical Representation have been determined, they must be placed on lines and pages. This requires sophisticated algorithms for spacing, line breaking and page filling. All of these issues have been thoroughly dealt with during work on this thesis. The spacing algorithm used in most current music notation system, which has been previously discussed in the literature [Gou87], has been enhanced to provide better results for complex interaction of rhythms. The line breaking algorithm has been extended to be usable for obtaining optimally filled pages, which is a standard requirement for graphical scores. There have been no previous publications on the subject of page filling, although one commercial music notation system includes an implementation of such an algorithm.
The stand-alone music notation system that has been developed using the data structures and algorithms described above is freely available and is being used by a growing number of people and institutions around the world. The implemented music notation system is also used in other areas: the GUIDO NoteServer is a free Internet service that converts GUIDO descriptions into pictures of scores. This service can be accessed using any standard web browser. Another part of research which was carried out while working on this thesis focused on the development of a Java based music notation editor.
The rest of this thesis is structured as follows: In the first part of chapter 2, GUIDO Music Notation is thoroughly described and numerous examples for the various features of GUIDO are given; several sections of this part of the chapter are enhanced

versions of previous publications [HHRK01, HHR99, HHRK98]. In the second part of chapter 2, GUIDO is compared to the following other music representation formalism in use today: MIDI, DARMS, MuseData, SCORE, Common Music Notation, Plaine and Easie Code, MusiXTEX, LilyPond, NIFF, and SGML based formats. Where appropriate, examples contrasting GUIDO and the compared formats are given. In chapter 3 an internal, computer-suited representation for GUIDO called Abstract Representation is introduced. It is shown, how this object-oriented class library matches the internal structure of GUIDO, and how a GUIDO description can be converted into it. The chapter closes with describing the required manipulation and traversal functions for the developed data structure. In chapter 4 the conversion of arbitrary GUIDO descriptions into conventional scores is described in detail. As was already mentioned above, the necessary music notation algorithms can be described as GUIDO to GUIDO transformations. Additionally, the Graphical Representation is introduced and the enhanced algorithms for spacing, line breaking and optimal page fill are elucidated. Parts of this chapter are currently being published in [Ren02]. Chapter 5 shows, which applications have been developed using the implemented music notation data structures and algorithms. These include a stand-alone NoteViewer and online applications for music notation. Additionally, a prototypical Java based interactive music notation editor is described briefly, and the requirements for a music notation editor are discussed. Most of the content of chapter 5 has been published previously [RH98, Ren00, RH01, HRG01]. Finally, chapter 6 gives an overview of the presented thesis and points out possibilities for further research projects.

Chapter 2

The GUIDO Music Notation Format

Introduction

In this chapter, GUIDO Music Notation,[1] a music representation language which has been developed since 1996 [HH96], will be introduced and compared to other representation formalisms. GUIDO Music Notation is named after Guido d'Arezzo (ca. 992-1050), a renowned music theorist of his time and important contributor to today's conventional musical notation [Blu89b]. His achievements include the perfection of the staff system for music notation and the invention of solmization (solfege). GUIDO is *not* primarily focused on conventional music notation, but has been invented as an open format, capable of storing musical, structural, and notational information. GUIDO has been split into three consecutive layers: *Basic* GUIDO introduces the main concepts of the GUIDO design and allows to represent much of the conventional music of today. *Advanced* GUIDO extends *Basic* GUIDO by adding exact score-formatting and some more advanced musical concepts. Finally, *Extended* GUIDO can represent user-defined extensions, like microtonal information or user defined pitch classes.

GUIDO Music Notation is designed as a flexible and easily extensible open standard. In particular, its syntax does not restrict the features it can represent. Thus, GUIDO can be easily adapted and customized to cover specialized musical concepts as might be required in the context of research projects in computational musicology. More importantly, GUIDO is designed in a way that when using such custom extensions, the resulting GUIDO data can still be processed by other applications that support GUIDO but are not aware of the custom extensions, which are gracefully ignored. This design also greatly facilitates the incremental implementation of GUIDO support in music software, which can speed up the software development process significantly, especially for research software and prototypes.

[1] In the following, the terms GUIDO Music Notation and GUIDO are used in parallel. Both terms describe the format as well as specific files. It will be clear from the context, what is meant.

GUIDO has not been developed with a particular type of application in mind but to provide an adequate[2] representation formalism for score-level music over a broad range of applications. The application areas include notation software, compositional and analytical systems and tools, musical databases, performance systems, and music on the web.

This chapter introduces the most important musical features of GUIDO Music Notation and then compares GUIDO to other music representation languages, namely the MIDI File Format, DARMS, MuseData, SCORE, Common Music Notation, Plaine and Easie Code, MusiXTEX, LilyPond, NIFF, SMDL, and XML-based representations.

2.1 Basic GUIDO Music Notation

Generally, GUIDO Music Notation is designed to adequately represent musical material, which can then be interpreted by computer music software and also be used for conventional music notation. The *Basic* GUIDO layer covers the relevant aspects for representing "simple" music, from primitive musical objects such as single notes or rests, to complete pieces with multiple voices, dynamics and other markings, as well as text. There is no inherent restriction on the length or complexity of pieces specified in *Basic* GUIDO (although applications supporting GUIDO might impose such restrictions). Rather, the limits of *Basic* GUIDO lie in the features it allows to express — focusing on basic musical concepts; it does not cover, e.g., the exact placement and graphical appearance of musical objects (which can be specified in the next layer, *Advanced* GUIDO). The advantage of this layered concept is the ability to adjust the level of encoding to the applications need. Only if exact score formatting information is really needed, this information *can* be encoded.

In general, GUIDO Music Notation is based on two basic syntactic elements: events and tags. An *event* is a musical entity which has a duration[3] (e.g. notes and rests) whereas *tags* are used to define musical attributes (such as e.g. a meter, a clef, a key, or a slur). Figure 2.1 shows a simple example of *Basic* GUIDO containing a single voice with some musical markup and a couple of notes.

[2]The notion of *adequateness* is a major design criteria of GUIDO. Synonyms for the term "adequate" are: decent, acceptable, all right, common, satisfactory, sufficient, unexceptionable, unexceptional, unimpeachable, unobjectionable. From Merriam-Webster's Collegiate Dictionary: adequate means "sufficient for a specific requirement". GUIDO is an adequate representation format, because simple musical concepts can be represented in a simple way; only complex musical notions may require more complex representations.

[3]Please note that the specified duration may be zero; an *event* with duration zero can be helpful in exact score formating, as will be shown in later chapters.

```
[ \clef<"treble"> \key<"D"> \meter<"4/4">
  a1*1/2 b a/4. g/8 f#/4 g a/2 b a/4. g/8 f#/4 g
  a/2 a b c#2/4 d c#/2 b1 a/1 ]
```

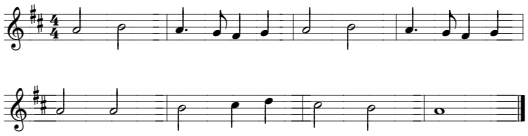


Figure 2.1: A simple *Basic* GUIDO example

2.1.1 Notes and Rests

In GUIDO, notes are represented by their names (eg. `c`, `d`, `e`, `do`, `re`, `mi`, ...) followed by optional additional parameters: accidentals, register and duration (including dots). By leaving out these parameters, incompletely specified music can be represented in GUIDO. A complete note description in GUIDO has the following form: "`c#1*1/4.`" which specifies the dotted quarter note c-sharp in the first octave (just above the middle C).[4] Note that register and duration information is carried over from one note to the next within note sequences. This *carry feature* can also be found in other music representation formats, which are described later in this chapter.

As GUIDO has been developed for a broad range of musical applications, different systems of diatonic note names are supported. Different types of note names may be mixed freely in a GUIDO description: "do" and "c" are equivalent, the same is true for "si", "h" and "b".[5] In addition to the well-known pitch classes, GUIDO also provides chromatic pitch-classes: There is, for example, a pitch-class called `cis` that is different from `c#` and `d&` (where the # denotes a sharp, the & denotes a flat), even though when played on a regular MIDI device all sound just the same. This concept is important to cover aspects of 12-tone-music, where twelve distinguished pitch classes of equivalent importance are used.

Rests are represented like notes; instead of a note name, an underscore is used: '`_*1/4`' is a quarter rest; alternatively, the special note name '`rest`' can be used. Obviously, rests don't have accidentals or register information; rest durations are specified as for notes. There is another "special" rest, which is denoted with the note name '`empty`'. An '`empty`' rest is not represented graphically in the score; it is used to represent music, where voices appear and disappear visually. This is an important feature, especially for exact formatting of scores.

2.1.2 Sequences and segments

GUIDO provides two orthogonal constructs for grouping notes and rests into larger

[4] The `*` between the register and the duration is needed to distinguish between register and duration information. If no `*` appears after the note, the first number is interpreted as the register.

[5] The "h" is the German name for "b"; the name "b" in German stands for b-flat. This is the reason why the composer J. S. Bach can compose his last name as music: the equivalent GUIDO description is `[ b& a c h ]`, which could also be written as `[ h& a c h ]`.

structures: *sequences* and *segments*. A *sequence* describes a series of temporally consecutive musical objects, whereas a *segment* describes a number of simultaneous musical objects. The most simple form of a segment is a chord where the separate voices consist of single notes.

Sequences and segments are realized in GUIDO using the following syntax: A sequence begins with a square bracket '[' and ends with the corresponding closing bracket ']'. Segments are enclosed in curly braces '{ ... }'. Musical objects within segments are separated by commas. Within sequences and chords, octaves and durations are implied from the previous note, if they are not explicitly specified.

GUIDO's two grouping constructs have not only been chosen for reasons of formal simplicity and elegance, they are also motivated by the fact that essentially most musical pieces can be represented in two fundamentally different ways: As a sequence of chords (chord-normalform) or as a set of simultaneous voices (poly-normal form). In the former case, the music is described as a sequence of chords (which is a sequence of segments), while in the latter case it is represented by a set of sequences (a segment of sequences). The usage of either one of these normalforms for the description of a piece depends heavily on the style of the piece: homophonic choral music can often very naturally be written in chord-normalform, whereas a fugue is typically more adequately described using poly-normalform. *Basic* GUIDO facilitates both normalforms by supporting an extended poly-normalform which allows chords within sequences of a segment (i.e. chords within polyphonic voices). Figure 2.2 shows, how a musical fragment[6] can be represented in GUIDO using either multiple voices (which is poly-normalform) or using a single voice with chords (which is chord-normalform). One detail needs to be pointed out: in the poly-normalform of Figure 2.2 there are two eighth-notes in the second bar of the third voice, which are omitted in the chord-normalform. The reason for that is that in a GUIDO chord, all notes must have the same duration.[7] Therefore, the two eighth-notes can not be put in a chord together with single quarter-notes. *Extended* GUIDO allows more complex nestings of sequence and segment constructors (see Sec. 2.3).

The graphical scores of Figure 2.2 are provided to visually describe the difference between the two normalforms; it is important to understand that poly-normalform does not require each voice to be placed on an individual staff-line. GUIDO offers simple mechanisms to place the individual voices on one common staff-line. Sometimes the choice between poly- or chord-normalform is difficult; in many cases, poly-normalform offers more freedom when exact score formatting is required.

[6] These are the first three bars from the Bach Choral "Wie wunderbarlich" from the St. Matthew Passion.

[7] It is possible to write [{ `c*1/4, e/8, g/16` }]. The GUIDO specification deals with this, as if [{ `c*1/4, e/4, g/4` }] had been written: the longest note of a chord determines the chord duration.

```
{ [ % soprano
     \clef<"treble"> \key<"b"> \meter<"C">
     b1*1/4 | b b a# f# h c#2 d d e d \fermata( c# )  ],
  [ % alto
    \clef<"treble"> \key<"b"> \meter<"C">
     a1*1/4 | g g# f# c# b0 f#1 f# b c#2 h1 \fermata( a# ) ] ,
  [ % tenor
    \clef<"g2-8"> \key<"b"> \meter<"C">
     f#1*1/4 | e d c# a#0 e1 e d/8 e f#/4 g f# \fermata( f#) ] ,
  [ % bass
    \clef<"bass"> \key<"b"> \meter<"C">
     d#0*1/4 | e e# f# f#  g# a# b b a# b \fermata( f# ) ] }

  [ \clef<"treble"> \key<"b"> \meter<"C">
     { d#0*1/4, f#1*1/4, a, b  } |
     { e0,e1, g, b } { e#0, d1, g#, b }
     { f#0, c#1, f#, a# } { f#0, a#, c#1, f# }
     { g#0, e1, b0, b1 } { a#0, e1, f#, c#2 }
     { b0, d1, f#, d2 } { b0, f#1, b, d2 }
     { a#0, g1, c#2, e} { b0, f#1, b, d2}
     \fermata( { f#0, f#1, a#, c#2 } ) ]
```

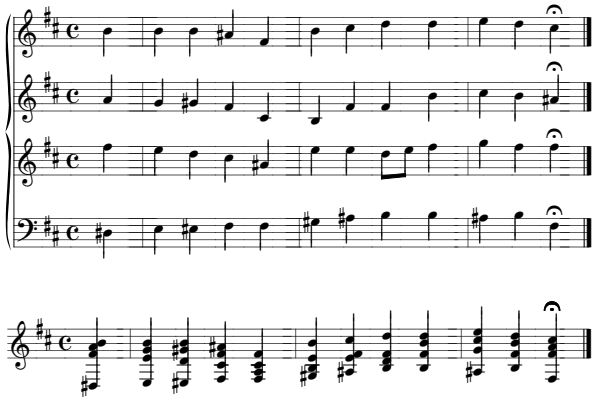


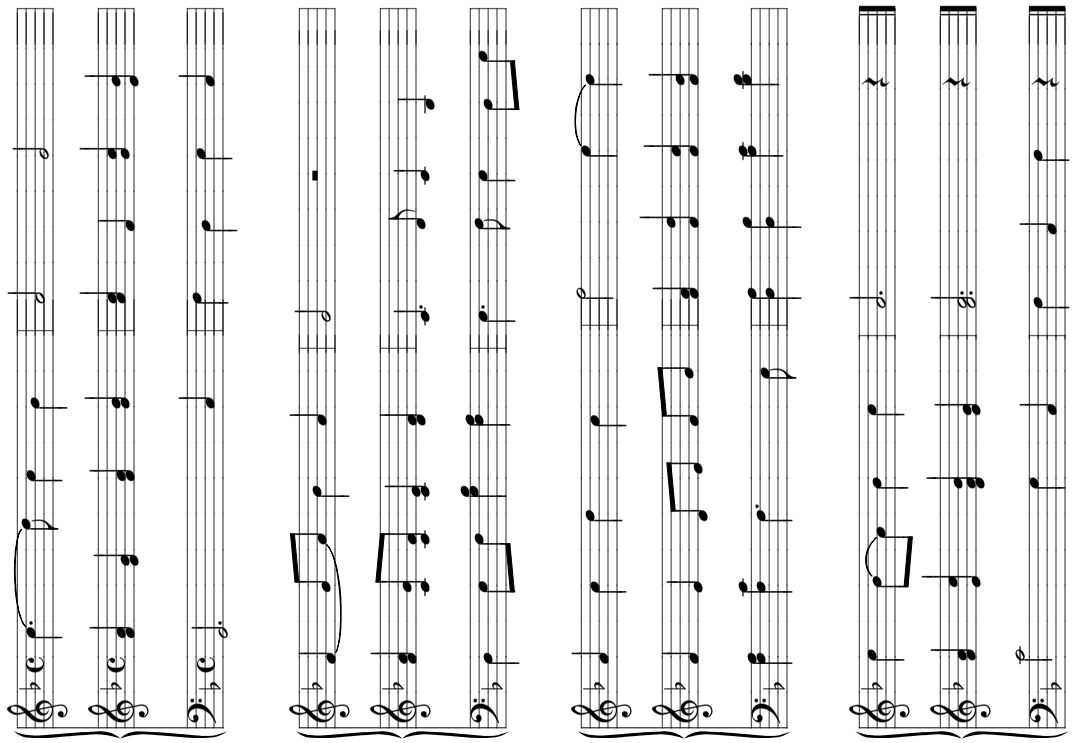
Figure 2.2: Poly- and chord-normalform of a Bach fragment

2.1.3 Tags and basic musical concepts

Written and played music does not only consist of notes and rests but also contains additional musical and graphical information. GUIDO can represent all the commonly known musical attributes by using *tags*. A *tag* has a name, optional parameters, and an optional range of application. A multitude of tags are defined in *Basic* GUIDO; for a complete description see [HH96]. Just to name a few, there are tags to describe clefs, meter, tempo, intensity, crescendo, accelerando, slurs, and more. The semantics of the tags defined in GUIDO can normally be inferred easily from their names. The example in Figure 2.1 illustrates the usage of tags in *Basic* GUIDO. Please note that the example contains no formatting information. It is up to the notation program to make good assumptions on standard formatting issues.
Syntactically, *tags* begin with a backslash followed by the name. Parameters are defined within pointed brackets ('<' and '>'); multiple arguments are separated by commas. If the *tag* has a range of application (for example a slur beginning at one note and ending at some later note), this is specified by round brackets ('(' and ')'). Figure 2.3 shows another example of *Basic* GUIDO. Here, tags are used to specify the clef, the meter and some slurs. All additional formatting information, like for example the beams, or the spacing and line breaking are automatically inferred from the original GUIDO description as they are not explicitly defined. Some additional points concerning tags should be noted:

- Usually, a number of *different* commonly used musical expressions can be used as parameter values for tags to produce the same result: this can be seen by looking at the `\clef`- and `\key`-tag of Figure 2.3. The term `\clef<"treble">` and `\clef<"violino">` both produce a treble clef. Another acceptable parameter value for the treble clef would be "g2".[8] The `\key`-tag both accepts a

[8] "g2" stands for a g-clef on the second line from the bottom. `\clef<"f4">` is an f-clef on the fourth



```
{
[ % voice 1
  \clef<"treble"> \key<"F"> \meter<"C">
  \slur(do2/4. re/8) do/4 si&1 \bar a/2 g \bar
  \sl(f/4 g/8 a) h&/4 a \bar g/2 _ | a/4 c2
    d c | f/2 \slurBegin e/4 d \slurEnd |
    c \sl( b&1/8 a ) h&/4 c2 | a1/2. _/4  ],

[ % voice 2
  \clef<"violino"> \key<-1> \meter<"C">
  {a1/4,f} {g,e} {a,f} {h&,g} | {c2,a1} f1
  {g1,c2} {h&1,e1} | {a,f} {g/8,c/8} {f,c}
  {c/4,e/4} {d,f} c/4. d/8 c/4 h&0  f1 e
  d/8  e f g {f/4,a/4} {f1,d2} {f1,c2}
  {f1,h&1} {f,a}{e1,c2}{d1,f1,h&1}{e,g}
  {f/2.,a/2.} _/4 ],

[ % voice 3
  \clef<"basso"> \key<-1> \meter<"C">
  g-1/2. c0/4  f d e c  d e/8 f
  { g/4 , h&/4 } { f , a } e/4. f/8 e/4 d/8
  e { f/4, a } { f0 , c1 }
  f0/4. e/8 { d/4, a } {d,h&} {a0,c1}
  {h&0,d1} c1/2 g0/4 c f c f _ ]
}
```

Figure 2.3: Another example of *Basic* GUIDO

number (-1) or a string “F” to produce the same key-signature (F-major, one flat).

- Abbreviations for tag names exists: the `\slur`-tag can be also specified as `\sl`. The same is true for a variety of tags (`\bm` for `\beam`, `\i` for `\intens`, and many more). For a complete set of tag-names and their abbreviations, see [HH96].

- Range tags (like the `\slur`-tag in the example) can be specified either by using round brackets *or* by using begin- and end-pairs. One of the slurs in the example of Figure 2.3 is specified using a `\slurBegin`- and a `\slurEnd`-tag. Using this mechanism, it is possible to allow a simple form of overlapping ranges. This feature is enhanced in *Advanced* GUIDO (see section 2.2.1).

- If a time signature has been set by using the `\meter`-tag, no bar lines have to be encoded explicitly. Bar lines can be set explicitly to denote an up-beat. The `\bar`-tag is equivalent to the use of “|” (using the `\bar`-tag it is also possible to explicitly set bar numbers).

2.2 Advanced GUIDO Notation

Advanced GUIDO Notation comprises some of the more advanced concepts not covered in Basic GUIDO Notation, such as glissandos, arpeggios, clusters, different types of noteheads, different types of staves, and many features from contemporary music notation. It also addresses the issue of advanced score formatting such as exact spacing and positioning of notational and graphical elements. Using *Advanced* GUIDO, it is possible to specify exact score-formatting information that can be used by any kind of professional notation software, which supports GUIDO import. This feature makes *Advanced* GUIDO an ideal candidate for a platform-and application-independent notation interchange format as was also shown in [HHR99].

2.2.1 Syntax extensions in Advanced GUIDO

Advanced GUIDO is based on the same syntax as *Basic* GUIDO with three slight extensions: named tag parameters, parameter values with units, and multiple overlapping tag ranges.

Named tag parameters

In *Advanced* GUIDO, tag parameters can be specified by using predefined names. An example for this is `\clef<type="g2",size=0.5>`, which specifies a g-clef on the second line (i.e., a treble clef) whose size is only 50% of the standard size. Note

line, which is equivalent to a standard bass clef, which could also be written as `\clef<"bass">`.

that parameter names are optional and can be omitted, as long as all required parameters are specified. Using parameter names not only increases the readability of *Advanced* GUIDO sources (which is beneficial for implementing GUIDO support), it also makes it possible to only partially specify parameters, and to assume default values for unspecified optional parameters.

Parameter values with units

Advanced GUIDO allows different measurement units to be used with tag parameters. These units, such as `cm`, `mm`, `in` (inches), `pt` (points), `hs` (halfspaces)[9], and `pc` (picas) can be optionally used with the numerical values of sizing and positioning tag parameters by specifying the unit type directly after the value. As an example, consider the tag `\pageFormat<20cm,40cm>`, which defines the width and height of a page to be 20cm and 40cm respectively.

Multiple overlapping tag ranges

Basic GUIDO already supports overlapping tag ranges by using pairs of begin- and end-tags, such as in `\slur(c1/2 \crescBegin e/8 d) g/4 \crescEnd`. This mechanism, however, does not handle the (rather rare) cases of multiple occurrences of the same tag with overlapping ranges. To support the most general case of arbitrary multiple overlapping tag ranges, *Advanced* GUIDO supports explicit disambiguation for begin- and end-tags. For every tag `\`*`id`* which can be used with a range, the tagged range `\`*`id`*`<...>( ... )` can be alternately represented as `\`*`id`*`Begin<...> ... \`*`id`*`End`. Both the `\`*`id`*`Begin` and `\`*`id`*`End` tags are used without ranges, and the `\`*`id`*`Begin` tag takes the same parameters as the original tag `\`*`id`*. To allow multiple overlapping tag ranges, the tags can be extended with a unique numeric postfix using the syntax `\`*`id`*`Begin:`n`<...> ... \`*`id`*`End:`n. Thus, three overlapping slurs can be represented as shown in Figure 2.4.

```
[ \slurBegin:1 g0*1/4 \slurBegin:2 f1
  \slurBegin:3 g2  a0 \slurEnd:1 g1
  \slurEnd:2 f2 \slurEnd:3 ]
```

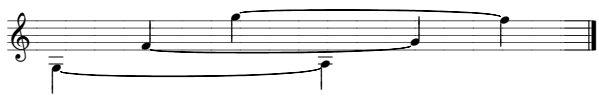


Figure 2.4: Multiple overlapping slurs

2.2.2 Exact formatting and score layout

Advanced GUIDO allows to completely specify the formatting and layout of scores. This is realized by defining some additional tags and adding a large number of new

[9] One halfspace is defined as the vertical distance between two adjacent notes, e.g. between `c` and `d`.

tag parameters to existing *Basic* GUIDO tags, which control the physical dimensions of score pages, the position and size of systems and staves, as well as the exact location of all graphical elements associated with staves. While notes, rests, most notational symbols, and text are typically associated with staves, *Advanced* GUIDO also supports graphical and text elements like, e.g., a title, which can be placed at specific locations on the page independent from staves.

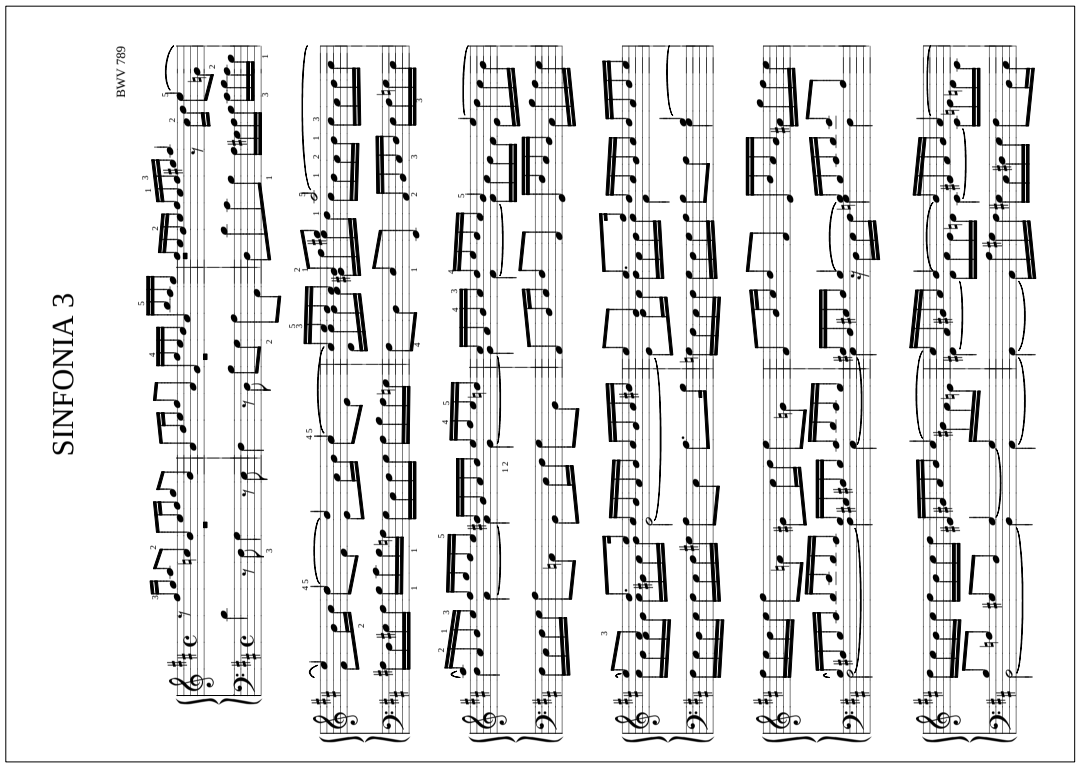
While the full set of mechanisms and corresponding tags for completely specifying score layout and formatting cannot be presented here in detail, some main concepts are highlighted while others are outlined rather briefly. A small set of tags is used to specify the physical dimension and margins of score pages (`\pageFormat`), the relative positioning of systems on a page (`\systemFormat`), type and size of staves in a system (`\staffFormat`) and their relative location (`\staff`), page and system breaks (`\newPage`, `\newSystem`), etc. Spacing and layout are based on graphical reference points, which *Advanced* GUIDO specifies for each notational element. Some elements (such as systems, or composer and title information) are positioned relative to the page, others are relative to the staves or notes they are logically associated with.

Generally, for elements associated with a staff, vertical offsets are usually specified relative to the size of the staff, using the relative unit halfspace (`hs`). Likewise, the size of many elements, such as notes or clefs, is always specified relative to the size of the staff. The most important mechanism for exactly specifying horizontal locations is the `\space` tag. Placed between two notational elements, e.g., two notes, it overrides any automatic spacing and forces the given amount of horizontal space to separate the horizontal reference positions of the elements.

Advanced GUIDO includes all tags defined in *Basic* GUIDO, but allows exact positioning and layout information to be specified using additional optional parameters. For instance, it is possible to exactly define the slope of a slur, or to change the size or type of a notehead, whenever this is needed. While automatically created *Advanced* GUIDO descriptions[10] will usually include complete layout and formatting information, in many cases this information is only partially specified. In this case, an application reading such a GUIDO description will try to infer unspecified information automatically where needed. This is the major goal of the presented thesis: Converting an "under-specified" GUIDO description into an acceptable graphical score; the required steps for this conversion process are examined in the following chapters.

If specific information is not required or supported by an application, it can easily be ignored when interpreting the GUIDO input. Note that this approach allows the adequate representation of scores: only when exact formatting information is really needed, the corresponding additional tags and parameters have to be supplied.

[10]Because *Advanced* GUIDO can be used as a notation interchange format, completely formatted scores can be saved as a complete *Advanced* GUIDO description by notation programs supporting GUIDO.




```
% The first page of a BACH Sinfonia, BWV 789
% Most of the element-positions are specified
% using Advanced GUIDO tags. The layout has
% been very closely copied from the URTEXT
% edition by the Henle-Verlag.
% This example has been prepared to show, that
% Advanced GUIDO is capable of exact score-
% formatting.
%

{ [ % general stuff
    \pageFormat<"a4", lm=0.8cm,tm=4.75cm,bm=0.1cm,rm=1.1cm>
    \title<"SINFONIA 3",pageformat="42",textformat="lc",dx=-2cm,dy=0.5cm>
    \composer<"BWV 789",dy=1.35cm,fsize=10pt>

    % indent of 1.05 cm for the first system.
    \systemFormat<dx=1.05cm>

% voice 1

    \staff<1,dy=0.85cm> % next staff is 0.85 cm below
    \staffFormat<style="5-line",size=0.9375mm>
    % barlines go through whole system
    \barFormat<style="system">
% measure 1 (voice 1)
    \clef<"treble"> \key<"D"> \meter<"C"> \stemsUp
            \restFormat<posy=-1hs> _/8 \beam( \fingering<text="3",
                dy=8hs,dx=-0.6hs,fsize=7pt>(f#2/16) g)
            \beam( \stemsUp<5hs> a/8
                \fingering<text="2",dx=1hs,dy=11.5hs,fsize=7pt>(
                \stemsUp<8hs> c ) )
            \beam( \stemsUp h1 e2/16 f#)
            \beam( \stemsUp<5hs> g/8 \stemsUp<8.5hs> h1 \stemsUp) \bar
    % measure 2 (voice 1)
    \beam( a1/8 d2/16 e) \beam( f#/8 a1)
                \beam( \stemsUp<12hs> g/16
        \stemsUp \fingering<text="4",fsize=7pt,dy=9hs,dx=-0.2hs>(f#2) e
            \stemsUp<8hs> d \stemsUp)
        \beam( \stemsUp<12hs> c#  \stemsUp
            \fingering<text="5",fsize=7pt,dy=9hs>(h) a
            \stemsUp<8hs> g \stemsUp ) \bar
    % measure 3 (voice 1)
    \beam( \stemsUp<5.5hs> f#2/16 \stemsUp e
        \fingering<text="2",fsize=7pt,dy=10hs>( d )
        \stemsUp<5.5hs> e \stemsUp)
    \beam( \stemsUp<6hs> f# \stemsUp
        \fingering<text="1",fsize=7pt,dy=11hs>(e)
        \fingering<text="3",fsize=7pt,dy=11hs>(f#)
        \stemsUp<6.5hs> g# \stemsUp)
    \stemsUp<4.75hs> a/4  \stemsUp
    \slurBegin <dx1=2hs,dy1=3hs,dx2=0hs,dy2=2hs,h=2hs>
    \fingering<text="5",fsize=7pt,dy=6hs>(  \stemsUp<4.5hs> e/4  )
        \bar \newSystem<dy=4cm>
% measure 4 (voice 1)
        \staff<1,dy=0.98cm>
        e/4 \slurEnd
    \tie<dy1=2.4hs,dx2=-1hs,dy2=2.4hs,h=1.75hs>(
                \fingering<text="4 5",dy=8hs,fsize=7pt>( \stemsUp<5hs> d) d )
                    \tieBegin<dy1=2hs,dy2=2hs,dx2=0hs>
                    .....
```

Figure 2.5: *Advanced* GUIDO example: Bach Sinfonia, BWV 789

Figure 2.5 shows an example illustrating *Advanced* GUIDO's exact formatting features. The GUIDO code includes page, system, and staff formatting as well as accurate formatting of slurs. Stem length and directions, beam groupings, and ties are also specified precisely. The complete GUIDO description for the page can be found in Appendix 6.

2.2.3 Advanced notational features

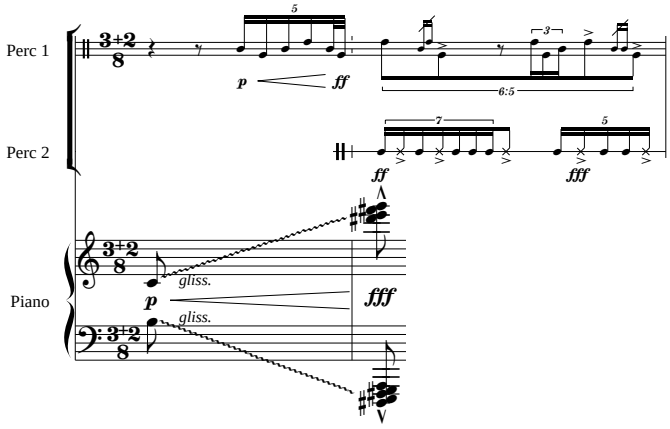
Advanced GUIDO also supports "advanced" notational features, such as glissandos, arpeggios, clusters, or figured bass indications, as well as less commonly used barline types, clefs, rehearsal marks, etc.
Not all of these advanced musical concepts have standardized representations in conventional music notation; this leaves notation software supporting GUIDO some freedom in dealing with them.
While, due to the restricted space, we cannot discuss advanced notational elements in detail here, the example in Figure 2.6 illustrates some of them. Note, for instance, the `\staffFormat` tag in the first voice, which not only defines the staff to have three lines, but also reduces its size (compare to corresponding size-parameter in third voice). Note also the use of a percussion clef (`\clef<"perc">`) in staves 1 and 2. As can bee seen in the first voice, nested tuplets can be described in a straightforward way using `\tuplet` tags with nested ranges; nested beams are represented analogously. Voice 3 shows the usage of the `\glissando` tag, which allows not only the style but also the exact offsets from the reference positions to be specified. Furthermore, the second voice demonstrates the use of the `\staffOff` and `\staffOn` tag, to prevent sections of the staff from being shown.

2.3 Extended GUIDO

Extended GUIDO includes various features, which enhance *Basic* and *Advanced* GUIDO beyond conventional music notation. The work on *Extended* GUIDO is not yet finished: as different user groups currently experiment with using GUIDO in different application contexts, the final version of *Extended* GUIDO will reflect the needs and experiences of these user groups. Some issues have been discussed by the "core" GUIDO developing group; these will be presented shortly in the following.

2.3.1 Microtonality and Tuning

While certain aspects of microtonality can already be realized in *Basic* GUIDO, such as the differences between `d##`, `e`, or `f&`, *Extended* GUIDO also introduces representations for other types of microtonal information, such that different notions of microtonality, such as just tuning, microtonal alterations, arbitrary scales, or non-standard tuning systems can be adequately represented.[ARW98] For example, us-




```
{[ (* voice 1*) ... \systemFormat<staves="1-4",dx=0>
  \accol<id=0,range="1-2",style="straightBrace"> ...
  \staff<id=1,dy=2.5cm> \barFormat<style="staff">
  \staffFormat<style="3-line",size=0.875mm> ...
  \clef<"perc"> ... \meter<sig="3+2/8"> ... \dottedBar<2>
  ... \tuplet<format="-6:5-",dy1=-14.3hs,dy2=-10.3hs>(
  \beam<dy1=-10hs,dy2=-6hs>(g*5/48 \grace<16,"/">(e g) ...
   \tuplet<format="-3-",dy1=4hs,dy2=6hs>(\beam(g*5/144 c e))
   ... ) ) \barFormat<"accolade"> \bar ... ],
 [ (* voice 2, system and staff-layout-tags removed *) ...
  \staffOff \empty*5/8 \staffOn \clef<"perc"> \space<2.5mm>
  \bar ... ], [ (* voice 3, some layout-tags removed *)
  \accol<id=2,range="3-4",style="curlyBrace">
  \barFormat<style="accolade">
  \staffFormat<style="standard",size=1.125mm> ...
  \glissando<style="wavy",dx1=2mm,dy1=1.5hs,
      dx2=-0.7mm,dy2=0hs>(  c1*1/8 \space<7mm>
  \text<"gliss."> empty*4/8 \bar<2>  \space<6mm>
  \stemsDown \marcato( { \headsRight(g3*1/8),
     f#, \headsRight(e), d# } ) )
  \staffOff ], [ (* voice 4 *) ... ] }
```

Figure 2.6: Another *Advanced* GUIDO example

ing the `\alter`-tag, microtonal alteration of standard pitch classes, such as quarter-tone accidentals can be realized; `\alter<-0.5>(a1/8)` represents an eighth note `a1` altered downwards by a quarter tone. Likewise, different tuning systems can be represented by two other new tags, `\tuning` and `\tuningMap`, which are used to select predefined tuning systems, such as just tuning, or to specify a tuning scheme based on systematic alterations of given pitch classes. Finally, concepts such as quarter-tone scales or absolute pitches can be represented by parameterized events, which generalize the concept of an event (such as a note or rest) in *Basic* or *Advanced* GUIDO.

2.3.2 Exact Timing

In *Basic* and *Advanced* GUIDO, note and rest durations are always specified as relative durations. *Extended* GUIDO also allows *exact* timing by specifying absolute durations: `c#2*2500ms` represents the note `c#2` with a duration of exactly 2500 milliseconds. Absolute and relative durations can be freely combined in a given fragment or piece. This can be extremely challenging for a notation system, because the traditional spacing[11] and horizontal synchronization of multiple voices may get very complicated.

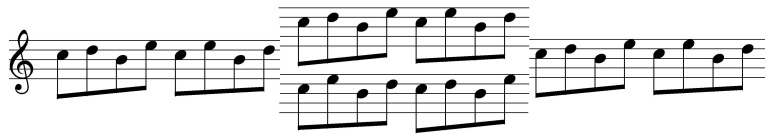
2.3.3 Hierarchical and Abstract Scores

Extended GUIDO facilitates the direct representation of *hierarchical score structures* by allowing *arbitrary* nesting of the sequence and segment constructs used in *Basic* and *Advanced* GUIDO (see section 2.1.2). A simple example for this is shown in Figure 2.7; note that while there is no standard graphical representation of such hierarchical scores, they can be extremely useful in analytical applications. Similarly, *Extended* GUIDO supports the representation of *abstract scores*, which are basically scores containing variable or "holes". These can be useful for representing incomplete structural information such as musical schemata.

2.4 GUIDO as a Music Representation Language

Experience has shown that GUIDO is a powerful and versatile music representation language. In most cases, a given graphical score can be converted fairly easily into a suitable GUIDO description. The opposite – generating a conventional score from an arbitrary GUIDO description – is sometimes more complicated (especially, if the GUIDO description was algorithmically "composed") but it will be shown in the following chapters, how acceptable scores can be generated in most cases.

[11] Spacing is the process of deciding how much space is put after a note or rest. Section 4.5.1 thoroughly covers this issue.



```
[ c2/8 d b1 e2 c e b1 d2 { [ c2/8 d b1 e2 c e b1 d2 ],
                           c2/8 e b1 d2 c d b1 e2 ] } c2/8 d b1 e2 c e b1 d2 ]
```

Figure 2.7: Hierarchical scores in *Extended* GUIDO

Using GUIDO as a music representation language offers some advantages over other representation languages, which will be described in the following sections. Nevertheless, one issue concerning the grouping structure of GUIDO has to be mentioned: When using several sequences to construct a piece (the piece is encoded in poly-normalform; see section 2.1.2), synchronous events appear at different locations in the description. This is a common problem for many music representation languages, especially if they are human readable. The issue is problematic, because these GUIDO descriptions can not be used for streaming purposes, where a client receives a stream of musical information from a server. In a streaming-scenario, the receiving party must (in the worst case) already have received the complete GUIDO description before it can (dis-)play the score or the music. To overcome this disadvantage, a mechanism is needed, which converts poly-normalform descriptions into chord-normalform GUIDO descriptions. Currently, one design issue in the *Basic* and *Advanced* GUIDO specification prohibits this: as was already mentioned in section 2.1.2, a chord is a collection of single events, which all have the *same duration*. There are many instances, where this restriction is too tight. One such instance can be seen in Figure 2.2 on page 11: the tenor voice contains two eighth notes in the second bar; these are omitted in the chord-normalform. To create a streamable GUIDO description, an extended chord-normalform must be defined, which allows chords to contain simple sequences instead of only notes. The details of this still need to be specified exactly.

Another issue, which is closely related to the streaming of GUIDO descriptions, is the question of *editing* complex GUIDO descriptions: in a complex piece containing multiple voices, a single musical time position is present up to n times (where n is the number of sequences). Therefore, changing the musical content at one time position may require editing the GUIDO description at n different positions in the file. This is a common problem that can be found with other representation formats aswell; the only solution for a text based format would be the usage of a two-dimensional spreadsheet which would basically look like a two-dimensional, textual representation of the graphical score. While this approach may solve the problems concerned with editing, it requires somewhat complex formatting even for simple pieces. To

overcome the issue in the case of GUIDO, a specialized GUIDO editor needs to be created, which is aware of different voices and keeps track of mapping time positions to individual voice positions.

2.5 Other Music Representation Languages

Quite a number of different music representation formats have been developed and used ever since computers have been used for music information processing. Some of these representations were built with graphical scores in mind, others are rather performance oriented while still others target the needs of musical analysis. When comparing GUIDO to other music representation formalisms, several similarities and differences can be found: Unlike the MIDI File Format [HwDCF+97] or NIFF [Gra97], GUIDO is a plain-text, human-readable format. This facilitates the realization of GUIDO support for music software and has considerable advantages for using GUIDO to interchange musical data between applications or across platforms and over the Internet. Compared to other plain-text approaches like DARMS [SF97b], SMDL [SN97], or music representations based on XML [CGR01], GUIDO shows improved representational adequacy (in the sense defined on page 7) which again facilitates its usage and implementation. Finally, in contrast to representations which are specifically designed for graphic music notation, like Common Music Notation (cmn) [Sch97], the GUIDO design is focused on musical information, while also supporting exact score formatting features.
The rest of this section compares GUIDO with several other widely used music representation formalisms. For an even more detailed description of musical codes the reader might consult "Beyond MIDI – The Handbook of Musical Codes" [SF97a], which offers a thorough and detailed overview of the most important music representation formalisms in use today.

2.5.1 MIDI

MIDI (Musical Instrument Digital Interface) was originally developed as a protocol for the communication of digital musical instruments (primarily synthesizers). Therefore, MIDI has a strong bias towards piano keyboards and mainly records keyboard events (keys being pressed and released) together with other control parameters. The MIDI file format [HwDCF+97] offers the means to store this protocol information in a file. In order to be an efficient streamable format, MIDI data is transmitted and stored as a stream of bytes which is not directly human readable.[12] MIDI is strongly performance oriented and was never intended to be used as a general music representation language. The usage of MIDI for the interchange

[12]Different MIDI formats exists; when using format 1, different tracks are placed sequentially in a file, therefore format 1 MIDI files are not suitable for streaming.

of musical data is therefore somewhat limited: "Standard MIDI Files have the disadvantage that they represent relatively few attributes of music" [SF97a], page 24. Standard MIDI files cannot, for example, distinguish between $D^{\#}$ and $E^{\flat}$, because these notes are realized by pressing the same key on a piano keyboard. There is also no mechanism for explicitly representing rests within MIDI files. Several extensions to MIDI have been suggested to overcome the various limitations of MIDI for a general musical and notational information interchange.[13] So far, none of the extensions have been widely accepted, therefore, the basic problems remain.

Regardless of the shortcomings of MIDI as a music interchange language, millions of MIDI files exists today. Almost every computer music application supports MIDI either for importing or exporting musical data in a commonly understood format. MIDI is one of the most used music file formats in the realms of score level representation. Everything that can be represented in a MIDI file can also be represented within a GUIDO description. Because GUIDO allows arbitrary event durations, there is no loss of information when directly converting MIDI into GUIDO.[14] This does not hold for the inverse: in most cases, information is lost when converting from GUIDO to MIDI. The reason for this lies in the fact that much of the musical information contained in a GUIDO description simply cannot be explicitly represented using MIDI (e.g. slurs, articulation, etc.).

The complexity of converting MIDI files into graphical scores very much depends on the recorded MIDI data: if the notes are quantized (which means, that all note onsets and durations are put into a predefined set of *allowed* timepositions), and the separation of simultaneous voices into several tracks has been carried out, then a conventional score can be created quite easily. Of course many issues and tasks remain, for example deciding which voice to print in which staff for a piano score. Another dissertation dealing with many of the complex problems of this domain is currently being written by Jürgen Kilian and will be published within the next year [Kil02].

The inverse process (producing MIDI files from GUIDO descriptions) is a straightforward process that has been implemented in [Fri00]. As stated above, most of the explicit musical markup is lost, although it is used for creating a musical "sensible" performance (for example, a `\slur`-Tag in the GUIDO description changes the played duration of the notes in the MIDI file, resulting in a legato playback).

Conclusion The MIDI file format is widely used for representing score level music but has significant drawbacks when being used as a notation interchange format. Because of the different underlying ideas, a direct comparison of MIDI and GUIDO does not yield interesting results. All information present in a MIDI file can be represented in GUIDO, which does not hold for the inverse.

[13] See pages 73–108 in [SF97a].

[14] A direct conversion from (unquantized) MIDI into GUIDO without any "intelligent" processing does generally not produce usable information (at least for music notation purposes).

2.5.2 DARMS

DARMS stands for **D***igital* **A***lternate* **R***epresentation of* **M***usical* **S***cores* [SF97b]. Its development started as early as 1963 and it is still used as a music (and mainly score) representation language today, although currently no "modern" application for reading or writing DARMS exists. Because DARMS was developed to allow complete scores to be coded with a generic computer keyboard, the resulting code is human readable.[15] A multitude of DARMS dialects exist, all specified for special extensions (like for example lute tabulature). All dialects are closely related to the so called *Canonical* DARMS, which was first described in 1976. The term "canonical" was thoughtfully chosen, meaning that the encoding of single elements of a score is unambiguous: there is only one way of encoding notes, rests etc.
One major difference between DARMS and GUIDO lies in the fact that in DARMS, pitch is represented by a number which encodes a position on a staff (which depends on the current clef) rather then being represented explictly like in GUIDO. The note "e" in the first octave would be represented by "1" if a treble clef was used, by "7" for an alto clef, and by "13" for a bass clef, because the position on the staff-line changes. Figure 2.8 shows the GUIDO and DARMS coding of a small musical fragment together with the matching graphical score.

```
DARMS (1): !G 1Q !C 7Q !F 13Q
DARMS (2): !G 1Q !C 7 !F 13

GUIDO (1): [ \clef<"treble"> e1/4 \clef<"alto"> e \clef<"bass"> e ]
GUIDO (2): [ \clef<"g"> e1/4 \clef<"c"> e \clef<"f"> e ]
```

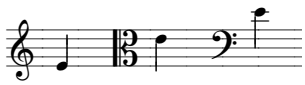


Figure 2.8: DARMS and GUIDO

One important similarity between DARMS and GUIDO is the *carry feature*: In DARMS, the duration or the pitch of a note is carried to the following note. This feature is used in the second DARMS encoding (denoted DARMS (2)) of the previous example (Figure 2.8): the duration (here a quarter note) is kept for all following notes. If the pitch stays the same and the duration is changing, it suffices to encode the new duration without repeating the pitch (which is actually just the line number for the current staff). If neither the duration nor the pitch changes, either pitch *or* duration can be used for encoding the repeating note.
In GUIDO, the carry feature does not work for pitch information, as pitch information is explicit for every note. The carry feature works the for duration and for the register (the octave); this makes hand coding of musical examples fairly fast.
In DARMS, so called push codes indicate non-printing rests: a similar feature is available in GUIDO. Here, the non-printing rests are called "empty" and behave

[15] In order to save memory and to optimize performance, DARMS uses single characters to encode musical information; K stands for key signature, G and F are the treble and bass clef. Even though the resulting code is human readable, the reader must know the meaning of the single characters in order to understand the encoded music.

just like normal events without being visible in the final score.
The concept of grouping several elements horizontally by using a comma (“,”) and whitespace (“ ”) for vertical placement is similar in DARMS and GUIDO.[16]
DARMS and GUIDO both encode music by storing several voices (called “parts” in DARMS and “sequences” in GUIDO) sequentially, which are then stacked horizontally.
One rather advanced feature of DARMS called “linear decomposition” allows to encode complex parts of a score by offering multiple parses, which is equivalent to “going backwards” in time. This feature is not directly available in GUIDO, but as GUIDO descriptions have no limitation on the contained number of voices, “linear decomposition” can be simulated easily by adding voices containing “empty” events together with notes and rests, which encode the relevant area of the score. Figure 2.9 shows a short fragment where linear decomposition (which is encoded in DARMS using “!& ... & ... $&”) is used for the last three notes. The equivalent GUIDO description is also shown: here, two voices are required.

```
DARMS:
!I1 !G -1QU 2QU !& 3EU(  2EU) /  2HU //
                & 0ED( -1ED) / -1HD // $&

GUIDO:
{ [ \staff<1> \clef<"g2"> \stemsUp c1/4 e  \beam( f/8 e ) | e/2 ] ,
  [ \staff<1> \stemsDown           empty/2 \beam( d/8 c ) | c/2 ] }
```

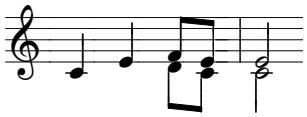


Figure 2.9: Example for linear decomposition in DARMS

Conclusion DARMS is a very well thought of encoding scheme for music notation – especially considering that it was created in the 1960’s. Because of the compressed nature of the encoding format, DARMS looks rather cryptic and can not be understood intuitively. Also, encoding pitch information by positions on staff-lines does not offer an advantage over explicitly encoding pitch information for each note. All information from a DARMS description can be encoded in GUIDO.

2.5.3 MuseData

The MuseData format has been developed by Walter B. Hewlett since 1982 at the Center for the Computer Assisted Research in the Humanities (CCARH) [Hew97]. It is strongly focused on encoding *logical* aspects of musical information, meaning that MuseData represents what could be thought of essential musical information that in terms can be (re-)used for analytical and notational purposes. Up to now, many classical works (by J. S. Bach, Beethoven and many others) have been encoded as MuseData files. Some of the encoded works have also been published as scores

[16] In order to allow an arbitrary amount of whitespace in the case of horizontal groupings (e.g. chords), GUIDO additionally uses curly and straight braces (“{,},[,]”).

which are sold commercially. One interesting aspect of MuseData is, that – similar to GUIDO – the encoded information does not need to be *complete* in the sense, that exact formatting information for notational elements needs not to be encoded; so called *print suggestions* can be used to guide notation programs in producing exactly formatted scores from a MuseData file.
MuseData files are text-based, human-readable files. Different from many other text based representation languages, the *formatting* within the file is an integral part of the represented music: each line is treated as a *record*, containing either header information (like the title and composer of a piece) or musical information. The musical attributes record contain attributes which usually pertain throughout the file (key signatures, time signatures, clef signs, etc.). So called "Data Records" encode musical information (such as notes, rests, etc.) by using different predefined letters and numbers placed at *specific column-positions*. This encoding makes it easy to find a specific location within a MuseData file, but additional overhead is required to ensure the correct encoding of a piece.
Figure 2.10 shows an example MuseData encoding together with the matching GUIDO description and the respective score. The header information is omitted in the MuseData encoding in order to concentrate on the representation of musical parameters. The first line encodes the key signature (2 sharps), the division per quarter note (4), the time signature (0/0), and the clef signs for the first and the second staff (4 is a treble clef, 22 is the bass clef). In the following lines, the notes and rests are encoded. A very powerful feature within MuseData is the "back" command, which is used to "rewind" the time. This is used in the example to print two simultaneous voices within the same part. This feature is similar to the "linear decomposition" feature of DARMS. The note names in the MuseData example can be easily identified ("D4", "G3", etc.). In MuseData the (musical) duration of a note or rest does not necessarily have to match the notated duration; this is an important feature for a complete music representation language, because it allows the representation of uncommon durations without producing an unreadable score. A detailed description of the column-oriented formatting features of MuseData can be found in [Hew97]. For a comparison of MuseData and GUIDO it suffices to know that MuseData offers a large variety of formatting instructions commonly used in conventional scores.
While there are quite some similarities between GUIDO and MuseData, the most important difference is the restrictive formatting of MuseData files. While this may have an advantage when navigating within a MuseData file, it results in quite an amount of overhead when producing the files. Because MuseData and GUIDO were both created as complete music representation languages, they share the notion of encoding musical content rather than music notation. Both formats offer exact formatting features, which may be ignored by applications. The "back" feature of MuseData has no equivalent in GUIDO. This feature can be simulated quite easily in GUIDO, as was shown in Figure 2.9. As the MuseData file format is very well documented and a large number of MuseData files exist, a converting application

(a) MuseData Encoding

```
$  K:2   Q:4   T:0/0   C1:4   C2:22
rest  16      2        1
back  16
rest   2      3 e      2
D4     2      3 e     d2 [
D4     2      3 e     d2 =
D4     2      3 e     d2 ]
G3     4      3 q     d2
B3     4-     3 q     d2       -
measure 2
rest   8      2 h      1
rest   2      2 e      1
A4     2      2 e     u1 [
A4     2      2 e     u1 =
A4     2      2 e     u1 ]
back  16
B3     2      3 e     d2 [
E3     2      3 e     d2 =
A3     2      3 e     d2 =
G3     2      3 e     d2 ]
F#3    4      3 q     d2
D3     4      3 q     d2
/END
```

(b) GUIDO Encoding

```
{
[ % voice 1, measure 1
  \staff<1> \clef<"g2"> \key<2> _*16/16 |
  % voice 1, measure 2
  _*1/2 _*1/8 \beamBegin \stemsUp a1 a a \beamEnd ],
[ % voice 2, measure 1
  \staff<2> \clef<"f4"> \key<2> _*1/8 \beamBegin
  \stemsDown d1 d d \beamEnd g0*1/4 \tieBegin:1 b |
  % voice 2, measure 2
  \beamBegin b*1/8 \tieEnd:1 e a g \beamEnd f#*1/4 d ]
}
```

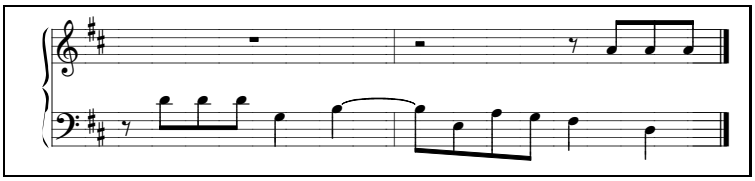


Figure 2.10: Example for MuseData encoding

from MuseData to GUIDO was realized [Fri01].

Conclusion MuseData is a human readable, measure oriented, complete music representation language. MuseData is capable of encoding all aspects of music found in conventional scores. The encoding follows a restrictive line- and column-oriented formalism, which simplifies navigation within the file, but cannot be understood intuitively. A large number of scores encoded as MuseData files exist, which can be automatically converted into GUIDO descriptions. In comparison to MuseData, GUIDO offers a more flexible way to encode musical information.

2.5.4 SCORE

SCORE is a commercial music notation system available for DOS and Windows. It has been developed since the 1980s by Leland Smith [Smi97]. Input to SCORE is encoded as an ASCII file, which contains a description of notes, time position and musical markup (like articulation and slurs).[17]

SCORE has been created with high quality publishing of musical scores in mind; it is therefore very strongly graphically oriented: one input file always describes exactly one page of a score. Many formatting decisions are made automatically by SCORE, but it is also possible to specify the exact location of all graphical elements through the use of parameter files. It is even possible to include arbitrary postscript

[17] In the following, the term SCORE will be used to describe both the input file format and the notation system. It will be clear from the context, what is meant.

symbols and files in a score, making it an ideal tool for contemporary music publishing.
Because SCORE is so much concentrated on the visual aspects of a score, the file format itself is not very useful as a general music representation language, although logical information for a piece can be extracted. Nevertheless, SCORE is a very successful notation program used by many publishers for high quality musical typesetting. The notation algorithms implemented in SCORE are among the finest available and include automatic formatting of various parts of a score. Unfortunately, the developer has not published anything on the implemented algorithms, therefore a direct comparison to other published notation algorithms can not be done.
Even though SCORE input files are ASCII files, they follow a somewhat non-intuitive approach by first specifying the pitch of all notes, followed by the respective durations, followed by articulations and finally ending with the description of beams and slurs.[18]
Conversion from SCORE to GUIDO would probably be possible; at least the basic data (like pitch, octave, most articulations, beaming and slurs) could be easily converted to their respective GUIDO descriptions. From [Smi97] (p. 279): "[SCOREs] strongly graphical nature produces occasional obstacles to analysis. For example, the differentiation of ties and slurs [...] must be inferred from other information."

2.5.5 Common Music Notation (cmn)

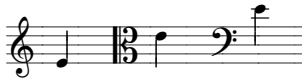
Common Music Notation (cmn) by Bill Schottstaedt [Sch97] is being described as "a simple little hack that can create and display traditional western music scores" on its web page [Sch98]. It actually is more than that: it is a primarily graphically oriented, lisp-based system that creates postscript files from a given textual description, which is actually a LISP expression (resulting in quite a large number of brackets, although in many cases the brackets may be omitted). It can be read and understood intuitively due to its use of common musical names like e.g. "treble" for a treble clef or "double-bar". cmn is very flexible: all output can be exactly formatted where needed (it is, for example, possible to change the beam-width and beam-spacing with simple commands). Because cmn creates postscript output, everything can be easily scaled, rotated, and moved arbitrarily on the page.
A score is encoded in cmn as a series of systems, which contain staves, which contain lists of notes, rests, and other musical objects. It is possible, to attach a list of notes to an already existing staff; this technique is somewhat similar to the linear decomposition mode in DARMS. It is also possible to reset the "musical time", effectively going backwards in time to place more then one voice on the same staff; this is very similar to the "back" command in MuseData.
cmn can be used to encode most conventional and even modern scores. It also allows

[18] Slurs and ties are treated alike in SCORE; this alone disqualifies the usage of SCORE as a general music representation language.

the integration of user defined graphics, although a profound knowledge of LISP is required to actually program new graphical output.
Figure 2.11 shows a short example of a cmn encoding together with the respective GUIDO description and the matching score. Differing from GUIDO, cmn does not offer a carry feature: each note requires an explicit pitch- and duration information. The "duration" is only interpreted as a visual parameter and there is no encoding of "musical" duration, as in MuseData or in GUIDO.

Common Music Notation (cmn):

```
(cmn treble e4 q alto e4 q bass e4 q )
```



GUIDO description:

```
[ \clef<"treble"> e1/4 \clef<"alto"> e \clef<"bass"> e ]
```

Figure 2.11: Example of Common Music Notation (cmn)

Conclusion cmn is a graphical oriented score representation language. Because cmn encodings are LISP expressions, the representation comes close to the "description being the program"-paradigm: the score is the result of evaluating the expression. Because of its graphical nature, cmn is not suited as a general music representation language. A conversion of cmn descriptions into GUIDO descriptions seems possible, although such a conversion should be realized using LISP. The same structural problems as mentioned when discussing the conversion of SCORE into GUIDO will apply.

2.5.6 Plaine and Easie Code

The development of the *Plaine and Easie Code* by Barry S. Brook and Murray Gould was driven by the same idea as the DARMS code: musical notation should be representable by using ordinary typewriter symbols. Plaine and Easie Code was developed for the use in bibliographic applications, for example to classify and retrieve musical themes and fragments in musical databases [How97]. The code was developed to represent melodies in conventional staff notation: therefore it primarily deals with a single voice. Some extensions to the Plaine and Easie Code have been proposed to also represent multiple voices [Gie01]. The Plaine and Easie Code uses ASCII characters to represent pitch class and duration. The code can be best understood by looking at the example in Figure 2.12: first, the f clef on the fourth line is denoted by "F-4". Then follows the key signature ($f^{\#}$ and $c^{\#}$) denoted by "xFC". The time signature is simply encoded as "4/4". Then follows an eighths-note rest "8-" and a beam group of three D's in the first octave (indicated by the high-comma) encoded as "{ 'D D D }". Because the Plaine and Easie Code is oriented solely on the notational elements visible in the score, it is not very useful as a general music representation language: as can be seen in Figure 2.12, all notes are encoded

without accidentals, because no accidentals – besides the key signature – are visible. The encoding is not concerned with the sounding pitch, but stores the visible pitch, ignoring the accidentals. When a melody needs to be written with another key signature, all following notes must be updated to reflect the change. This greatly differs from GUIDO and other formats, where the sounding pitch is encoded.

Plaine and Easie Code:

```
F-4 xFC 4/4 8- {'D D D} 4,G B+ / {8B E A G} 4F D
```

GUIDO description:

```
[ \clef<"f4"> \key<2> \meter<"4/4"> _*1/8
  \beam( d1 d d ) g0*1/4 \tieBegin b |
  \beam( b*1/8 \tieEnd e a g ) f#*1/4 d ]
```

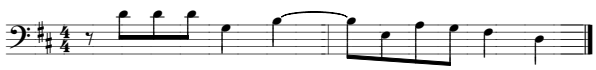


Figure 2.12: Plaine and Easie Code

Conclusion Plaine and Easie Code is a compact notation representation formalism that is not suitable as a general music representation language. The format produces very compact descriptions of melodies, but focuses exclusively on the visible elements of a score. Despite the graphical nature of the format, exact formatting information can not be encoded. A conversion from Plaine and Easie Code to GUIDO could be realized quickly, because the supported notational elements are present in both formats.

2.5.7 MusiXTEX

MusiXTEX [TME99] is part of a whole family of TEX based music typesetting systems. It is a set of TEX macros to typeset orchestral scores and polyphonic music. It does not decide on esthetic problems in music typesetting, therefore the process of "coding MusiXTEX might appear to be (...) awfully complicated (...)".[19] The manual states right at the beginning: "If you are not familiar with TEX at all I would recommend to find another software package to do musical typesetting." It is therefore not intended to be easily used by the average computer user. Nevertheless, being a non-commercial open-source development, MusiXTEX could be used as the underlying framework for a user friendly notation system, although this requires a lot of work.
The basic idea of MusiXTEX is to describe a score as a textual description, which is then converted into a graphical score. As indicated above, MusiXTEX does not make formatting decisions: it is up to the user to provide the system with parameters for every little formatting detail (for example the slope of a beam). Because of its graphical nature, MusiXTEX contains a very large number of musical symbols but it is not very useful as a general music representation format. Because MusiXTEX is based on (plain) TEX, a score description encoded in MusiXTEX can be created and read

[19] Citation taken from the MusiXTEX manual [TME99]

by humans only with intensive training. Therefore, a number of input languages exists that are then converted into MusiXTEX descriptions using converters.
Figure 2.13 shows an example MusiXTEX encoding. For a detailed description of the commands and their parameters, please refer to [TME99].

```
\begin{music}
\instrumentnumber{1}
\setname1{Piano}
\setstaffs1{2}
\gerenalmeter{\meterfrac44}
\startextract
\Notes\ibu0f0\qb0{cge}\tbu0\qb0g|\hl j\en
\Notes\ibu0f0\qb0{cge}\tbu0\qb0g|\ql l\sk\ql n\en
\bar
\Notes\ibu0f0\qb0{dgf}|\qlp i\en
\notes\tbu0\qb0g|\ibbl1j3\qb1j\tbl1\qb1k\en
\Notes\ibu0f0\qb0{cge}\tbu0\qb0g|\hl j\en
\end{extract}
\end{music}
```

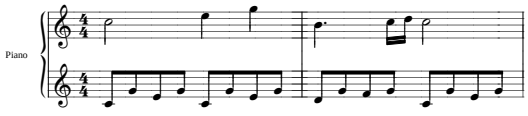


Figure 2.13: Example for MusiXTEX

Conclusion MusiXTEX is not meant to be a simple score description language; it is a complex formatting language for musical scores based on TEX. Because a MusiXTEX description already contains all formatting information, different input languages to MusiXTEX exists. Therefore, it seems possible that complete *Advanced* GUIDO descriptions can be converted into MusiXTEX files, which are then used to render a score.

2.5.8 LilyPond

GNU LilyPond[20] [NN01, MNN99] is another musical typesetting system based on TEX. Differing from MusiXTEX, LilyPond does make formatting decisions. The whole LilyPond system works very similar to the GUIDO Notation Engine:[21] a text file is converted into an internal score representation, which contains the exact location of each notational element that together constitute the graphical score. LilyPond uses TEX for the actual typesetting, whereas the GUIDO Notation Engine uses the so called GUIDO Graphics Stream to describe which elements to put where. Even though the name LilyPond encompasses the whole system for typesetting music, the name is also used to describe the input format.[22] In the following, it should be clear from the context, whether the complete system or just the input file format is meant.

[20] LilyPond is distributed under the GNU Public License [GNU]; in the following, the term LilyPond will be used as a synonym for GNU LilyPond

[21] The GUIDO Notation Engine and the GUIDO Graphics Stream is described in detail in chapter 5.

[22] In earlier versions of the LilyPond documentation, the name Mudela (short for "Music Description Language") was used as the name for the input format. This name seems no longer to be used in current versions.

The encoding of musical data in LilyPond does not only address typesetting issues, but also performance aspects of a musical score.[23] A LilyPond description is human readable plain text, which can be intuitively understood (as long as simple examples are encoded) due to the use of commonly used musical names (for instance note names like "c,d,e" etc., and instructions like "key", "meter", "time" and the like). This feature of LilyPond is very similar to GUIDO. LilyPond differs from GUIDO as it allows the usage of variables and the definition of functions, which can change the default behavior of the notation system. These features make LilyPond more a "programming language" than just a music representation formalism. This greatly enhances the expressive power of the language, but it also makes it more difficult to process or convert LilyPond to other formats.

As LilyPond clearly addresses users, who are comfortable with specifying their musical input as text, great care has been taken to make the encoding of music as comfortable as possible: some nice features include a relative-mode for note-entry, which is used to interpret the octave of a note with respect to the preceeding note by taken the shortest interval – this eliminates the need to explicitly switch octaves too often, a common source of encoding errors when using GUIDO.

The LilyPond system converts the input file into a TEX description, exactly specifying, where each glyph[24] has to be put on a page. It is left to TEX to actually create the graphical output (either a DVI-file or a postscript output). In order to make musical typesetting decisions, LilyPond accesses the font information available (the width and height of individual glyphs). The used musical font – called feta – is distributed freely with the system.

The LilyPond system can be extended by advanced users. Because the input is interpreted and can contain complete functions (using Scheme, a functional programming language similar to Lisp), the LilyPond system is a very versatile and powerful notation system. One problem of LilyPond is its dependency on a variety of other programs and tools: a complete installation of TEX, GUILE, Scheme, Python, etc. is required to run LilyPond. The system setup is rather complicated and can not usually be carried out by people, who have only interest in music. The targeted user of LilyPond is the computer literate, probably already using TEX for typesetting. It is not intended as a replacement of commercially available notation packages and it does not offer any graphical frontend.

Finally, LilyPond is work in progress. It is constantly being enhanced, new features are added. Being an open development, it will probably attract more programmers and users in the near future.

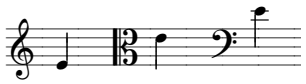
Comparing the LilyPond input-format to GUIDO is not easy, because the intended use is different. LilyPond strongly focuses on notational issues while GUIDO is meant to be a general music representation language. By being extremely flexible,

[23] At the time of this writing, the only performance aspect that can be manipulated is the tempo of playback when converting a LilyPond file into a MIDI file.

[24] A glyph is a single element of a font; most musical symbols are constructed by combining several glyphs.

LilyPond can nevertheless be expanded to include additional musical information that is not used solely for printing. The main drawback of using LilyPond as a general music representation language stems from the fact, that the number of tools needed to run the system is complicated to maintain. An integrated system, that encapsulates the internal mechanism would make LilyPond a very powerful and accessible music notation system.
Figure 2.14 shows an example of a very simple LilyPond entry compared to the same example encoded in GUIDO.

```
LilyPond:
  \score {
     \notes {
       \clef violin
       e4
       \clef alto
       e4
       \clef bass
       e4
     }
   \paper { }
  }
```



```
GUIDO (1): [ \clef<"treble"> e1/4 \clef<"alto"> e \clef<"bass"> e ]
GUIDO (2): [ \clef<"g"> e1/4 \clef<"c"> e \clef<"f"> e ]
```

Figure 2.14: LilyPond and GUIDO

Conclusion LilyPond is a powerful music notation system based on TEX. Input to LilyPond is human readable text, which can be intuitively understood, especially if simple music is encoded. Because the main focus of LilyPond is music notation, its use as a general music representation format is somewhat restricted. Conversion of LilyPond into other formats is aggravated due to the fact that LilyPond encodings may include variables and functions, which are first interpreted internally using the functional programming language Scheme. LilyPond depends on many helper-applications (TEX, Python, Scheme, Perl, etc.), therefore an installation of the notation system is rather complicated. Nevertheless, LilyPond is a complete music notation system, which creates nicely formatted scores.

2.5.9 NIFF

The Notation Interchange File Format (NIFF) [Gra97, Bel01] is a binary format that was meant to be *the* notation interchange format used by the majority of commercial and non-commercial music notation programs as a commonly agreed format to exchange music notation. The specification of the format was completed in 1995.

NIFF is based on the Resource Interchange File Format (RIFF), a data structure defined by Microsoft (and also used in WAV files, for example).

The developers of NIFF, among them the major software companies dealing with music notation, envisioned a complete and universally usable description formalism that would allow the description and interchange of a large variety of scores. Therefore, the actual specification grew rather large, resulting in very few complete implementations of the format. As major sponsors dropped their support, there is currently no further development of the original format (see [Bel01]); the format is nonetheless completely specified and can be used free of charge by anyone interested. A software development kit (SDK) is available for free, which allows reading and writing of NIFF files [Bel01].

One major difference between NIFF and GUIDO lies in the fact that NIFF is a binary format, which requires specialized software for reading and writing a file. Another difference is the lack of adequacy in NIFF – to encode something like a simple scale, the complete layout of the score has to be specified aswell. This makes an intuitive use of NIFF almost impossible, which is not necessarily wrong by itself, as NIFF was never intended to be an easy to use representation formalism. Nevertheless, a direct comparison of GUIDO and NIFF is not a fruitful endeavor. However, the NIFF developers have identified and addressed many of the commonly understood problems of representing music notation. Their effort is a valuable resource for others working in this field.

as commonly understood problems of

2.5.10 SGML based Music Representation Formats

Standard Generalized Markup Language (SGML) is an international documentation standard (ISO) that exists since the 1980s [SGM86]. The most known instance of SGML is the Hypertext Markup Language (HTML) [W3C02], which is used for most of the web pages found on the Internet. Because HTML covers almost no document structuring issues, a simplified version of SGML called XML (eXtensible Markup Language) has been developed [Eck00]. XML is currently gaining popularity and it can be assumed that in the near future a huge proportion of the documents available on the Internet will be written in XML.

Because of the rapid growth of the Internet, some effort has been spent to create music description languages that use the underlying markup language. The first development in this area was the Structured Music Description Language (SMDL) [SN97], a not yet published ISO Standard [SMD95]. SMDL is based on SGML and HyTime (another ISO Standard [HYT95]).

Structured Music Description Language (SMDL)

"SMDL can represent sound, notation in any graphical form, and processes of each that can be described by any mathematical formula."[25] The design of SMDL is based upon several domains: the logical domain which is also called "cantus" is used for music representation, while the visual domain is responsible for graphical aspects, like for instance one or several images of a score. Another domain, the "gestural domain", covers performance aspects of a piece. The general idea is to allow one piece of music in the logical domain to have many score representations in the visual domain and also different performances in the gestural domain. The most important musical representation aspects occur in the logical domain. Pitch can be represented either as frequency or by using names, which are associated with frequencies by using a previously defined lookup table. Duration may be specified using either real time or "virtual" time (which just means that the duration of events is defined relative to each other; conventional scores use virtual time). The use of virtual time is supported by HyTime, which is used to map the virtual time of an event to a real duration based upon tempo information. Any articulation and dynamic information can be encoded in SMDL.
The syntax of SMDL follows the well known syntax of SGML, which is similar to the syntax of HTML. An important aspect of SMDL is the fact that the code was not devised to be easily human-readable; it is supposed to be used as an interchange format, which is read and written by computer programs. The overhead required for representing even simple music prohibits to show an example here; a rather large SMDL description can be found in [SN97]. One specific goal of SMDL is the ability to represent any item found in any other music representation code.
SMDL has not been widely accepted by music software developers so far. One of the reasons for that may be the fact that it is difficult for potential users and tool-developers to see how SMDL might apply to their particular application. SMDL's stated goal to cover everything found in any other music representation code makes the format very broad – a rather large overhead is required even when simple music is represented.

XML based Music Representation

Because of its gaining popularity, XML based formats are being created in many domains (such as mathematics, chemistry, and music). Recently, a couple of music representation languages based on XML have been presented [CGR01]. XML by itself is a meta-format, which mainly defines the *syntax* of the document. The semantics of any XML document is defined by so called Document Type Definitions (DTD's). The main work for any developer of a music representation language is the creation of a DTD.

[25] [SN97], page 470

One major advantage of XML over other representation formalisms is the fact that a growing number of software tools exist for validating and creating XML documents. This does not solve any of the problems found when representing logical aspects of music or when converting a (logical) music representation into a graphical score; but it helps in the validating of the documents (the syntax and semantics of the DTD can be checked without knowledge of the document domain).
At the time of this writing, a dozen different XML-based formats for music representation have been proposed, some of which are prototypical, others cover almost all aspects required for general music representation. Because XML is a hierarchical format, an XML developer has to define which elements of music can be described by using hierarchies (for example one piece contains many voices, one voice contains many measures, a measure contains chords, a chord contains notes, and so on). One interesting feature of XML is a grouping construct based on identifiers. An entity can be made part of group using this mechanism. Some XML-development tools are able to validate these grouping mechanisms (for example to ensure that all identifiers are unique). As an example consider the following:

```
<beam id= "beam1" .../>
<note beamRef= "beam1" .../>
```

where the beam has a unique identifier, which can be used by notes belonging to the particular beam group.
As each XML-based music representation language defines a different DTD, there is no general structure common to these encodings. One thing that is common to all formalisms is the descriptive richness of the formats. For example, there is a `<note .../>` construct in almost any XML-based format; this results in rather long encodings even for simple pieces.
Because XML is a *syntactic* rather than a *semantic* formalism, the representation of music has no ideal or natural match within XML. To demonstrate that XML per se (as opposed to one distinct XML-based representation) has *no advantage* over GUIDO, a DTD and converters for an XML-based music representation based on GUIDO, called GUIDOXML, were developed. A GUIDO description can be simply converted into the one-to-one corresponding GUIDOXML description and vice versa. Figure 2.15 shows a GUIDO description and the matching GUIDOXML-code. Note, that both descriptions of Figure 2.15 contain the *same* information. Because the conversion in both directions is almost "trivial", it can be argued that GUIDO itself is a suitable XML-based music representation formalism.
XML-based music representation languages are still quite new; the future will show, whether any one of the proposed formats will be accepted and used widely.

2.5.11 Commercial file formats

Most of the commonly used commercial music notation products (the two major ones being Finale and Sibelius) have internal file formats that are generally not

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE guido SYSTEM "guido.dtd">
<guido>
   <segment>
      <sequence>
         <clef s="&#x22;treble&#x22;"/>
         <key s="&#x22;B&&#x22;"/>
         <meter s="&#x22;3/4&#x22;"/>
         <note name="h" octave="0" duration="1/4" accidentals="b"/>
         <note name="c" octave="1" duration="1/4"/>
         <note name="d" octave="1" duration="1/4"/>
         <note name="e" octave="1" duration="1/4" accidentals="b"/>
         <note name="f" octave="1" duration="1/4"/>
         <note name="g" octave="1" duration="1/4"/>
         <note name="a" octave="1" duration="1/4" accidentals="b"/>
         <note name="h" octave="1" duration="1/2" accidentals="b"/>
      </sequence>
      <sequence>
         <clef s="&#x22;bass&#x22;"/>
         <key s="&#x22;D&#x22;"/>
         <meter s="&#x22;4/4&#x22;"/>
         <note name="d" octave="0" duration="1/8"/>
         <note name="e" octave="0" duration="1/8"/>
         <note name="f" octave="0" duration="1/8" accidentals="#"/>
         <note name="g" octave="0" duration="1/8"/>
         <note name="a" octave="0" duration="1/8"/>
         <note name="h" octave="0" duration="1/8"/>
         <note name="c" octave="1" duration="1/8" accidentals="#"/>
         <note name="a" octave="0" duration="1/8"/>
         <note name="d" octave="1" duration="5/4"/>
      </sequence>
   </segment>
</guido>
```

```
{ [ \clef<"treble">
    \key<"B&">
    \meter<"3/4">
    b&0/4 c1 d
    e& f g
    a& b&/2 ],
  [ \clef<"bass">
    \key<"D">
    \meter<"4/4">
    d0/8 e f# g
    a h c#1 a0
    d1*5/4 ] }
```

Figure 2.15: GUIDOXML and matching GUIDO description

disclosed to the public and which quite frequently change whenever a new version of the program is released.[26] Finale offers a plugin-interface, which can be used to extract musical information from a file, but this technique requires a running version of the program (for which a purchased license is needed). The same holds for Sibelius. Therefore, the process of converting the files *directly* requires some degree of reverse engineering. Most available conversion programs,[27] which either write or read the commercial formats, rely on limited documentation of the file formats and only extract partial information from the files.

Even though a large number of scores exists, which have been encoded in Finale or Sibelius, these file formats can not be regarded as general music representation languages. As long as the underlying formats are not made publicly available, the data contained within must first be converted into some “open” format before it can be truly interchanged with other applications. Currently, the producers of commercial notation software seem to have no interest in an open interchange format.

[26] Finale does offer limited information on its format (called Enigma); the information is not complete and changes with every new release.

[27] For example, there exists a converter from Finale-files to LilyPond.

2.5.12 Other representation formats

There are sill many other music representation formalisms found in [SF97a] and elsewhere, which have not been mentioned in the previous sections. Some of these formats are either binary formats which often can be read or written only using certain applications, others focus more on music analysis (as, for example, Humdrum/Kern [Hur97]). We believe that the relevant formats, especially the ones concerned primarily with music notation and interchange, have been covered.

2.5.13 Comparison chart

Table 2.1: Comparison chart of Music Representation Languages

Format	**P**erformance, **L**ogical and/or **N**otational representation	Supports exact formatting	Used as interchange format	Intuitive to read	Easily extensible	Convertible to GUIDO
MIDI[a]	P	no	yes[b]	no	partially	yes[c]
DARMS	N	no	yes	no[d]	no	yes
MuseData	L & N	yes	yes	no[d]	no	yes
SCORE	N	yes	no	no	no	yes
cmn	N	yes	no[e]	yes	yes	yes[e]
PaE-Code[f]	N	no	yes	no[d]	no	yes
MusiXTEX	N	yes	no	no	no	no[g]
LilyPond	N[h]	yes	no[i]	yes	yes	partially[i]
NIFF[a]	N	yes	yes	no	no	yes
SMDL	L & N	yes	yes	no	yes	partially[k]
XML	L & N[l]	yes[l]	yes	yes[l]	yes	yes[l]
GUIDO	L & N	yes	yes	yes	yes	yes

[a] Binary format
[b] MIDI is the most used interchange format for score level music
[c] Conversion may require additional processing to obtain readable scores
[d] Dependent on training
[e] cmn descriptions are LISP expressions and require a LISP interpreter
[f] Plaine and Easie Code
[g] MusiXTEX files contain little information on musical structure
[h] If needed, the input file can be coded so that logical information is extractable
[i] LilyPond files are first interpreted using a functional programming language (Scheme)
[k] SMDL may transport more information than is commonly stored in a GUIDO description
[l] Depends on the implemented DTD

Table 2.1 summarizes the properties of the different music representation languages that where compared to GUIDO in the previous sections. Obviously the table does not describe any of the properties in detail; it is merely useful as a rough guideline

to sum up the features of the compared music representation languages concisely.

2.6 Conclusion

In this chapter, GUIDO Music Notation was described in detail and features and examples for each of the three layers of GUIDO (*Basic*, *Advanced*, and *Extended* GUIDO) were given. Then, other music representation languages were compared to GUIDO and their respective advantages and disadvantages were elucidated. Each one of the music representation formalisms, which were presented in this chapter, has a distinct area of application – there is no general format that covers all conceivable aspects of music representation and is easy to use at the same time. Because of the different descriptional focus – some formalisms represent musical ideas better than notational ones and vice versa – it is a matter of the task at hand, which music representation language to use. Many of the above described formalisms concentrate strongly on the graphical appearance of a score (e.g. SCORE, MusiXTEX, and others). These representations are definitely suited for a music notation system, but they lack the power to completely represent the musical content of a piece. As SMDL clarified, a score is just *one* possible view of a piece. Just as different editions (or extractions) may exists, other views (like performance oriented views) may exist. Therefore, the focus on notational aspects alone is not sufficient for a general music representation format. MuseData focuses on the logical domain of music representation, while still being able to cover aspects of exact score-formatting. While this approach is somewhat similiar to GUIDO, it has drawbacks because of the very strict formatting rules. MuseData is not easily extensible, and it is also focused on music from the classical and romantic period. Music without a regular measure structure (either lacking a time signature or having different time signatures for different voices) can not be easily encoded in MuseData. Nevertheless, MuseData is a complete music representation formalism.

There are many reasons for choosing GUIDO as the underlying music representation language for a music notation system. First of all, GUIDO descriptions are easy to create and parse. Secondly, the possibility to represent exact score-formatting information within a GUIDO description is of great advantage – all algorithms of the notation system can be described as GUIDO to GUIDO transformations. But, as GUIDO is primarily describing musical content, the formatting aspect within a GUIDO description is just additional information. The fundamental musical structure of a piece is always closely related to the structure of the GUIDO description. Additionally, tools for removing all formatting information from a GUIDO description exist. Therefore, the essential musical information can be easily extracted.

Because GUIDO is such a flexible representation language, the conversion of arbitrary GUIDO descriptions into conventional scores can be complicated: just consider non standard note durations being specified (like `c*11/14`), or a complex piece given without exact formatting instructions. The algorithms required for converting such

GUIDO descriptions into acceptable conventional scores will be presented in the following chapters.

Chapter 3

A Computer Representation of GUIDO Music Notation

Introduction

In chapter 2 it was shown that GUIDO is a powerful general music representation language which is not only suitable for representing musical structure but encodes notational aspects as well. As GUIDO is a text-based format, applications that work on GUIDO descriptions require an internal storage format in order to efficiently access the represented musical and notational data.[1] This storage format generally differs depending on the application – a music notation application has other requirements than a musical analysis tool. In the context of music notation, great care must be taken when defining the internal representation: as will be shown in the following chapters, music notation can be defined as transformations of under-specified GUIDO descriptions into well-defined GUIDO descriptions: The under-specified GUIDO description does not contain complete score formatting information, which is automatically added by the music notation system. These transformations require a powerful internal representation, as many of the notation algorithms will be carried out on this structural level. A well structured inner representation of GUIDO is not only valuable in the context of music notation, but can be used for a broad range of applications. It should be evident that all notationally relevant information present in a GUIDO description is encoded in the implemented inner representation.
The work presented here did not start from scratch. Other applications using GUIDO have been developed before this thesis was written.[2] Nevertheless, most

[1] Storing GUIDO descriptions internally as textual representations requires continuous re-parsing of the strings, which would be inefficient.

[2] One major application certainly was the SALIERI-System [HKRH98]; this comprehensive computer music system incorporates a musical programming language, whose musical data type is closely related to GUIDO. As the definition of GUIDO became more complex, the development of SALIERI stagnated; therefore the inner representation used in the SALIERI system can not repre-

of these applications are using a task-oriented inner representation that does not cover all aspects of GUIDO. One common tool being used by almost every GUIDO compliant application is the GUIDO Parser Kit [Hoo], which includes a complete lex/yacc-generated parser[3] for GUIDO. The GUIDO Parser Kit does not include an inner representation for GUIDO; it merely offers a collection of hook-functions that are called whenever a GUIDO-event or tag is recognized in the parsed string. It is up to the user to build and fill an inner representation from these hooks.
The inner representation for GUIDO descriptions that was developed as part of this thesis is called "Abstract Representation" (AR). This chapter deals with the object-oriented design and the structure of the AR. It will further be shown that any GUIDO description (and its corresponding AR) can be converted into a semantic normal form; this normal form simplifies the structure of the AR and is also useful for clarifying the semantics of a GUIDO description. In the following, the transformation of GUIDO descriptions into the AR will be explained. Finally, the requirements and methods for accessing and manipulating the AR are presented.

3.1 Defining the Abstract Representation (AR)

When an inner representation for GUIDO is defined, certain requirements must be met. The most important aspect is, that for each GUIDO description there exists a one-to-one corresponding instance of the inner representation. In this section, the ideas and requirements that led to the creation of the Abstract Representation (AR) are presented.

3.1.1 Document View Model

The idea of splitting the process of converting GUIDO descriptions into conventional music notation into several layers was inspired by the document view model (dvm), which was first introduced in Smalltalk and made popular in the Microsoft Foundation Class library [msd02]. The dvm is an an abstract model that describes how an application can **encapsulate** its data from the actual display and manipulation of the data. Within the dvm, the *document* contains the data, which is displayed and manipulated by different *views*. Retrieval and change of the data by its views is done by using only strictly defined interface functions; this ensures that an alteration of the data in one view is reflected immediately in all other views of the same data. The separation of *data* and *user-interface* helps in focusing on the different requirements for each part. As different users may have different preferences for the display and manipulation of the data, different views can be implemented.

sent all of GUIDO.

[3] Lex and yacc are tools for automatically generating parsers from grammars; the GUIDO Parser Kit contains a complete grammar for GUIDO.

As an example, consider a word processing application: the *document* is the abstract internal representation of the collection of words, graphics and formatting information that the user manipulates thru different *views*, which could either be a WYSIWYG[4]-view or a non-formatted textual representation of the document (as in the case of a LaTeX-document for example).
In the case of GUIDO Music Notation, the *document* is the Abstract Representation (AR) containing all information present in a GUIDO description; a *view* of this AR can either be the conventional music notation (the creation of which will be explained in chapter 4), or it can be a textual representation (which would be the (textual) GUIDO description itself), or some other (graphical) interpretation (a hierarchical display of a GUIDO score for example). Figure 3.1 shows the AR in the context of the document view model.

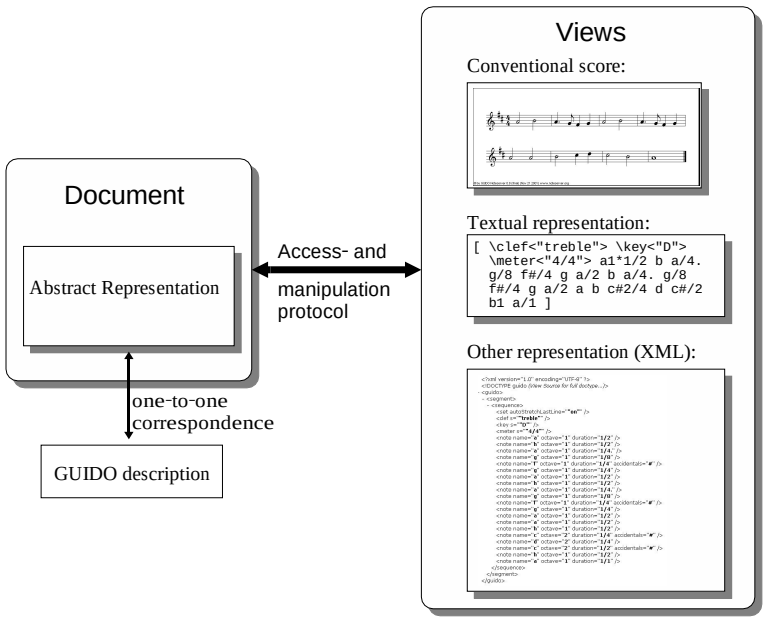


Figure 3.1: The Abstract Representation in the Document View Model

The dvm was chosen as the model for the inner representation of GUIDO, because this design pattern closely matches the structure of GUIDO descriptions with respect to music notation: the musical information contained in a GUIDO description do not necessarily represent a score. As will be shown in chapter 4, the process of converting an arbitrary GUIDO description into a conventional score requires a large number of steps. One graphical score represents only one possible view of the GUIDO description. Other music notation algorithms may result in another score. Therefore, the dvm captures the inherent structural properties of GUIDO.

[4]WYSIWYG: What You See Is What You Get

3.1.2 Definition and Requirements

As suggested by the dvm, the Abstract Representation (AR) is the document containing all information present in a GUIDO description. This leads to the following definition:

Definition (Abstract Representation (AR)):

> The **Abstract Representation (AR)** is a data structure capable of storing all information contained in a GUIDO description. The AR provides efficient access and manipulation functions to the stored (musical) information.

This definition does not solely focus on music notation, but is valid for all kinds of applications. The current implementation of the AR has been developed with conventional music notation in mind, but it can easily be used in other application areas, where GUIDO is used as the underlying data format.
Several requirements must be met when designing and implementing the AR:

- efficient conversion of GUIDO descriptions into the AR
- efficient conversion of the AR into GUIDO descriptions
- the possibility to easily manipulate the AR for musical and/or notational algorithms
- efficient conversion of the AR into conventional music notation
- portability between different computer platforms and different musical applications

These requirements lead to a data structure, which is closely tied to the general design of GUIDO. The data structure and the manipulation functions will be described in other sections below. Figure 3.2 shows the interactions required of the AR when a GUIDO descriptions is converted a conventional score.

3.2 GUIDO Semantic Normal Form

The GUIDO specification [HH96] clearly defines the syntax of GUIDO descriptions. However, there exist numerous ways of expressing a musical and/or notational idea within GUIDO. In order to be able to handle arbitrary GUIDO descriptions in an unambiguous way, the "GUIDO Semantic Normal Norm" (GSNF) has been developed in the course of work on this thesis. The notion of this normal form does not mean that all possible descriptions of a musical and/or notational idea can be mapped onto one single (and equivalent) normal form, but that semantically equivalent descriptions have the same normal form.

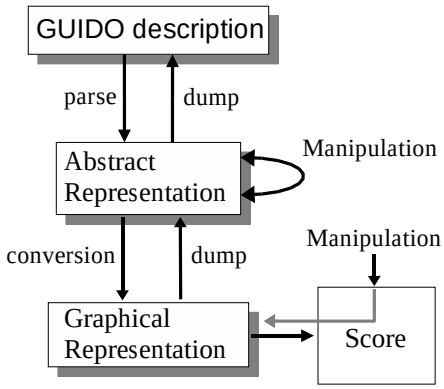


Figure 3.2: Interactions of the Abstract Representation

3.2.1 Semantic equivalence

Syntactically correct GUIDO descriptions contain musical information which is subject to interpretation by different applications. The syntactical rules for GUIDO descriptions however leave room for some ambiguity; consider the following two sequences describing the same musical content (they are musically equivalent):

(1) [\slur(\clef<"treble"> c d e)]

(2) [\clef<"treble"> \slur(c d e)]

In the first sequence, the tagged range of the \slur-tag encompasses the \clef-tag and the following events, whereas the tagged range of the second sequence only contains events. In order to define unambiguous semantics for handling the above examples, a simple rule was introduced in the GUIDO definition:

> *Tagged ranges only affect the contained events.*

This simple rule has far reaching consequences when interpreting GUIDO descriptions:

1. Tagged ranges that do not contain events can be ignored.
2. If tagged ranges begin (or end) with a tag, the beginning (or ending) can be shifted forward (backward) to the next event.
   (In the example above, the first GUIDO description (1) contains the \slur-tag, which covers a range beginning with the \clef-tag. Therefore, the beginning of the range can be shifted; the result is the second GUIDO description (2).)
3. When more than one range tag begins or ends at the same event, the order in which the range tags appear is not significant.
   It is therefore possible to reorder these tags in lexicographical order (examples for this will be shown below).

No	GUIDO description	GSNF
1	`[ c \slur( ) d ]`	`[ c d ]`
2	`[ c \slur( \clef<"treble"> ) d ]`	`[ c \clef<"treble"> d ]`
3	`[ c \slur( \clef<"treble"> d e ) ]`	`[ c \clef<"treble"> \slur( d e ) ]`
4	`[ \slur( \cresc( c d e ) ) ]`	`[ \cresc( \slur( c d e ) ) ]`
5	`[ \cresc( \intens<"p"> c d ) ]`	`[ \intens<"p"> \cresc( c d ) ]`

Table 3.1: Semantically equivalent GUIDO descriptions

Taken these consequences a step further, it is now possible to define the GUIDO Semantic Normal Form:

Definition (GUIDO Semantic Normal Form (GSNF)):

> A GUIDO description G is in GUIDO **Semantic Normal Form (GSNF)**, if and only if the following holds:
>
> 1. G contains no empty tagged ranges
> 2. All tagged ranges in G begin and end at events
> 3. All tagged ranges that begin or end at the same event appear in alphabetical order

The GSNF can now be used, to easily define the notion of *semantical equivalence* as it is needed when interpreting different GUIDO descriptions:

Definition (Semantical Equivalence):

> Two GUIDO descriptions are **semantically equivalent**, if they have the same GSNF.

Table 3.1 shows semantically equivalent GUIDO descriptions based on the above rules; each line in the right column of Table 3.1 is the GSNF of the GUIDO description in the left column. Some of the examples in Table 3.1 give rise to several questions; for example, there seems to be no possibility, to attach a range tag with a simple tag (as in example no. 5, where one might interpret the `\cresc`-tag being attached to an `\intens`-tag. If the musical/notational intent is to attach a range explicitly to a tag (as could be the case in example no. 5), it is however possible, to define `empty` events with no duration; the example no. 5 would thus be written as `[ \cresc( empty*0/1 \intens<"p"> c d ) ]`. Different graphical offsets could then be applied to the `\cresc`-Tag, which would be attached to the empty event, which is not directly shown in the score.

3.2.2 Transforming GUIDO descriptions into GSNF

GUIDO descriptions can be easily transformed into GSNF by shifting the beginning and ending of tag ranges where needed. The desired effect can be obtained by applying a simple algorithm:

1. Find all range tags and apply the following tests:
2. If a range does not begin at an event, shift the beginning of the range to the right until either an event or the range end is reached.[5]
3. If a range does not end at an event, shift the end of the range to the left until either an event or the beginning of the range is reached.[5]
4. Remove empty ranges (and tags belonging to them).
5. If more than one range tag begins or ends at the same event, order them alphabetically by their name.

As an example, consider the third row of Table 3.1: By applying rule 1, the beginning of the slur range is shifted towards the right and the GSNF is obtained.
The GSNF leads towards a data structure for GUIDO sequences that distinguishes between events, non-range tags, and range tags. Events and non-range tags can be stored in an ordered list, where the order of elements is taken directly from the GUIDO description. The time positions of the elements form a monotone rising function.[6] For example, the sequence of elements for the third line of Table 3.1 is [c \clef<"treble"> d e][7] and the respective time positions are $0 \leq \frac{1}{4} \leq \frac{1}{4} \leq \frac{2}{4}$.[8]
The range tags are each split into a begin- and an end-tag, which store a pointer to the location of their respective events within the element list. Figure 3.3 visualizes the data structure for the third row of Table 3.1. Note that the range tags are itself ordered by the position they point to in the element list and also by alphabet, if they begin or end at the same event.
The described data structure will be discussed in more detail when presenting the structure of the AR in section 3.3. The advantage of separating the range tags from non-range tags and events lies in the fact that they can thus be easily added, removed or exchanged. These operations will be needed for the different notation algorithms in chapter 4.

[5] If another range tag is encountered during the search, it can be ignored.

[6] There are some minor exceptions to this rule; consider for example the \cue-tag, which is used to describe cue notes. All events within the range for the cue tag add their duration to the time position, but when the range is closed, the time position is reset to the time position of the starting event in the range. The internal representation deals with this by introducing a completely independent voice and therefore maintaining the monotone rising time function for each individual voice.

[7] Note that the \slur-tag is not present in this list.

[8] In GUIDO, the duration for the first event is $\frac{1}{4}$ if nothing else is specified.

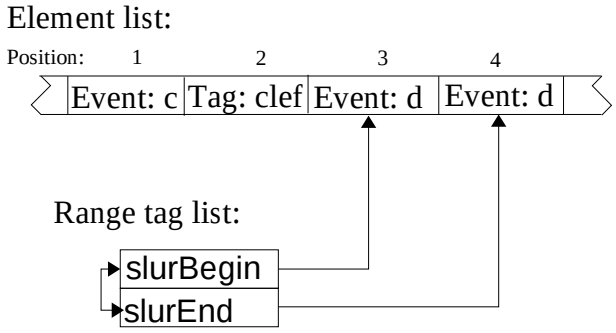


Figure 3.3: Visualization of the GSNF for example 3 of Table 3.1

3.3 Structure of the Abstract Representation

When looking at a GUIDO description, the notion of "objects" comes quite naturally: a GUIDO-event or a GUIDO-tag can be mapped directly onto an event- or tag-object. These objects store all relevant data, for example the time position or the note- or tag-name and tag parameters (if present). Depending on the event type (an event is either a note, a chord, or a rest), a note-, or chord-, or rest-object can be build, which inherits from the generalized event-class. As object-oriented design facilitates software-development, especially when large systems are concerned, this technique was used for the the development of the AR [Boo91]. For reasons of efficiency, C++ [Str91] was used as the object-oriented implementation language. C++ is available for a large number of platforms and offers all features required for modern object-oriented design (like templates, runtime-type-information, multiple inheritance, etc.). Naturally, the design of the AR closely resembles the structure of GUIDO, which greatly simplifies the process of converting GUIDO descriptions into the AR (see section 3.4).

The two main structural concepts of GUIDO, namely *sequences* and *segments* are also represented in the AR: Sequences are represented by a class called `ARMusicalVoice` and segments are represented by a class called `ARMusic`. Class `ARMusic` is basically a collection of voices, combined with methods for parallel accessing the contained voices (see section 3.5.2).

Figure 3.4 shows a class diagram in UML[9]-Notation [FS00] of the main classes of the AR (altogether, there are more than 70 classes for the AR). Every class name within the AR begins with the prefix `AR` and all but some classes inherit directly or indirectly from class `ARMusicalObject`, which contains data and function-definitions required for all musical objects, like for instance the time position, and the duration.

As described in section 3.2, each GUIDO description can be converted into GSNF. This fact has been used in the design of the AR, so that an efficient conversion from

[9]UML: Unified Modeling Language

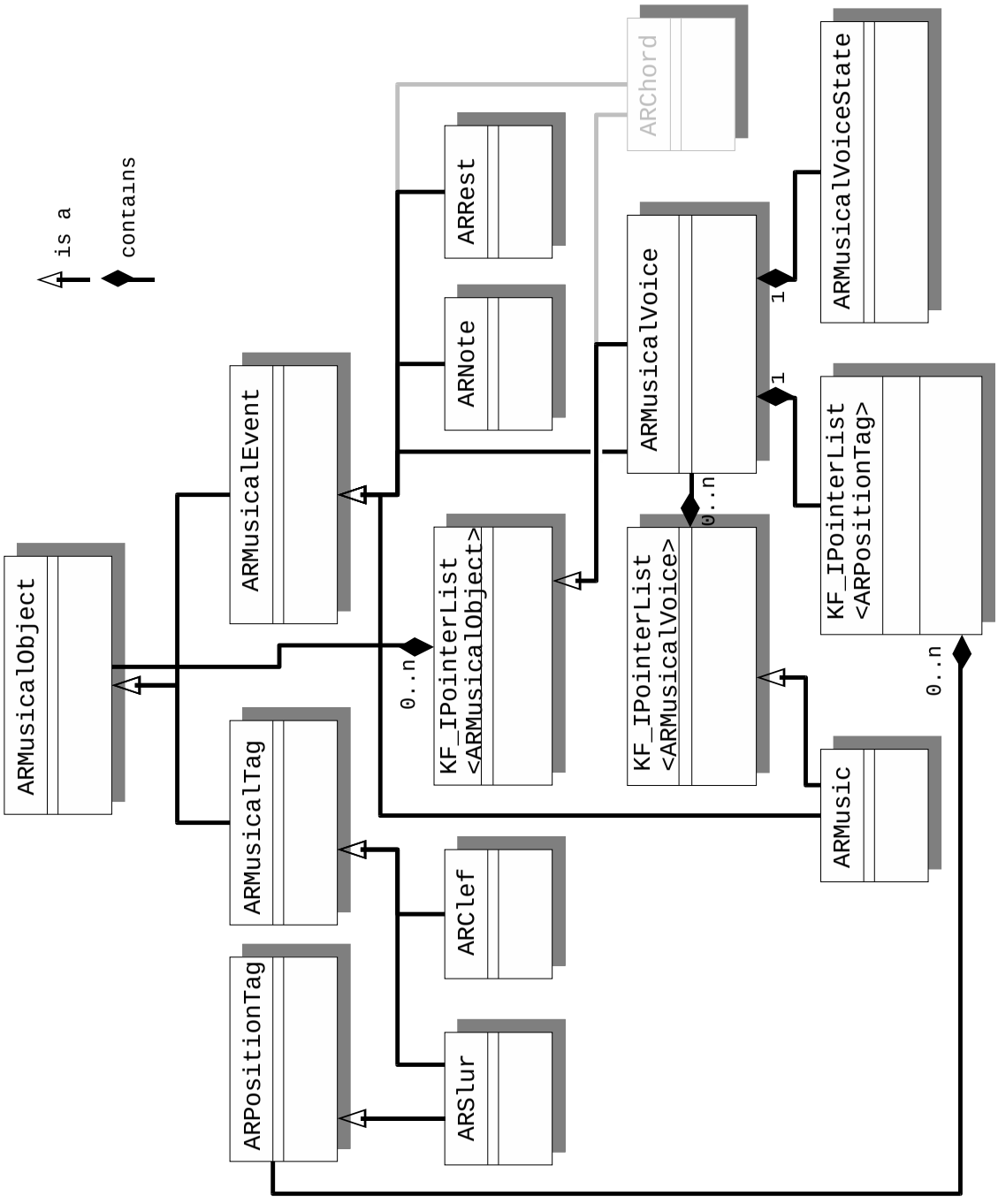


Figure 3.4: Class diagram for the main classes of the AR

and to GUIDO descriptions is possible. When a GUIDO description is converted into the AR, it is first converted into GSNF before any manipulation algorithm is carried out. All manipulation algorithms of the AR require the internal representation to be in GSNF; therefore, when a GUIDO description is extracted from the AR, it will always be in GSNF.
In the following, the main classes of the AR that store musical information will be introduced.[10]

Class `ARMusicalObject`

All classes within the AR inherit directly or indirectly from class `ARMusicalObject`. This class is the base entity for all musical information. Class `ARMusicalObject` stores the duration and the time position of the respective instance, as this information is shared by all musical objects in a GUIDO description. In the case of tags, the duration is always zero.[11] The time position depends on the entities' position within the GUIDO sequence.
Because almost all classes are descendants of class `ARMusicalObject`, collections of events, or tags can be defined as lists containing objects of type `ARMusicalObject`. Using the runtime-type-information feature of C++, it is possible to determine the actual type of an object at runtime.

Class `KF_IPointerList<class T>`

This helper class implements a doubly linked list of pointers to objects of type `T`; it is implemented using the template-feature of C++. All functions for traversing or adding and removing elements are implemented. Class `KF_IPointerList` is intensively used throughout the AR. In the future, class `KF_IPointerList` might be exchanged with the list class from the Standard Template Library (STL) [Rob98], as the STL usually offers very efficient data structures. Because of the object oriented design, such a change would be transparent to the user of the AR.

Class `ARMusic`

A complete GUIDO description can be represented by one instance of class `ARMusic`. As class `ARMusic` directly inherits from class `KF_IPointerList<ARMusicalVoice>`, it can be viewed as a list of voices, just as GUIDO descriptions are collections of sequences. Class `ARMusic` also directly inherits from class `ARMusicalEvent`, because it has all properties of an event. This feature will be useful for

[10] A thorough introduction to all classes of the AR is not possible here because of space restrictions, but the general concept can be understood by studying the main classes.

[11] Events may also have zero duration, but this is mostly needed for sophisticated graphical formatting of a score.

the definition of hierarchical scores (see also section 2.3.3): a complete score can be interpreted as an event contained in another score.[12]
Class `ARMusic` defines some notational transformation functions, which will be described in detail in chapter 4. Most importantly, class `ARMusic` offers parallel access to the contained voices. This feature is important for manipulating the AR when it is converted into a conventional score.

Class `ARMusicalEvent`

Class `ARMusicalEvent` is the base class for all GUIDO events: notes, chords[13] and rests. An event has a duration (which may be zero) and it can be the beginning or end of a range tags. As GUIDO events can specify durations as fractions in combination with dots (as in `[ c1*1/4.. ]`); class `ARMusicalEvent` has a member variable, which holds the number of dots for an event (which would be two in the previous example). The actual duration of the event is stored in a member variable of class `ARMusicalObject`.

Class `ARNote`

Class `ARNote` inherits from class `ARMusicalEvent` and describes GUIDO note events. Class `ARNote` has member variables for the note name, pitch, octave, intensity,[14] and a list of accidentals. When parsing arbitrary GUIDO descriptions, it may very well be, that the duration of an instance of class `ARNote` is not displayable as a single graphical element; these notes will be treated by the music notation algorithms of chapter 4.

Class `ARRest`

Class `ARRest` inherits from class `ARMusicalEvent` and describes GUIDO rests. Rests do not store additional information, because they are completely defined thru their duration, which is already held in class `ARMusicalObject`.

Class `ARMusicalTag`

Class `ARMusicalTag` is the base class for all tag classes. It stores common tag data and offers common tag functions; class `ARMusicalTag` stores the following data:

- `id`: The ID of the tag, if an id was used in the GUIDO description as in `\slurBegin:3`
- `hasRange`: is 1, if the tag has a range (as in `[ \slur( c d e ) ]`)

[12] This concept has not yet been realized in the graphical representation.
[13] Chords are handled different from notes and rests; see section 3.3.1.
[14] The intensity is stored so that the AR can be used for playback-applications.

- `allowRange`: is 1, if the tag is allowed to have a range (as for example the `\fermata`-tag). This parameter is used during the parsing of GUIDO descriptions: if it is zero, and the parsed tag has a range, the `error` flag is set and the tag is ignored.

- `assoc`: the association of the tag. This flag describes, the direction of the association that a tag has with respect to events. The `assoc` flag has one of the following value: Left-Associated (LA), Right-Associated (RA), Don't-Care (DC), Error-Left (EL), Error-Right (ER). If the tag is a begin-tag (like `\slurBegin`), the association is always RA, which means that the tag is associated with the the next event (being on the right hand side in the GUIDO description); if the tag is an end-tag (like `\crescEnd`), the association is always LA, which means that the tag is associated with the previous event (which is on the left hand side in the GUIDO description). If the tag is not a range tag, the association is DC. In the case of an error (`error`-variable is set), the association is one of ER or EL.
  These parameters will be explained further in section 3.4, when the conversion of GUIDO descriptions into the AR is presented.

- `error`: is 1, if an error occurred while handling tag-parameters or tag ranges.

Class `ARPositionTag`

Class `ARPositionTag` is one base class for all range tags.[15] An instance of class `ARPositionTag` stores a pointer to a position in the element list of class `ARMusicalVoice` and also a pointer to the matching ending position tag (which is again an instance of class `ARPositionTag`): the begin-tag stores a pointer to its matching end-tag and vice-versa. One range tag from a GUIDO description is internally broken into two instances of class `ARPositionTag`: one begin- and one end-tag. Therefore, the AR of the two GUIDO sequences [`\slur(c d e)`] and [`\slurBegin c d e \slurEnd`] is identical.[16]
A tag, which can occur either as a range tag or as a non range tag (the `\text`-tag, for example) inherits from both class `ARMusicalTag` and class `ARPositionTag` so it can be handled as either one.

Class `ARMusicalVoice`

Class `ARMusicalVoice` completely represents a GUIDO sequence. It is used by class `ARMusic` to represent a complete voice within a piece. Class `ARMusicalVoice` directly inherits from class `KF_IPointerList<ARMusicalObject>`, so it

[15] As the AR uses multiple inheritance, a range tag usually has at least two base classes.

[16] Internally, there is a distinction between the `\slur`-tag with a range and the `\slurBegin`- and `\slurEnd`-tag, so that the original structure of the input is not destroyed. Nevertheless, the two sequences are treated completely alike.

can be interpreted as a **container** for musical objects. The inherited list stores all events and non-range tags in the order in which they appear in the GUIDO description. Additionally, class `ARMusicalVoice` contains a list (class `ptaglist`), which holds instances of class `ARPositionTag`. This list stores all range tags (begin- and end-tags) ordered by their respective start- end end-positions in the voices element list. The list-order follows directly out of the definition of the GSNF in section 3.2. Figure 3.5 visualizes the structure of class `ARMusicalVoice` for the given GUIDO sequence.

```
[ \slurBegin c \crescBegin \tie( d \slurEnd \crescEnd d ) ]
```

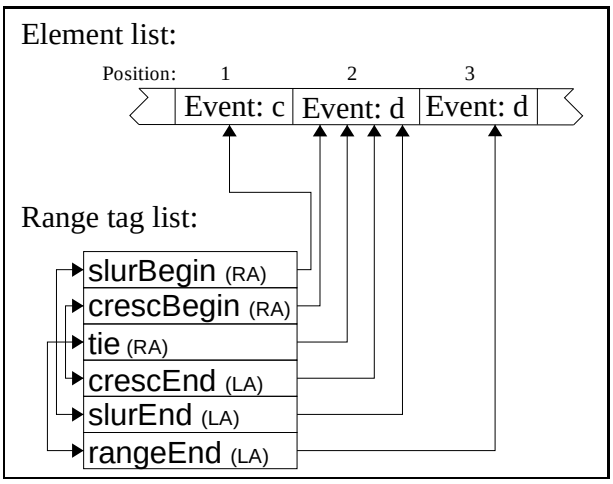


Figure 3.5: Structure of `ARMusicalVoice` for a GUIDO sequence

Class `ARMusicalVoice` offers all operations required for traversing the voice (see section 3.5 for details on accessing the AR). The required state information is stored in the helper class `ARMusicalVoiceState`.

Class `ARMusicalVoiceState`

Class `ARMusicalVoiceState` is needed when a voice is traversed sequentially (see section 3.5). Class `ARMusicalVoiceState` stores all state information for an instance of `ARMusicalVoice`, so that it is possible to resume a position within a voice exactly. The information stored includes the current time position, the current position-tags, and the current location within the element- and the position-tag-list.

Class `ARSlur`

Class `ARSlur` is included in this list of classes as an example for a range-tag. Class `ARSlur` represents the `\slur`-tag; being a tag, it inherits from class `ARMusicalTag`.

Because the \slur-tag is a range-rag, class ARMusicalTag also inherits from class ARPositionTag.

Class ARClef

Class ARClef is an example of a non-range tag; it represents the \clef-tag in a GUIDO description. Class ARClef inherits directly from ARMusicalTag and is usually included in the element list of class ARMusicalVoice. The class stores all information required for representing a clef: a name ("treble", "bass", etc.), the octave, the base-line-number, etc.

3.3.1 Representing chords in the AR

The class ARChord in Figure 3.4 is shaded in gray, because it is not present in the current implementation of the AR. The representation of chords within the AR is rather complex issue, which will be elucidated in this section.

Chords as single events

The first approach for handling chords in the AR treated chords like other events (notes and rests). This approach worked quite well at first, because chords were treated as single entities that had no additional markup. This changed when it became clear, that tags within individual chord notes are required for musical and notational markup.[17] Figure 3.6 shows a GUIDO description together with an inner representation, in which chords are stored as single musical events (containing other events). Because the chord is stored at one single position in the event list, range tags can only point to this one single location. In the example of Figure 3.6 the \tieBegin-tag points to the second position of the event list. As can be seen from this example, the chord must "redirect" the tag to its correct location within the chord. This redirection must be handled by some mechanism within the data structure – this shows that a chord cannot be regarded as a simple event. Representing chords as single events is also difficult when considering GUIDO Semantic Normal Form: if a chord is a single event, a tag can only affect the whole event. There is no mechanism for "partial" tags in the GSNF. Additionally, the conversion of this inner representation into a conventional score has to deal with chords in a specialized way, introducing an additional layer of complexity for dealing with tags within chords. Therefore, a new solution was developed:

[17] Think of partial ties, for example, that begin at a single note within a chord and end somewhere later.

```
[ c { c, \tieBegin e, g } e \tieEnd ]
```

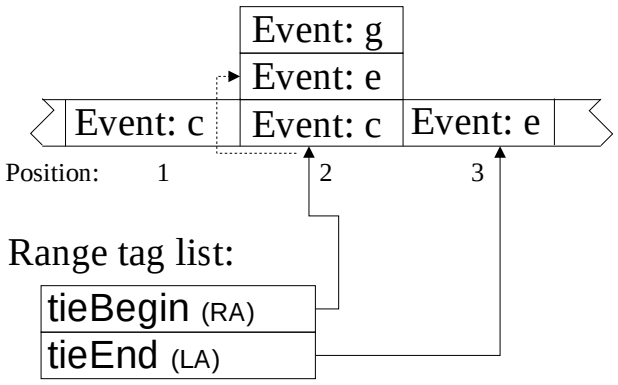


Figure 3.6: Chords as single events in the Abstract Representation

Splitting chords into a series of events

The idea for a suitable inner representation of chords came while looking at the graphical representation of a chord in a score: A chord is an entity, where several noteheads share a common stem, or at least a common horizontal position. Using this observation, the idea of "graphical equivalence" was created: instead of representing a chord as a special form of an event, it can be represented by a series of events combined with some formatting information. The implemented inner representation of chords handles chords as graphical entities, which share a common stem. Using this representation, all aspects of the GSNF (see section 3.2) can be maintained. Figure 3.7 shows how the same example as presented in Figure 3.6 is currently represented in the AR. The GUIDO description

```
[ c { c, \tieBegin e, g } e \tieEnd ]
```

is internally converted into the GUIDO description

```
[ c \chordBegin \displayDurationBegin<1,4,0>
  \shareStemBegin empty*1/4 \chordComma \tieBegin e*0
  \chordComma g*0 \displayDurationEnd \shareStemEnd
  \chordComma empty*0 \chordEnd e*1/4 \tieEnd ]
```

At first, this transformation might seem exaggerated, but using this formalism, all issues concerning tags within chords can be resolved. Four new internal[18] tags are used:

[18] Internal tags are non-standard GUIDO tags that are used within the developed notation system. They are not part of the GUIDO specification.

- `\chord`: This tag groups all events and tags of a chord together. The *musical duration* of the chord is encoded in the first (empty) event of the `\chord`-tag range, which is added automatically during the conversion. All the other events in the `\chord`-tag range get a *musical duration* of zero. This is important, because otherwise, the musical time of the voice would not match the musical time of the original GUIDO description. In the example above, the musical duration of the chord (and therefore of the first empty event within the `\chord`-tag range) is a quarter note.

- `\displayDuration`: This tag enforces a specific *visible duration* to be used when creating the notes and rests within its range. Instead of using the musical duration (as given in the GUIDO description) for creating the notation symbols, the tag parameters are used to create note heads and rests. In the example above, the displayed duration is $\frac{1}{4}$ without dots (the last parameter is zero). Therefore, all events within the tag-range are displayed as quarter notes, even though their musical duration is zero.

- `\shareStem`: All notes within the tag range share a common stem.

- `\chordComma`: This tag groups the individual voices of the chord. Using the `\chordComma`-tag in the transformed chord representation is mandatory, because otherwise the inner chord representation could not be re-converted into the equivalent GUIDO description.[19]

The current chord representation has more consequences than might be first expected: when traversing a voice, an *event-mode* and a *chord-mode* must be distinguished. In event-mode, everything within the voice is iterated sequentially. This mode is used when converting the abstract representation of a voice into a conventional score. In chord-mode, all events that are part of a chord are read in one step. This mode is used for many manipulating routines, which will be described later in this chapter.

3.4 Transforming GUIDO descriptions into the AR

Because the design of the Abstract Representation closely follows the GUIDO design, the creation of one specific AR for a given GUIDO description is a straightforward process. Following the notion of the "factory design pattern" [GHJV94], the class `ARFactory` was implemented. A "factory" is an object that "builds" other objects. In our case, class `ARFactory` constructs all objects of the AR depending on the given GUIDO description. Class `ARFactory` serves as the link between the GUIDO parser

[19] Without the `\chordComma`-tag one could not distinguish between `[ { c \intens<"p"> , e } ]` and `[ { c, \intens<"p"> e } ]`.

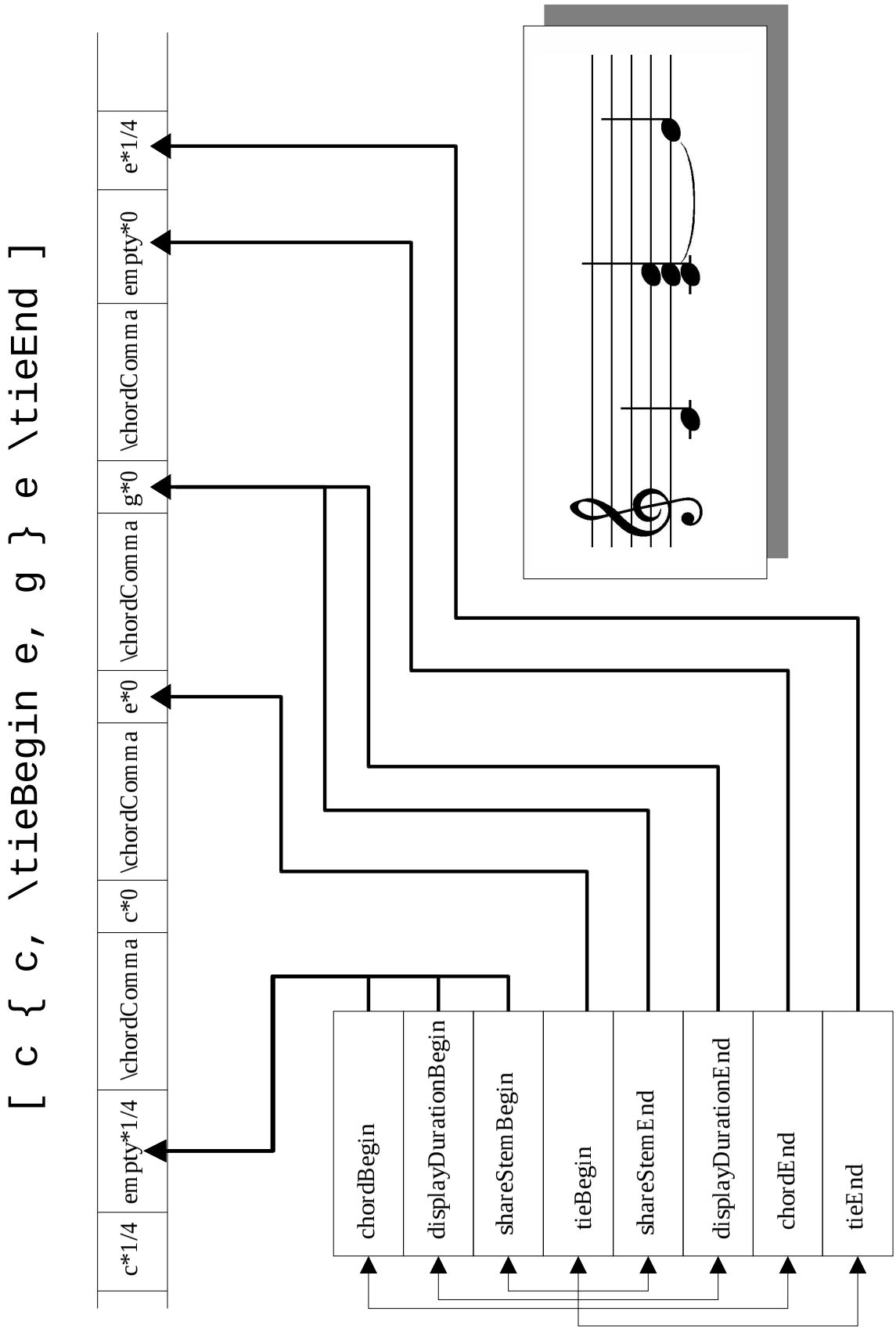


Figure 3.7: Chords as a Series of Events in the Abstract Representation

kit [Hoo] and the AR. During the parse, the hook-functions of the GUIDO parser kit successively call the routines of class `ARFactory`, which then builds the AR.
Class `ARFactory` is responsible for the following tasks:

- Creating one instance of `ARMusic` that is a container for instances of class `ARMusicalVoice`
- Creating one instance of `ARMusicalVoice` for each sequence in the GUIDO description and ensuring that each voice is in GSNF.
- Creating events (notes and rests) and handling the given parameters (duration, accidentals, pitch, and octave)
- Creating range and non-range tags and handling tag-parameters
- Checking that arbitrary pairs of begin- and end-tags are correctly matched.
- Creating the inner chord representation, which was introduced in section 3.3.1.

After the GUIDO description has been parsed, each voice is transformed into GUIDO Semantic Normal Form. This is important, because all further manipulation functions on the AR require the input to be in GSNF; they "guarantee" (contract-model) the output to be in GSNF as well.

Generic Handling of Tag Parameters

As was shown in section 2.2, *Advanced* GUIDO allows different types of tag parameters: one may use named tag parameters, or tag-parameters are assigned by their position in the tag-parameter list. There is also a distinction between optional and required parameters. As each tag has unique parameter names and values, a generic handling of tag parameters was implemented in the notation system. Each tag stores a parameter-string that defines names, types, and default values for the parameters. As an example for such a string consider the parameter string for the `\clef`-tag:
`"S,type,treble,r;S,color,black,o;U,dx,0,o;U,dy,0,o;F,size,1.0,o"`
Each tag parameter is defined by four entries: the first entry is a single character that defines the type of the value. Allowed types are "S" for string, "U" for unit-parameter (the value can be followed by a unit-parameter like "cm", "hs", etc.), "F" for a floating point number, or "I" for an integer number. The second entry defines the name of the tag-parameter. The third entry is the default-value for the tag-parameter. The fourth (and last) entry defines, whether the entry is required ("r") or optional ("o"). For the example above, there are five possible tag-parameters: the first is a required ("r") string ("S") with the name "type" and a default value of "treble". The second parameter is an optional ("o") string ("S") with the name "color" and a default value of "black". The third parameter is an optional ("o") unit-value

("U") with name "dx", default value "0". The fourth parameter is similar to the third; the fifth parameter is an optional ("o") float-value ("F") with name "size" and a default value of "1.0". A GUIDO description may contain the following `\clef`-tag: `[ \clef<type="bass",size=0.75,color="blue"> ]` which defines a bass-clef with 75 % of the regular size and a blue color.
It is possible to define more than one parameter-value string for each tag, so that different parameter-constellations may be used in parallel. Class `ARFactory` uses some helper classes to ensure that a tag in the given GUIDO description is checked against all tag-parameter-lists. Using this mechanism, it is quite easy to implement arbitrary parameter names and values for any of the defined GUIDO tags.

3.5 Accessing the AR

The conversion of a GUIDO description into the AR can be accomplished quite easily, as could be seen in the previous section. When dealing with music notation algorithms (or algorithms from other domains) it is necessary to access the data contained in the AR. As GUIDO descriptions usually are collections of GUIDO sequences, the access mostly concentrates on the voices (class `ARMusicalVoice`). Some algorithms (as will be seen in chapter 4) also require the parallel access to all voices. In this section, the different modes of accessing the AR will be introduced.

3.5.1 Accessing class `ARMusicalVoice`

Class `ARMusicalVoice` was introduced in section 3.3; as it was described there, class `ARMusicalVoice` is a container for other objects of type `ARMusicalObject`. The realized container type is a doubly linked list; therefore all access functions to information contained in the voice operate are commonly known list functions. The current state of an iteration of class `ARMusicalVoice` is stored in an instance of class `ARMusicalVoiceState`. All functions that access class `ARMusicalVoice` therefore require one instance of class `ARMusicalVoiceState`. This class stores all currently active state-tags,[20] all currently active range-tags, and also information about the current (time-)position within a measure, if a meter is specified for the voice. As indicated in section 3.3.1, two different access-modes exist for accessing an instance of class `ARMusicalVoice`: event-mode and chord-mode. In event-mode, all events are iterated sequentially. In chord-mode, all events that belong to one chord are iterated in one step. The following four functions are used for sequential access:

1. `GetHeadPosition(ARMusicalVoiceState &curvst)`: This function is used to start a new iteration on the current voice. The function returns the

[20] State-tags are tags that define the musical state of a voice; these include clef, key, or meter and information.

start position of the element list and resets the voice state information in `curvst`.

2. `GetNext(POSITION pos,ARMusicalVoiceState &curvst)`: This function iterates to the next position in the element list and updates the state information in `curvst`. The object at the current position of the element list is returned.

3. `GetAt(POSITION pos)`: This function just returns the object at the current position of the element list.

4. `GetPrev(POSITION pos,ARMusicalVoiceState &curvst)`: This function iterates to the previous position in the element list and updates the state information in `curvst`. This function may only be called, if the element at the current position of the element list is an event (and not a tag). Otherwise it may be that the state information in `curvst` is not be updated accurately.

By storing all iteration-relevant information in one instance of class `ARMusicalVoiceState`, it is possible to completely save the state of a voice. This is very helpful, when multiple iterations on the same voice are carried out in parallel.

3.5.2 Parallel Access

Parallel access to all voices contained in an instance of class `ARMusic` is sometimes necessary. To simplify this iteration process, a helper class called `ARVoiceManager` was developed. For each voice in the current instance of class `ARMusic`, one instance of class `ARVoiceManager` is created. When iterating thru all voices in parallel, all the instances of class `ARVoiceManager` are used; they do not only return the elements contained in the voice, but also information on the current time-position and the current state of a voice (for example, if the next entry in the element list is an event or a tag). This information is important for ensuring that the iteration is truly a parallel access. To give an example for a parallel access, Figure 3.8 shows a GUIDO description and the matching score. The order in which the elements are iterated when doing parallel access is documented by the numbers in the GUIDO description and the graphical score of Figure 3.8. Note that for a given time position, first all tags are read. Only after all tags at the time position have been read, the events at the time position are processed.

3.6 Manipulation of the AR

The definition of the AR in section 3.1.2 included efficient manipulation functions. In order to define algorithms for music notation as transformations of the AR, some generic operations on the AR are required, which will be described in this section.

```
{          1              3          6    9  10   12   14
  [ \clef<"treble"> \meter<"3/4"> c2/4. h1/8  |  c2/2  _/4  ],
           2          4          5                7          8    11  13
  [ \clef<"bass"> \key<"D"> \meter<"3/4"> { d0/4,  f# } { c, e } |  c/2. ]
}
```

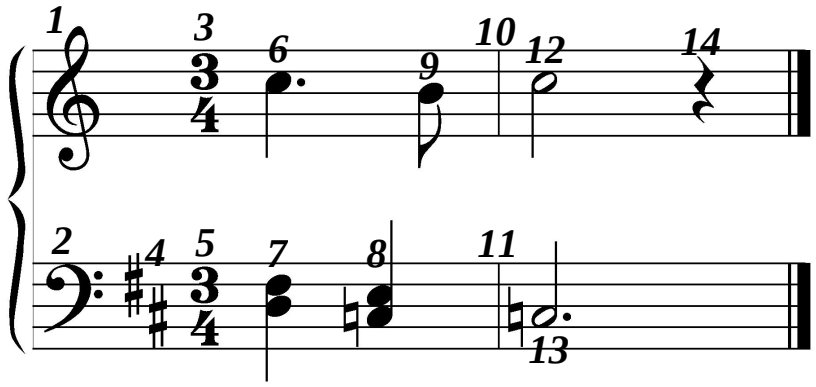


Figure 3.8: Parallel access to the AR

Note that all operations described here require the AR to be in GSNF and likewise ensure that the returned AR is in GSNF as well.

3.6.1 Operations on range tags

Operations on range tags exclusively modify the list of position tags (internally called `ptaglist`; in Figure 3.5 the list is called “Range tag list”) of class `ARMusicalVoice`. All of the manipulation routines must ensure that the correct order of the `ptaglist` is maintained. The `ptagpos`-pointer, which remembers the current position within the `ptaglist` for an iteration, of class `ARMusicalVoiceState` must also be set correctly. This position depends on the current position in the element-list. The following manipulation routines for range tags are present in the AR.

Removing range tags There are situations, where range tags need to be removed from the AR. Removing one tag actually removes two entries from the `ptaglist`, as there is always a begin- and a matching end-tag. The removal-operation can be carried out in a straightforward way: the begin- and end-tags are removed from the `ptaglist` and the `ptagpos` is set accordingly: if the current position within the `ptaglist` was set on a position-tag that is being removed, the position is set to the following position-tag. If there is no following tag, the position is set to `NULL`, which

is interpreted as a traversal beyond the end of the list. Reading Figure 3.9 from top to bottom visualizes the removal of a the \cresc-tag.

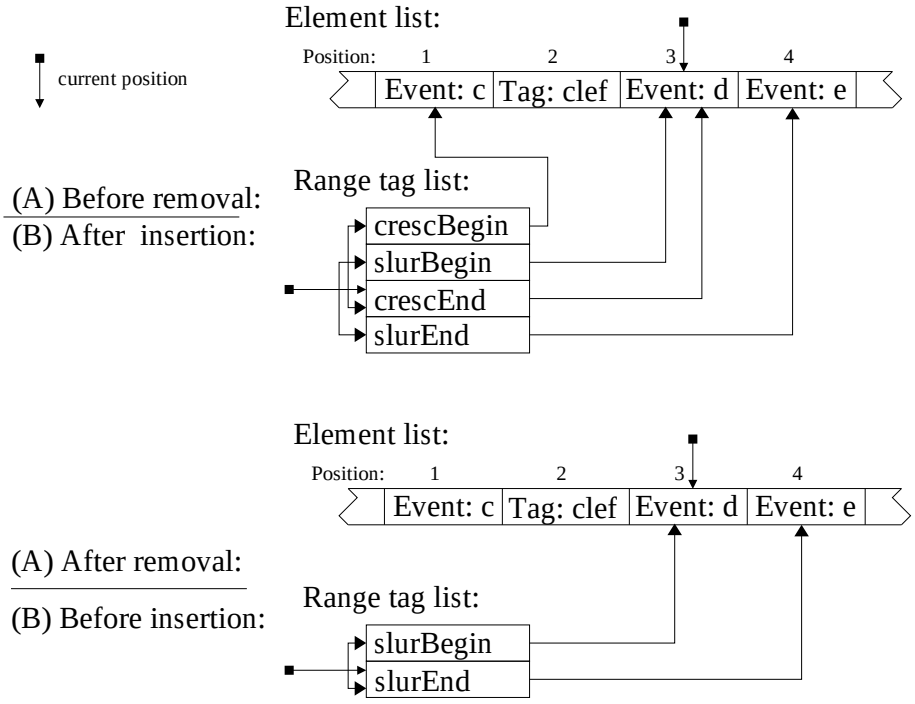


Figure 3.9: Visualization of range tag removal (A) and range tag insertion (B)

Inserting range tags When inserting range tags (like for example a \tie-tag), the begin- and end-position of the range in the element list is needed, so that the correct location in the ptaglist can be found. The actual insertion of the two instances of ARPositionTag – the begin- and the end-tag – can then be done easily. Some care has to be taken, if there are no previous range tags present in the AR, in which case the ptaglist must be created. Reading Figure 3.9 from bottom to top visualizes the insertion of a \cresc-tag.

Changing the start and end positions This operation is mainly required, when events in the element list are being removed or split. Because the position tags in the ptaglist store pointers to locations in the element list, all position tags pointing to such events must be updated to match the new position in the element list. As long as the order of the ptaglist is not disturbed, the re-pointering is a straightforward process. Otherwise, the order of the elements in the ptaglist must be changed to match the new situation. Figure 3.10 shows the effect of changing the positions within the ptaglist in order match the situation in the element list: before the change, the \slur and the \beam-tag both ended at the same event. After

the change, the \slur-tag now ends before the \beam-tag. The order within the ptaglist is changed.

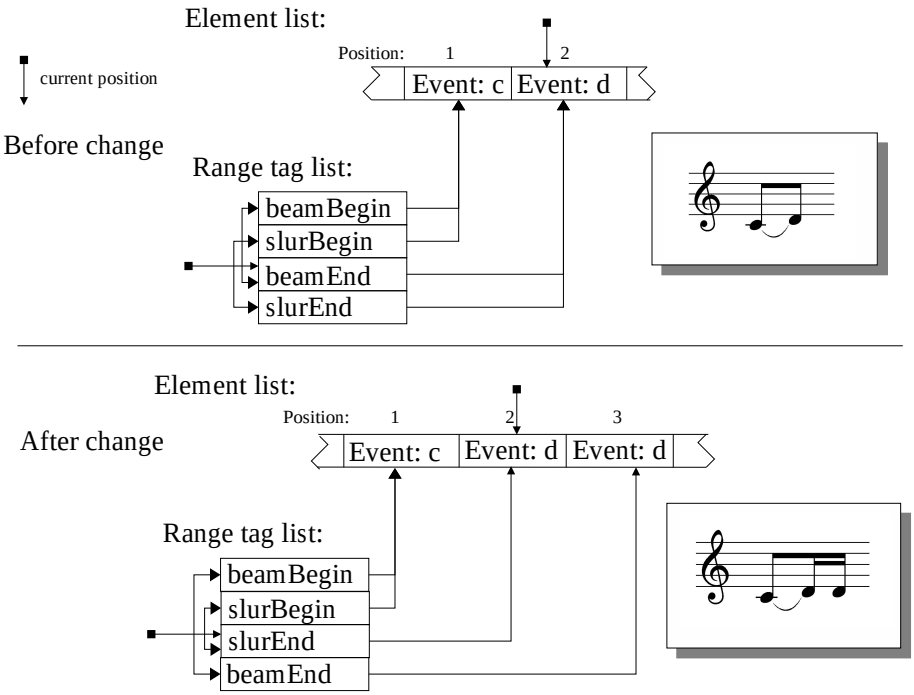


Figure 3.10: Changing the order within the range tag list

3.6.2 Operations on events and non-range tags

The operations on events and non-range tags modify the element list and, in some cases, also the position tag list (ptaglist) of ARMusicalVoice. When manipulating events and non-range tags, the operations must assert that the correct (time) order of the elements in the element list is maintained and always forms a monotone rising time function. The following manipulation functions are included in the AR:

Inserting and removing non-range tags Sometimes, non-range tags have to be inserted into the AR. This happens, for example, when \clef- or \key-tags are automatically added after a \newSystem- or \newPage-tag. As non-range tags are never associated with range-tags, and therefore are never pointed to be any entry in the ptaglist, the process of inserting and removing non-range tags can be done directly by inserting or removing the new tag in the element list at the appropriate time position.

Splitting events When dealing with automatically inserted bar lines or \newSystem- or \newPage-tags, it is sometimes necessary to split one event into two events. This happens, if the new bar-line has to be inserted at a time position that lies *within* the duration of an event. If an event is split, the resulting events need to be joined by a \tie-tag, so that the *musical* idea remains the same. Splitting events is a manipulation that is required often, especially when a time signature is set and automatic bar lines are inserted. To split an event, the duration of the resulting two events must be known. If the original event has duration d, then $d = d_1 + d_2$ must hold, where d_1 and d_2 are the durations of the two resulting events. First, the original event changes its duration to d_1. Then a new event with duration d_2 is inserted (using the routine described below). Finally, the position tags must be "repointered". Depending on the tag-type, tags that began at the original event can now either begin at e_1 or at e_2. Figure 3.11 shows an example of splitting an event because of a \bar-tag.

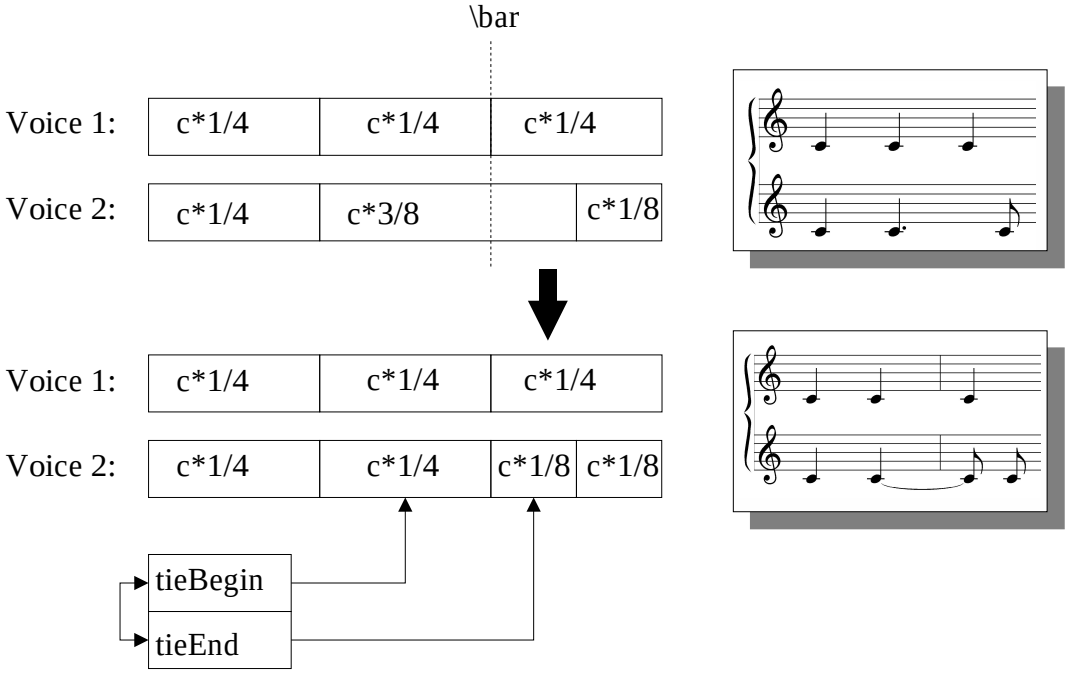


Figure 3.11: Splitting an event

Because of the special inner representation of chords (see section 3.3.1), the process of splitting a chord into two chords is much more complicated. In this case, all elements of the chord must be copied and joined by \tie-tags. Because some tags within the chord may alter the *state* of the voice (for example a \staff-tag, that changes the staff on which the voice is displayed), the state of the voice at the first element of the original chord must be saved. This saved state information must be restored before the second chord is inserted. Figure 3.12 shows a chord containing

a \staff-tag and the implications for splitting this chord. All added or changed information is displayed in bold type. Note that a new \staff-tag is inserted at the beginning of the second chord. In order to focus on the main points, the Figure does not show the inner representation but uses the equivalent GUIDO descriptions instead. Internally, the inner chord representation of section 3.3.1 is used.

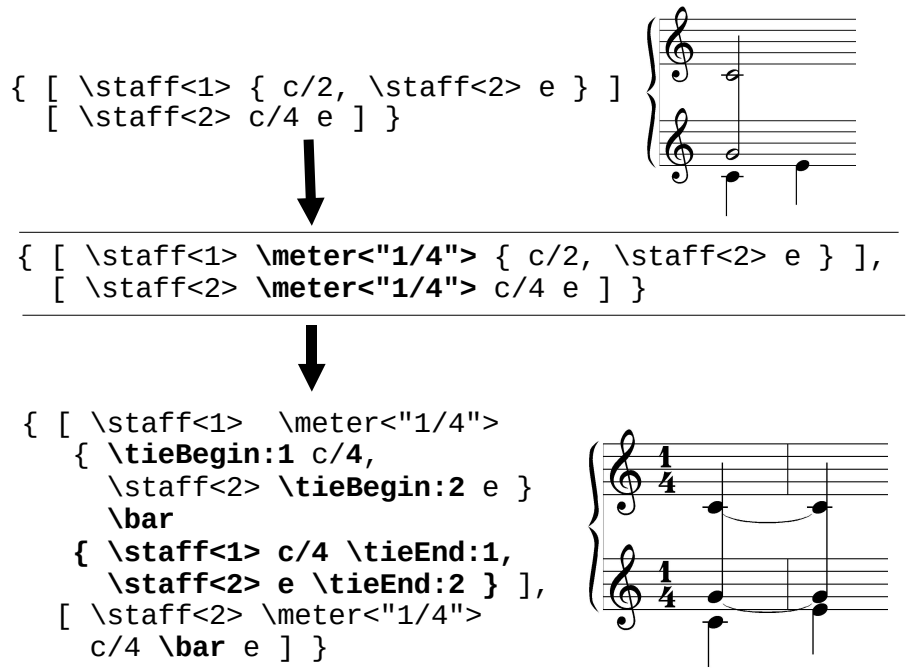


Figure 3.12: Splitting a chord

Inserting Events The insertion of new events into the AR can be accomplished rather easily. Once the location for inserting the event into the doubly linked element list has been found, it can be easily inserted. Because all elements of the list store their time position, all elements following the inserted event must increase their time position by the duration of the inserted event. Because the event is stored at a new position within the element list, the range tags are not affected by this change.

Removing Events The case of removing events is a little bit more complicated than the insertion of events, because the range tags may have to be considered as well. At first, the same premises as in the insertion case hold. The event that is about to be removed has a position in the doubly linked list. When it is removed, the following elements have to update their time position. Then it has to be checked, whether any range tag begins or ends at the removed event. Depending on the tag,

it can either be removed from the position tag list, or its position can be changed to either the previous or the following event. This depends on the type of the tag and can not be generalized. As an example consider the GUIDO description
`[ \cresc( c d e ) f ]`.
If the `e` is removed, the resulting GUIDO description looks like
`[ \cresc( c d ) f ]`.
In this case, the `\cresc`-tag updates its range to the previous event.

3.7 Conclusion

In this chapter an application-oriented inner representation for GUIDO descriptions called Abstract Representation (AR) was introduced. It was shown that the AR can be regarded as the document in the context of the Document-View-Model, a design pattern that helps in encapsulating the data from a concrete type of display or manipulation. In order to facilitate the handling of GUIDO descriptions, the GUIDO Semantic Normal Form (GSNF) was identified and the conversion of arbitrary GUIDO descriptions into GSNF was described. The GSNF is a normal form that leads to a clear distinction of GUIDO elements: events, non-range tags, and range-tags. In the following, the object oriented design and the structure of the AR was described in some detail. The main classes of the AR were presented, and it was shown, how GUIDO descriptions are converted into the AR. The final section of this chapter dealt with access and manipulation of the AR. All of the described routines are required by the notation algorithms, which will be described in the following chapter.
Generally, applications dealing with any form of music representation need a suitable inner representation. In this chapter, it was shown that the AR offers such a flexible and powerful representation for GUIDO descriptions. As already stated above, the AR can not only be used for music notation purposes, but is a complete inner representation for arbitrary GUIDO descriptions. The contained manipulation functions and routines make it useful for a whole range of computer music systems.

Chapter 4

Converting Arbitrary GUIDO Descriptions Into Conventional Scores

Introduction

The conversion of an arbitrary GUIDO description into a conventional score is a process requiring a multitude of music-notation algorithms which will be described in detail in this chapter. As was shown in the previous chapter, there is a one-to-one correspondence between GUIDO descriptions and the Abstract Representation (AR). Because the AR is a powerful data structure, which was developed to allow direct and efficient manipulations, many of the notation algorithms described in this chapter work directly on the AR. Basically, all conversions needed to convert the AR into a conventional score can be described as GUIDO to GUIDO transformations: Beginning with an arbitrary GUIDO description and its corresponding AR, a series of algorithms add notationally relevant information to the AR (like for example automatically creating bar lines, if a time signature has been encoded in the GUIDO description). After these transformations, the AR is converted into the so called "Graphical Representation" (GR), which is an object oriented class structure whose classes closely correspond to the visible elements of a graphical score. Similar to the AR, the GR and all necessary algorithms have been developed as part of this thesis. The conversion of the AR into the GR is a two step procedure: first, the graphical elements are created from the data contained in the AR, then another set of music notation algorithms is responsible for finally placing the notational elements on lines and pages.

This chapter is structured as follows. First, the notion of GUIDO to GUIDO transformations is explained in detail. Then, generic music notation algorithms are classified and their scope – which structural representation level they use – is identified; the implemented algorithms are presented. After that, the Graphical Representa-

tion (GR) is introduced, and it is shown, how the AR is converted into the GR. The conversion process automatically determines all information that is necessary to build conventional score. Finally, conventional and newly developed algorithms for spacing, line breaking, and page-filling are explained in detail. Where appropriate, similarities to text processing are mentioned.

4.1 Music Notation as GUIDO to GUIDO Transformations

Arbitrary GUIDO descriptions contain musical entities (like notes, rests, and musical and graphical markup) that are not necessarily closely related to one single element of a graphical score. When creating a visual score representation from an arbitrary GUIDO description, the entities of the AR are modified (split, copied, added) to more closely represent single graphical elements present in the visual score. Figure 4.1 shows an example of how the GUIDO description (a) can be transformed into the GUIDO description (b), which much more closely resembles the actual graphical appearance (c). In this example the `\meter`-tag forces bar lines to appear in the score; the bar lines split musical notes into two graphical notes, joined by a tie. Even though the *musical* information of descriptions (a) and (b) is similar, description (b) can be used far more easily to create the graphical elements of the score.

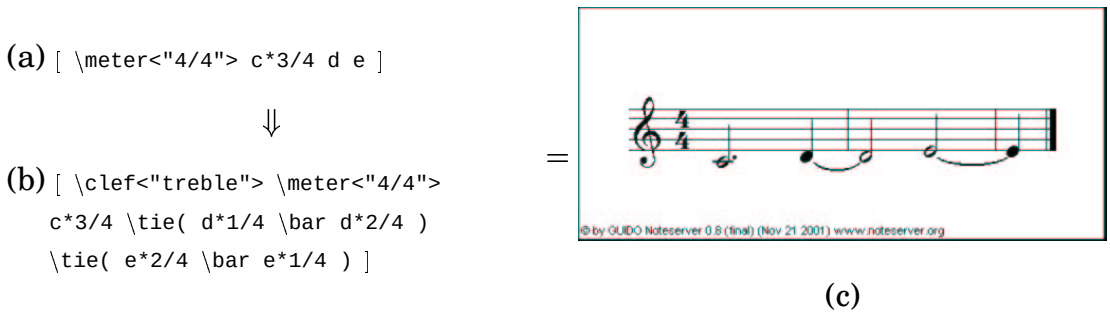


Figure 4.1: Simple GUIDO to GUIDO transformation

The notation system, which was developed as part of this thesis, was implemented in such a way that a GUIDO description is first converted into a corresponding Abstract Representation (AR).[1] Then, a number of notation algorithms, which will be described in detail in the next section, manipulate the AR. These algorithms are all GUIDO to GUIDO transformations: they get a GUIDO description as their input and return an enhanced or modified GUIDO description as their output. Each algorithm only manipulates a strictly defined musical or notational aspect of a score. Then the AR is converted into the Graphical Representation (GR). All elements of the GR are

[1] As indicated before, this correspondence is a one-to-one relationship.

directly connected to elements of the AR; this is important so that all further algorithms, which directly manipulate the GR, can still be described as GUIDO to GUIDO transformations: the manipulation of an element of the GR is directly reflected in the corresponding element of the AR.
Figure 4.2 shows, how a GUIDO description is first converted into the AR, which is then manipulated through a set of notation algorithms. Afterwards, the AR is converted into the GR, where it is again manipulated through another set of algorithms. The GR can then be used to directly render a score – either within an application or as a graphics file. The results of all notation algorithms are stored within parameters of the AR; the corresponding enhanced GUIDO description contains a textual representation of the completely formatted score.

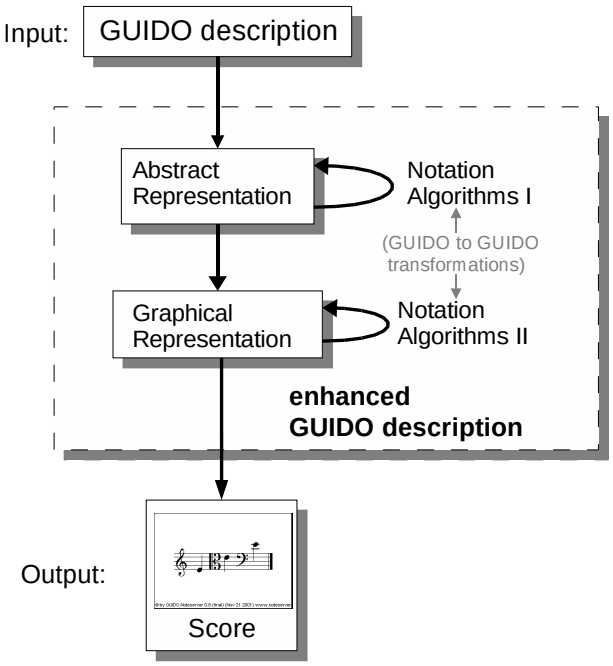


Figure 4.2: General approach for GUIDO to GUIDO transformations

Because each notation algorithm deals only with strictly defined musical or notational aspects of a score, it is possible to exchange some of the algorithms to get a different score. As an example, consider the algorithm that decides on automatic beam groups: the “standard” algorithm uses the current meter to decide on beam groups. Another algorithm may use a Neural Network approach to find beam groups.
The *musical* information present in the original GUIDO description is not changed by the notational transformations. The original information is rather intensively used to to derive graphical “knowledge” that is added to the original description. In conjunction with built in notational knowledge (spacing, line-breaking, etc.), the collection of GUIDO to GUIDO transformations build the core of the notation renderer.

The concept of describing the creation of a conventional score as a sequence of GUIDO to GUIDO transformations distinguishes the presented implementation from other notation software: the contained notational knowledge can be extracted by studying the results of individual transformation routines. Because every transformation routine returns a graphically enhanced and valid GUIDO description (or at least the equivalent internal representation of it), no explicit knowledge of the internal representation is required to understand the single transformational steps. It is also possible to exchange only some of the transformation algorithms to change the overall appearance of the score. When all notation algorithms have dealt with a given GUIDO description, the notation system finally outputs a completely formatted *Advanced* GUIDO description. If the original description already was a complete *Advanced* GUIDO description, the algorithms within the notation system will not add or change any information.
In the next section, the different music notation algorithm will be described as GUIDO to GUIDO transformations.

4.2 Music Notation Algorithms

The process involved in music setting has been classified by Don Byrd [Byr84] as involving three distinct steps:[2]

1. *Selecting*: deciding what symbols to print.
2. *Positioning*: deciding where to print the symbols
3. *Printing* the symbols

These three steps have to be performed in any music processor that converts musical information into a printed score. When considering interactive music notation systems, some of these steps are not clearly defined: an interactive system may not require the first step, because in this case, the user directly decides on the symbols.[3] Also, in interactive systems, the positioning of symbols is usually done semi-automatically: the system applies internal notational knowledge to place the symbols first; then the user can manually adjust their graphical positions if needed. The printing process has vastly improved since 1984 (when Don Byrd wrote his thesis); today, high resolution laser printers are available at little cost even for home users. Nevertheless, there are still issues of device-independent output that have to be considered – especially if the notation system is used on screen *and* for printing. The music notation algorithms, which will be discussed in this section, may be classified according to the three steps from above. One class of algorithm deals mainly

[2] Pages 74–75

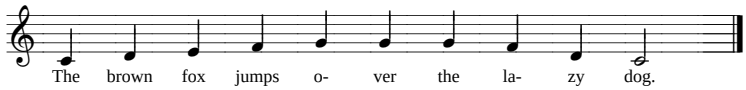
[3] Sometimes it is still required to break one musical note into several graphical entities as in the case of automatically added bar lines.

with the *selection* process: this relates to the first step mentioned above. These algorithms work on the inner representation only; they can thus be described as AR to AR transformations. As an example for this class consider the algorithm, which decides upon which notes to group together for beaming (the group selection depends mainly on the current meter). Another class of algorithm deals with the *positioning* of symbols: this relates to the second step from above. These algorithms require knowledge of the graphical environment (for example the width and height of the page, the width and height of a staff-line, of a note, etc.). These algorithms mainly work on the graphical representation. In the following, the notation algorithms on the AR level are explained. The algorithms that manipulate the Graphical Representation will be presented in sections 4.4 and 4.5.

Music Notation Algorithms on the AR level

In the following, the music notation algorithms, which manipulate the AR are presented. These algorithms perform a sort of pre-processing of the musical data given in the original GUIDO description. The intent of all algorithms is to ensure a strong coupling of the Abstract Representation with the notation elements of the conventional score. Most algorithms on the AR level deal only with a single voice (GUIDO sequences), one algorithm deals with the whole piece (GUIDO segments). The order in which the algorithms are carried out is crucial, because implicit dependencies are present: some algorithms depend on the output of other algorithms. Figure 4.3 shows, which algorithms have to be performed before others. This is denoted by the arrows: an arrow pointing from algorithm a to algorithm b means that a has to be computed before b.

- **AutoDispatchLyrics** This algorithm goes through a single voice and dispatches the lyrics of a `\lyrics`-tag onto the events within the tag range. It is essential that this step is performed *before* any event is modified (which may happen in the AutoBarlines routine), because the `\lyrics`-tag is closely tied to the original GUIDO description.
  Example:[4] The GUIDO description:
  ```
  [ \lyrics<"The brown fox jumps o-ver the la-zy dog."> ( c1/4 d e f g g g f d c/2 ) ]
  ```
  with the matching score



[4]All examples in this section are given using GUIDO descriptions. In the actual implementation all algorithms of this section work directly with the (inner) Abstract Representation. For a better understanding of the functionality of the algorithms, the equivalent GUIDO descriptions are used.

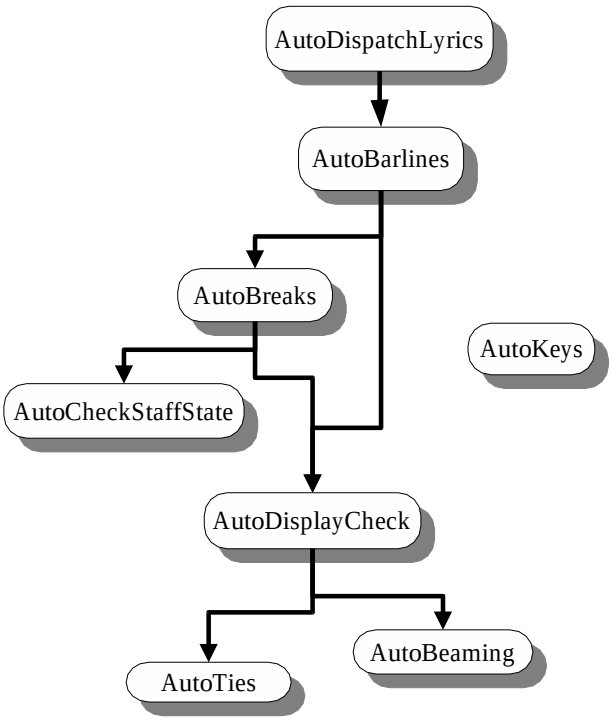


Figure 4.3: Dependencies of music notation algorithms

is converted to

```
[ \text<"The">( c1/4 ) \text<"brown">( d ) \text<"fox">( e ) \text<"jumps">( f )
\text<"o-">( g ) \text<"ver">( g ) \text<"the'>( g ) \text<"la-">( f )
\text<"zy">( d ) \text<"dog.">( c/2 ) ]
```

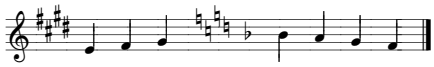
- **AutoKeys** This routine iterates through a voice and checks for changes in key-signature. This is done, because if the key-signature changes in a score, the old key-signature must be naturalized. This is done by placing a `ARNaturalKey`-object before the changed key, which will later be used to create the corresponding graphical naturalization elements in the score. This routine does not depend on any other routine.
  Example: The GUIDO description:

  ```
  [ \key<"E"> e1/4 f# g# \key<"F"> h& a g f ]
  ```

  is converted to

  ```
  [ \key<"E"> e1/4 f# g# \natkey \key<"F"> h& a g f ]
  ```

  [5]



- **AutoBarlines** This is a very important routine that iterates through a voice and automatically inserts bar lines based on the current meter. The algorithm looks for meter changes and also for explicitly specified bar lines in the original

[5]The `\natkey`-tag is an internal tag which is not part of the official GUIDO specification. It is only used internally to create the graphical elements for the naturalization key in the score.

GUIDO description. If a time position tp for a new automatic bar line has been determined, the algorithm must check, whether an event is currently active: if the onset time of the current event is earlier than tp and the end time of the current event is later than tp, the event must be split into two events, and an additional `\tie`-tag must be added to graphically join the split notes. The algorithm makes use of all manipulation functions provided by the AR, which are described in section 3.6 on page 60.
An **example** for the AutoBarline routine can be seen in Figure 4.1 on page 68. All of the following algorithms depend directly or indirectly on the output of this algorithm.

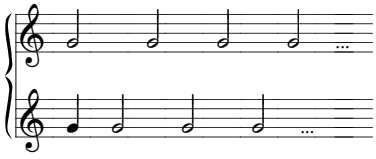
- **AutoBreaks** This is the only one of the notation algorithms, which requires a parallel sequential access of all voices contained in the AR as it is described in section 3.5.2. The AutoBreaks-algorithm does two things: first, it "multiplies" `\newSystem`- and `\newPage`-tags; this means that if a `\newSystem`- or a `\newPage`-tag is encountered in one voice, a similar tag is added at the same time position in all other voices, if no explicit break tag is already present there. This may result in events being split (similar to the AutoBarlines algorithm from above) in some voices. The second task this algorithm performs, is to determine "possible break locations": when traversing the voices in parallel, the algorithm looks for possible break positions, which are vertical cuts through all voices of a line of music that can later be used as line break positions. For conventional scores, good possible break locations usually are the common bar lines of all voices. In order to work for all kinds of music, the AutoBreaks-algorithm does not rely solely on bar lines but evaluates how "good" a specific time position within a specific voice is suited as a break position. The "goodness" of a break position for a voice is returned as a floating point number that is calculated within each voice by taking into account the current measure position[6] – a time position at the very end of a measure is better suited than a time position within a measure – and checking, whether an event is active at the given time position. By adding the results for all voices for a given time position, it can be determined, if the time position is a suitable possible break location. If such a location is found, a `\pbreak`-tag (short for potential break) is inserted in all of the voices;[7] sometimes, an event must be split using the manipulation routines of the AR, because of an added potential break. For conventional scores that have common time signatures for all voices, this algorithm returns very good results. Nevertheless, problems arise, if music that has no common time signatures (or no time signatures at all) is converted into a conventional score. One can easily construct music that

[6] Note that each voice may have a different time signature. Even though this is not very common, it is easily possible to specify such music in GUIDO.

[7] The `\pbreak`-tag is again an internal tag that is only used within the described notation system; it is not part of the official GUIDO specification.

contains no “natural” break locations. Consider the following (partial) GUIDO description and the matching score. In this case, the two voices have no common onset times, because all notes overlap each other.

```
{ [ g/2  g g g ... ] ,
[ g/4 g/2 g g g ... } ]
```



In this case, there are no good possible break locations. One solution for the AutoBreaks-algorithm is to simply leave the issue to the user: no automatic line breaks would be calculated. Another solution is to simply add \pbreak-tags, if for a certain amount of musical time no “natural” breaks were added to the music. This sometimes leads to falsely split events, because events are split at time positions that are later not used as real line break positions; at least, this solution returns a readable score. The current implementation uses the second approach.

- **AutoCheckStaffStateTags** This algorithm iterates through a voice and does two things: first, it checks, if a \clef-tag is present at the very beginning of the voice. If not, a \clef is added, because conventional score display is not possible without a clef that determines the pitch of the notes being displayed. The algorithm also sorts the clef, key, and meter symbols at the beginning of a voice; this feature can be turned of by an explicit flag. The second task, the algorithm performs, is to check, whether \newSystem- and \newPage-tags are followed by correct clef- and key-signatures. If no \clef or \key-tag is found, they are automatically added to the voice using the previous valid clef and key-signature of the voice. This is important because the line-break algorithm requires the first line after an explicit \newSystem- or \newPage-tag to contain complete clef and key information.

  This algorithm obviously depends on the AutoBreaks algorithm, because it looks for \newSystem- and \newPage-tags, which might have been added in the AutoBreaks routine.

  Example: the GUIDO description

  ```
  [ \clef<"treble"> \key<"F"> c d e \newSystem f g ]
  ```

  is converted to

  ```
  [ \clef<"treble"> \key<"F"> c d e \newSystem \clef<"treble"> \key<"F"> f g ]
  ```

- **AutoDisplayCheck** This algorithm iterates through a voice and checks, if the events in the voice are displayable as single graphical elements, which means that there is a one-to-one correspondence between one abstract event and one notational element of the conventional score.

There are two reasons for an event not being directly displayable: either the duration has a non-regular duration, which means that the denominator in the events duration is not a multiple of two, or the events duration requires more than one graphical element, because the numerator of the events-duration is not equal to one, three or seven (if it is three, or seven, dots can be used to display the event). As an example for these two cases, consider the following table, where the original GUIDO description and the resulting conversions are shown.

Input	Output
`[ c*5/8 ]`	`[ \tie( c*1/2 c*1/8 ) ]`
`[ e*1/6 f/12 ]`	`[ \tuplet<2,3>( e*1/6 f/12 ) ]`
`[ e*11/37 ]`	`[ \tuplet<22,37>( e*11/37 ) ]`

In order to be able to display irregular durations, a parameter called display-duration-base (ddb) was introduced. Usually, notes or rests in the graphical score have durations, which are fractions of multitudes of 2^{-k} (with $0 \leq k < 8$) as in $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, etc. The ddb is used as a multiplication factor to determine the display-duration for an irregular duration. Let d_{ev} be the (irregular) duration of an event. Then the display duration d_{disp} of the graphical note (or rest) is determined by

$$d_{\text{disp}} = ddb \cdot d_{\text{ev}}$$

The ddb is set by the `\tuplet`-tag. Initially, the ddb is set to one. As an example consider a regular triplet: here, the ddb is set to $\frac{3}{2}$, so that the display duration for the event `c*1/12` is calculated as

$$\frac{3}{2} \cdot \frac{1}{12} = \frac{1}{8} = d_{\text{disp}}$$

which means that the event is displayed as an eighth-note. This mechanism works for *any* event duration: it is just a matter of setting the ddb to convert the events duration into a displayable regular note. The algorithms flowchart is shown in Figure 4.4 on the right.

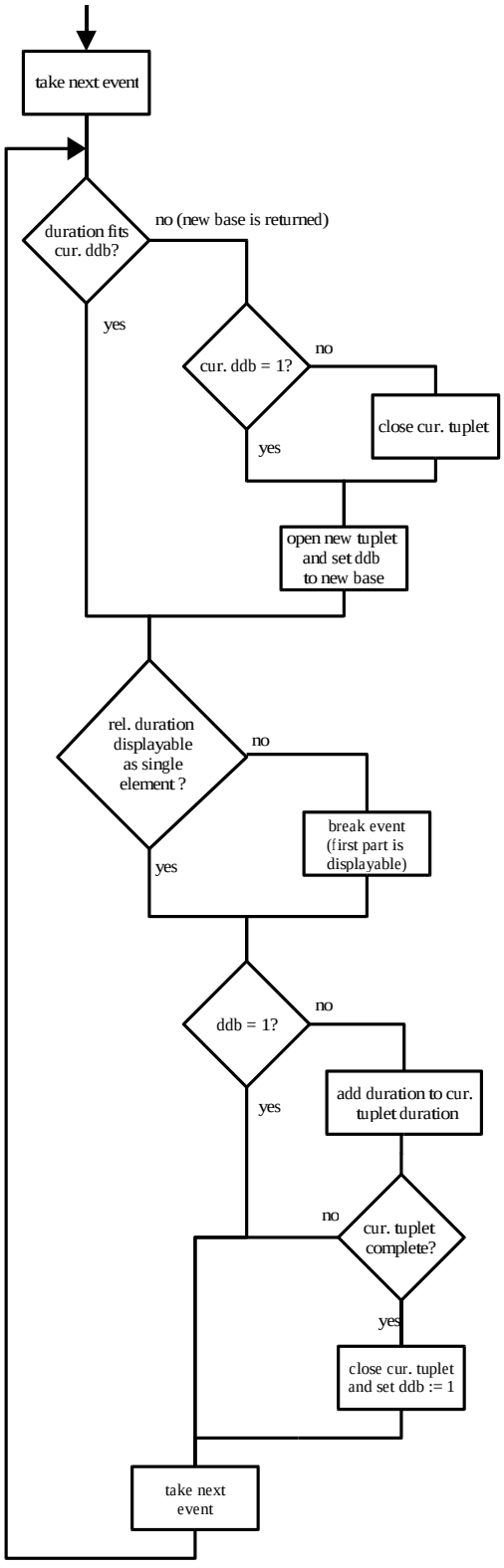
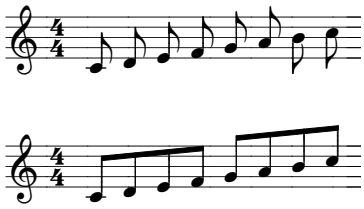
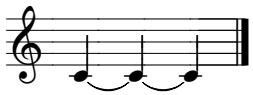


Figure 4.4: Flowchart AutoDisplayCheck

- **AutoBeaming** This algorithm iterates through the events of a voice and determines, which notes can be part of a beam-group. The algorithm strongly depends on the current meter and also respects explicitly specified beams.

  Example: the GUIDO description
  ```
  [ \meter<"4/4"> c1/8 d e f g a h c2 ]
  ```
  is converted to
  ```
  [ \meter<"4/4"> \beam( c1/8 d e f )
    \beam( g a h c2 ) ]
  ```
  

- **AutoTies** This algorithm iterates through a voice and looks for `\tie`-tags. As many of the previous algorithms automatically split events and join them using `\tie`-tags, this algorithm has to be computed very late. The algorithm breaks all `\tie`-tags, which cover more than two events into a series of `\tieBegin`- and `\tieEnd`-tags that cover two events each. After the algorithm has been computed, any `\tie`-tag found in the AR can be represented by one single graphical tie in the score.

  Example: the GUIDO description
  ```
  [ \tie( c/4 c c ) ]
  ```
  is converted to
  ```
  [ \tieBegin:1 c/4 \tieBegin:2 c \tieEnd:1 c \tieEnd:2 ]
  ```
  
  Corresponding Score

Music Notation Algorithms on the GR level As indicated in Figure 4.2, another set of music notation algorithms manipulates the Graphical Representation (GR). In order to describe these algorithms, it is essential to first introduce the GR. Afterwards, the other algorithms are described in detail (see sections 4.4 and 4.5).

4.3 The Graphical Representation (GR)

The Graphical Representation (GR) is an objected oriented class hierarchy, in which individual classes actually represent graphical entities seen on the pages of the conventional score. The GR is closely tied to the AR, but there exist quite a number of classes in the GR for which no direct match exists in the AR: the classes `GRStaff`, `GRSystem`, `GRPage`, for example, have no counterpart in the AR. This is not really surprising because an Abstract Representation may consist of a single sequence of events or markup, which needs to be broken into several systems on different pages when it is displayed as a conventional score. The GR does not only contain objects that can be seen on a score but also includes helper (or manager) classes that are needed during the conversion of the AR into the matching GR, and also during the manipulation of the GR.
One important aspect during the design of the Graphical Representation was platform independence. As the implemented notation system should be usable on dif-

ferent computers using different operating systems, the function calls that produce graphical output are encapsulated in a small module, which can be adopted for different graphical operating systems without changing the overall layout of the structure. This issue is not trivial, as each graphical operating system requires different methods to access information regarding for example the width and height of font symbols, or offers different capabilities with respect to virtual coordinate spaces being mapped onto a screen or a printer.
In the following, the graphical requirements for any (conventional) music notation system will be exemplified. Then, the major classes of the GR will be presented.

4.3.1 Graphical Requirements for Conventional Music Notation Systems

Regardless of the underlying data structure, certain requirements are common to all conventional music notation systems. Usually, a conventional score is printed on several pages, containing lines of music (which are sometimes called "systems"), which contain one or more staves. A line of music can be vertically sliced into so called system slices: these slices are important for line breaking, as will be shown later. Notational elements, like notes, rests, or musical markup are put on top of the staves. Figure 4.5 shows one page of a conventional score together with a description of the used notational elements.
When placing a notational element on a page of a score, it is essential to decide on its frame of reference. In most cases, the frame of reference is the staff on which a notational element (like for example a note) is placed. Then, the position of the notational element is specified *with reference to the staff* it belongs to. A single staff has the containing system as its frame of reference, therefore the position of a staff will be specified with respect to the containing system. A system (or line of music) will have the page coordinate system as its reference frame on which it will be placed. Therefore, moving a staff, or a system just requires one explicit change in position, because all underlying elements are then automatically moved. An essential requirement for a conventional music notation system is its ability to reflect the *hierarchical order* of the relevant notational elements. When dealing with object-oriented terminology, this idea is reflected in the usage of so called *containers*, which are capable of storing other elements.

4.3.2 Structure of the GR

After the graphical requirements have been elucidated in the previous section, we can now introduce the object-oriented structure of the Graphical Representation. The developed GR closely follows the requirements shown in the previous section; especially, the hierarchical order for all notation elements are reflected in the design of the GR. Figure 4.6 shows a UML diagram for the major classes of the GR;

One Page of a Score

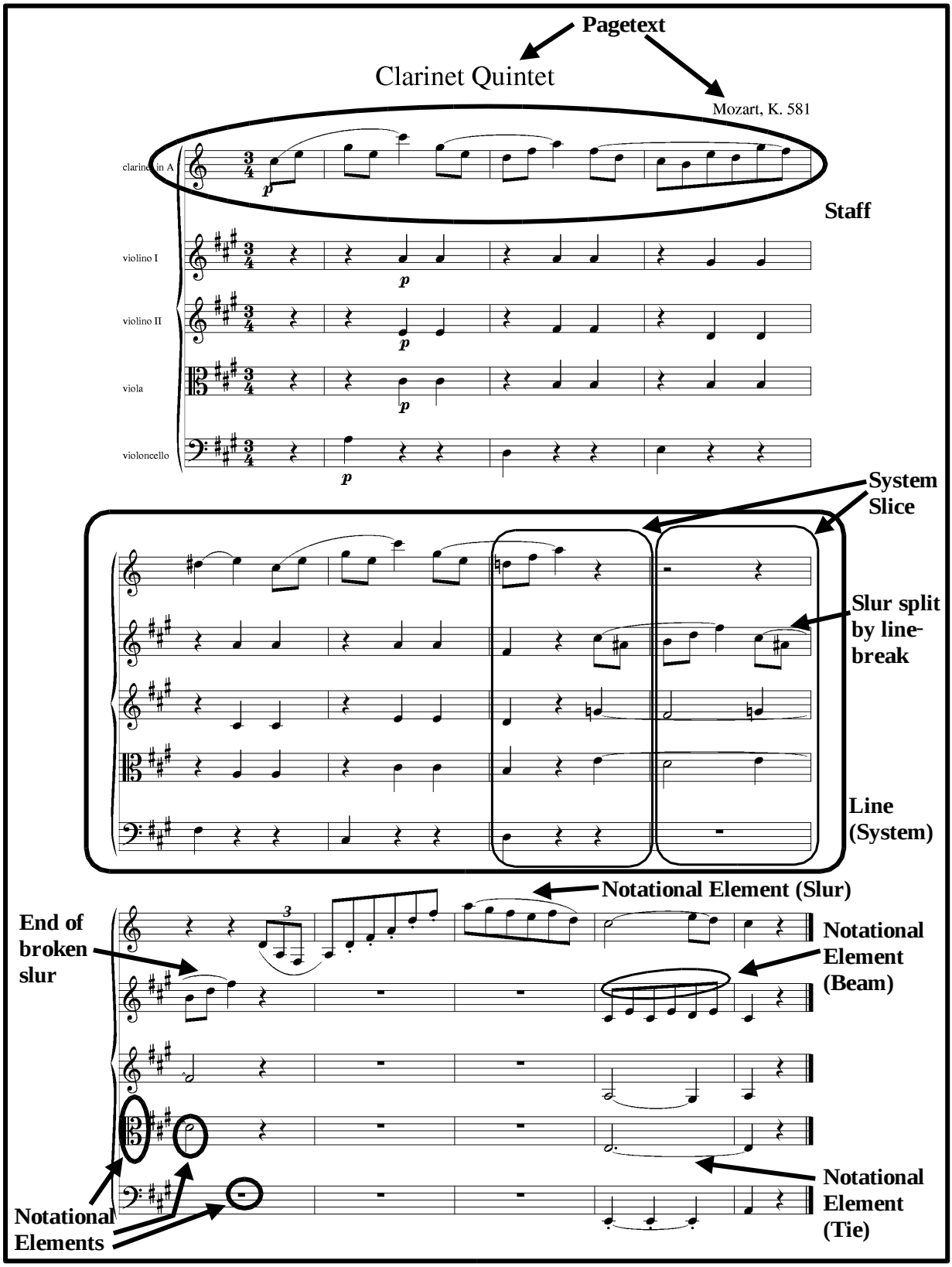


Figure 4.5: Elements of one page of a conventional score

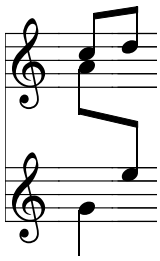
as the current implementation has more than 90 classes, this overview covers only the most important classes. Almost all of the classes in the GR directly or indirectly inherit from class `GObject`, which contains the data and functionality required for any visible element. The data of `GObject` includes, for example, a (graphical) position and a bounding box.

- **Class GRMusic** can be found on the bottom left of Figure 4.6. One instance of this class is created for every score that is being created from a GUIDO description. As can be deduced from the diagram, `GRMusic` contains one or more instances of class `GRPage`, which represent the pages of the score. To create the pages and lines of a score, `GRMusic` "employs" a class `GRStaffManager`, which is not shown in the diagram but will be explained below, when the conversion of the AR into the GR is discussed in detail.

- **Class GRPage** contains one or more instances of class `GRSystem`. An instance of class `GRPage` knows about its height and width and also about the size of its margins. Some graphical elements of the score, like for instance the visible title and composer of a piece are also controlled by class `GRPage`. Class `GRPage` contains one or more instances of class `GRSystem`.

- **Class GRSystem** represents a line of music in the score. It is made up from one or more instances of class `GRSystemSlice`. When the line breaking algorithm has decided, which slices belong to a system, class `GRSystem` must adjust the spacing of the notation elements, and must also adjust its extent to match the desired line width (spacing and line breaking is dealt with in sections 4.5.1 and 4.5.2).

- **Class GRSystemSlice** is a part of a single line of music. A set of `GRSystemSlices` build a `GRSystem`. For a simple understanding, it is convenient to think of system slices as measures of a score. Class `GRSystemSlice` contains one or more instances of class `GRStaff`. Class `GRSystemSlice` is also responsible for graphical elements that belong to several staves; this might be, for example, a beam that begins and ends in different staves, as it is shown on the right.



- **Class GRStaff** represents a part of a single staff of a line of music. The time position and duration of the part being represented by an instance of class `GRStaff` is directly determined by the containing `GRSystemSlice`. Class `GRStaff` directly inherits from class `GRCompositeNotationElement`, which is capable of storing an arbitrary number of instances of class `GRNotationElement`. Using this storage, class `GRStaff` stores graphical elements being placed upon it. This might be musical markup, like for example an instance of class `GRClef` or a musical event like an instance of class `GRNote`.

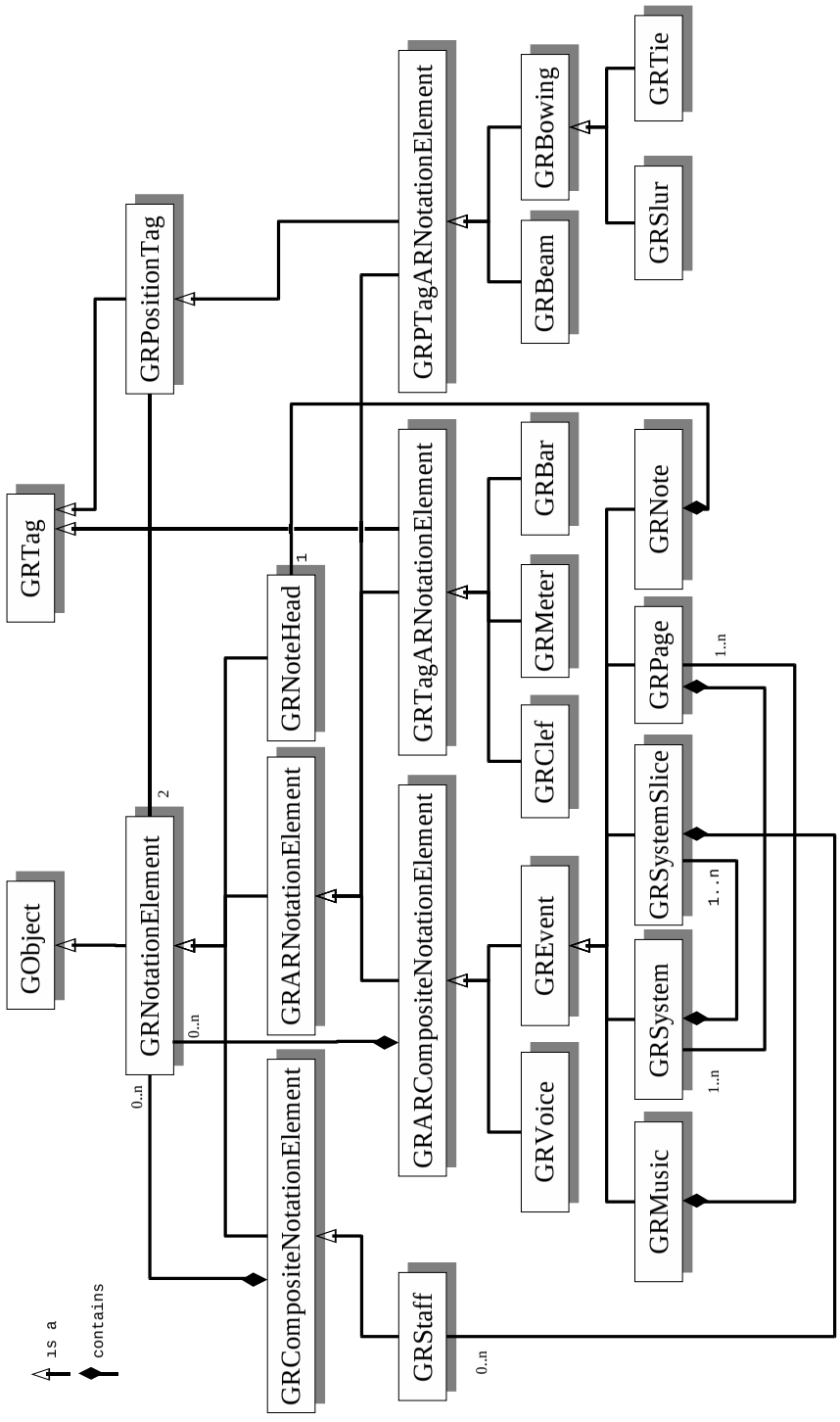


Figure 4.6: Major classes of the Graphical Representation

- **Class GRTag** is the base class for all GUIDO-tags. As can be seen in Figure 4.6, a GUIDO-tag either inherits directly from class `GRTagARNotationElement`, or it inherits directly from class `GRPTagARNotationElement`. This mechanism is used to distinguish between range- and non-range tags (see section 3.2.2 on page 47 for a detailed description). As a range tag has a start- and end-element, class `GRPositionTag` contains two pointers to instances of class `GRNotationElement`. These are used to determine the start- and end-point for a range tag; for example, the beginning and ending note of a slur for class `GRSlur`. Because all tags use multiple inheritance to inherit not only from class `GRTag` but also from class `GRARNotationElement`, all tags can be contained in one of the composite classes (either `GRARCompositeNotationElement` or `GRCompositeNotationElement`) or their derivatives (like, for example, `GRStaff`, or `GRSystem`, or `GRPage`).

- **Class GRPositionTag** is the base class for representing range-tags (as explained in the previous paragraph). The issue of line breaking is crucial when dealing with range-tags within the GR: if both the begin- and end-event of a range-tag are located on one line of music, the graphical object (like, for example, a slur) can be created directly. If the begin- and end-event of a range-tag are located on different lines, then the graphical object must be split into several graphical objects. This case can be seen in Figure 4.5, where a slur begins in the last slice of the second voice in the second line and ends in the following line: here, the slur, which is represented by only one object in the Abstract Representation, is broken into two graphical objects. A mechanism within class `GRPositionTag` has been implemented to deal with these cases.

- **Class GRARNotationElement** is the base class for all graphical objects that have a direct counterpart in the Abstract Representation (which is shown by using "AR" as part of the name). Consider, for example, class `GRClef`, which inherits indirectly from class `GRTag` and from class `GRARNotationElement`. This reflects, that the graphical "clef" object has a direct counterpart in the AR (which obviously is an instance of the class `ARClef`). An example for a class that does *not* inherit from class `GRARNotationElement` is class `GRStaff`: there is no class called `ARStaff` in the AR, because a staff is a purely graphical entity, which is not directly reflected in the underlying abstract representation.[8]

The complete set of classes of the GR is too huge to be discussed in detail here. It should be clear that *any* visible element of a score has a matching class in the GR.

[8]Note that there is a `\staff`-tag in GUIDO, which can be used to define the staff on which a voice is being displayed. Nevertheless, there is no one-to-one correspondence of the graphical staff and the `\staff`-tag.

4.4 Converting the AR into the GR

Before the AR is converted into the GR, it has been prepared by a multitude of algorithms (see section 4.2), which ensure that the contained abstract music representation can be easily converted into a conventional score. The actual conversion of the AR into the GR is a process that is performed by so called *Manager*-classes: class `GRStaffManager` is responsible for creating a set of system slices, which where introduced in Figure 4.5. A single system slice contains a number of staves, on which the notation elements (like notes, rests, and musical markup) are placed. The staves of one system slice are subsequently filled by `GRVoiceManager`-classes (one instance of class `GRVoiceManager` for each voice of the AR), which iterate through `ARMusicalVoice`-classes and are also responsible for the creation of the corresponding graphical elements.
The overall process when converting the AR into GR is depicted in Figure 4.7: the AR is converted by `GRStaffManager` and its helper classes `GRVoiceManager`. There is one instance of `GRVoiceManager` for each voice in the AR. The graphical elements for each voice are created and distributed into the instances of `GRStaff`, which are contained in instances of `GRSystemSlice`. Class `GRStaffManager` creates a set of system slices which are finally converted into the Graphical Representation by another set of music notation algorithms, which are mainly responsible for spacing, line-breaking and page-filling.

4.4.1 Manager-Classes

The two manager-classes, which play an important role when converting the AR into the GR, will be presented in more detail in the following.

Class GRStaffManager

An instance of class `GRStaffManager` is used by class `GRMusic` to read the data from the AR and to create a set of `GRSystemSlice`-classes. `GRStaffManager` uses instances of class `GRVoiceManager` to read the data from the individual voices; all voices are traversed in parallel, so that notational elements, which need to be synchronized horizontally, are created in parallel. `GRStaffManager` is responsible for the following tasks:

- Creating instances of `GRVoiceManager`-classes.
- Traversing the voices of the AR in parallel (using the `GRVoiceManager`-classes).
- Creation of springs and rods for the spring-rod-model, which will later be used for spacing and line breaking; details of this model are described in section 4.5.

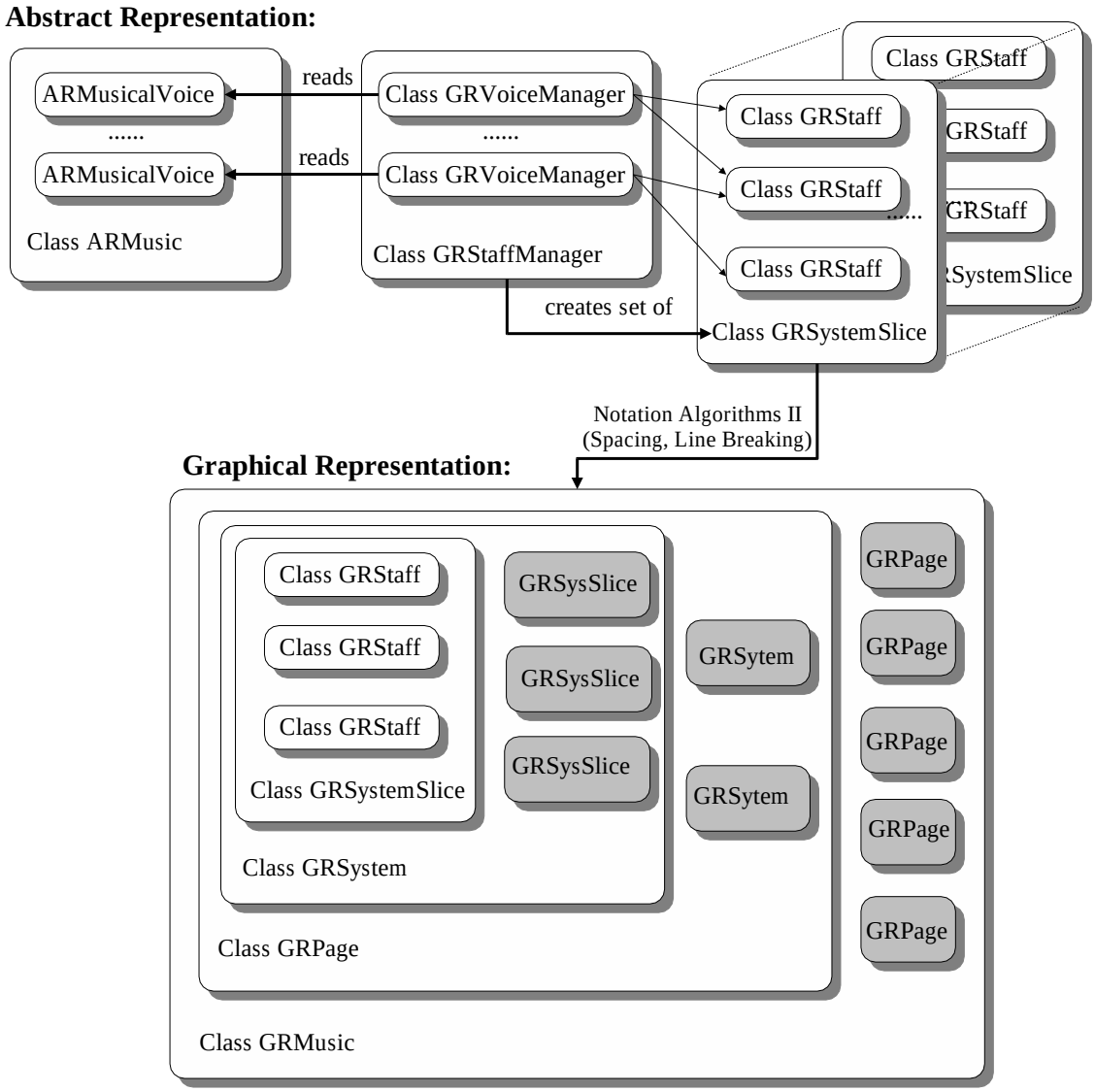


Figure 4.7: Converting the AR into the GR

- Assigning a spring to each notational element, which is created by the GRVoiceManager-classes; this task includes determining, which graphical elements are connected to the same spring. This is required to horizontally synchronize elements at equal time positions (like for example clefs or note heads).

Class GRVoiceManager

As indicated above, class GRStaffManager creates a set of instances of class GRVoiceManager. Each instance of GRVoiceManager iterates through one class ARMusicalVoice. For each event or tag that is found in the voice, an equivalent graphical element is created by class GRVoiceManager. The graphical elements are sent to class GRStaffManager, which is responsible for placing the elements on the staff. Because every event or tag is contained in a voice – there is no external or global header in GUIDO – all tags that have global relevance, like for example the \pageFormat-tag, is directly sent to class GRStaffManager, which is then responsible for processing the information.

4.4.2 Other helper classes

Some other classes, which have no direct graphical counterpart, are used during the conversions from the AR into the GR. Class GRSpaceForceFunction, class GRSpring, class GRRod, and class GRPossibleBreakPosition are mainly used for spacing and line-breaking. The details of this will be explained in sections 4.5.1 and 4.5.2. An instance of class GRPossibleBreakState is created every time a \pbreak-tag is encountered in a voice (see section 4.2). This object saves the complete state of all staves at the current time position; if the position is later used as a line-break location, the saved information is used to create the graphical elements at the beginnings of the staves in the next line. The required information is mainly the current clef, the current key signature and the current time signature. It is also essential to remember, which tagged ranges are currently active. In the case of a slur or a tie, the correct endings and beginnings have to be created.

Another class, which saves state information is class GRStaffState. It stores the current state of an instance of class GRStaff. Because different voices can be written on the same staff, it is essential that the current state of a staff is always accessible. The information of class GRStaffState is for example used, to determine the vertical position of notes by comparing the pitch and register information with the current clef. Class GRStaffState keeps also track of the current position within a measure (which depends on the current meter), and the current key signature.

4.5 Spacing, Line Breaking, and Page Filling

Even though music is perceived as a continuous stream of musical events, a traditional score is not printed on a continuous roll of paper, but is rather printed like a book. A score is usually printed on several pages, which contain multiple lines (which are sometimes called systems in the context of music notation), which in turn are made up from one or more staves. The decision where to introduce line breaks in music scores is analogous to line breaking in traditional text processing. It might seem that the problem of optimum line breaking in text and optimum line breaking in music is similar; unfortunately, the issues involved in music typesetting are more complex than in plain text formatting, as will be shown in the following section. Therefore, many of the elegant principles used in text setting can not directly be used when dealing with musical scores.
Three essential tasks must be performed when setting music:

1. Deciding where to break the lines of a score. This is called *line breaking*.
2. Distributing the space between graphical elements of a single line of music. This is called *spacing*.
3. Connected to line breaking is the task of distributing the lines of a score in such a way that all printed pages are full. A traditional score usually ends at the end of the last page. The problem of finding line breaks so that all pages are full is called *page filling*.

While the first of these tasks has a somewhat strong connection to text formatting, the second task is far more complicated when dealing with music and the third task is usually not needed when setting text. In standard text formatting, a single line of text is stretched or shrunk so that its width fits the desired paragraph width. This is accomplished by stretching or shrinking the white-space between the words in a uniform way.[9] In music typesetting, the space between notational events (notes, rests, markup etc.) is primarily determined by the duration of the respective event: a half note is followed by more space than a sixteenth note, for example. Additionally, simultaneous events in different voices must be printed vertically aligned, something that usually only occurs when formatting tables in standard text formatting. When setting music, special attention has to be given to detecting and preventing collisions of different notational elements: in some music, there is more than one voice being displayed on a single staff. In this case, collision detection and prevention can be very complicated.
In the following section, spacing of a single line of music and line breaking will be explained in detail. Furthermore, it is shown how optimal page fill can be realized. Wherever appropriate, a comparison to standard text formatting will be made.

[9] In TEX, white-space after punctuation marks is treated differently than white-space between words. In general, the parameters required for stretching and shrinking can be calculated fairly easily.

4.5.1 Spacing

When typesetting music, the issue of distributing leftover space in between the graphical elements of a line is fundamental for obtaining good and esthetically pleasing results. In the following, approaches for human and automatic spacing are described. An improved algorithm for spacing, which was developed as part of this thesis, is presented.
In order to estimate, how "good" a spacing algorithm performs its task, it is essential to define, what constitutes good spacing. Helene Wanske defines the requirements for spacing as follows [Wan88]:

1. Spacing should follow the rhythmical structure of the music.
2. The rhythmical interaction of multiple voices must be clearly recognizable in the score.
3. The line up of notational symbols shall be optically balanced.[10]
4. The disposition of notational symbols depends on printing requirements: a certain amount of music has to be put on one page and respectively on one line of music; the lines of music shall be justified.
5. The overall impression of a score page shall be smooth; there shall be no "black clusters" nor "white wholes". Any "surprise-effects" shall be prevented. Nevertheless, the disposition of notational elements on one page shall follow economic considerations.

These rules are ranked according to their importance for music notation. They are definitely "fuzzy", and it is generally not clearly decidable, whether a line of music is "good" or "better" spaced than another one. During the work on this thesis, the primary goal was the automatic calculation of spacing as it is found in "real" scores, which have been produced by human engravers. In most cases, different editions of one piece of music can be found. Often, these editions differ in the used musical font and almost certainly in differences in spacing. Therefore, a secondary goal was the ability to *adjust* the spacing algorithm to suit the users need. One thing that needs to be kept in mind is that different readers may require different scores: it is probably much easier for a professional musician to read a tightly spaced score, whereas someone learning to read music notation clearly is better off with loosely spaced music.
In conventional music notation, the space after a notehead (or a rest) depends primarily on the duration of the event. Traditionally, these relationships are specified with respect to the note durations of $\frac{1}{16} : \frac{1}{8} : \frac{1}{4} : \frac{1}{2}$. Different engravers have used

[10] Wanske gives several examples where a spacing, which is calculated strictly according to the given rhythmical structure, may lead to lines that *appear* to be unequally spaced. A line of music is "optically balanced", if equal note durations are *perceived* to be equally spaced by a human reader.

slightly different relationships between note duration and space. Helene Wanske, who gives a thorough[11] overview on (human) spacing, specifies a relationship of $1 : 1.4 : 1.8 : 2.2$ meaning that an eighth note is followed by 1.4 times more space than a sixteenth note [Wan88]. When calculating the values for a fixed distance of 5mm for the space after a sixteenth note, the resulting music notation can be seen in Figure 4.8. These values give only a rough guideline for the actual engraving of music: in most cases, additional symbols (like for example accidentals) or the need of vertically aligning different voices leads to different amounts of white space after noteheads or rests. Additionally, when spacing different voices with unorthodox note durations (for example a triplet versus a quintuplet) the individual tuplet-groups should be spaced evenly (this is a consequence of Wanske's third spacing rule).

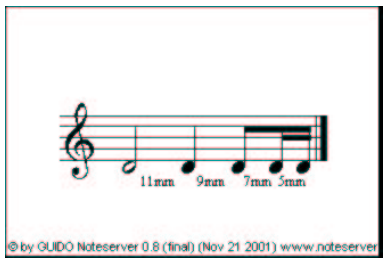


Figure 4.8: Spacing with Wanske relationship

Automatically spacing a line of music has been discussed quite often in the past [Byr84, Gou87, BH91, HB95, Gie01]. The most important contribution to the problem was given by Gourlay in 1987. His spacing model, which in turn was inspired by Donald Knuth's work on TEX [Knu98], seemingly builds the basis for a large number of the currently implemented notation systems.[12] His model can be described as a spring-rod-model:[13] a line of music is interpreted as a collection of springs with some spring constants on which a force is exerted to stretch or shrink the line to a desired length. Rods are introduced to ensure that noteheads and musical markup (like for example accidentals) do not collide, especially when spacing is tight. The rods are used to pre-stretch the springs to have a guaranteed minimal extent. In contrast to formatting text, the issues found in formatting music are more complex. Because simultaneous voices must be vertically aligned, a spacing algorithm must be capable of handling overlapping note durations in different voices while still maintaining a spacing that closely follows individual note durations.

[11] The section on spacing covers 46 pages! (pages 107-153)

[12] Obviously, it is not really possible to detect the implemented spacing algorithm for software that is distributed without the source code.

[13] The original article by Gourlay does not use the word "springs" but speaks of boxes and glue, which in turn was inspired by TEX. We find that the term "spring" more closely corresponds to the physical model being used.

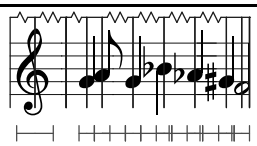
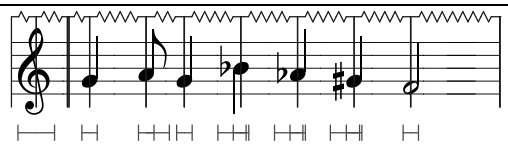
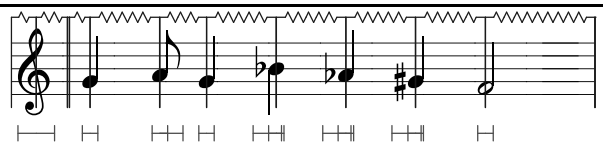
Force 0	
Force 350	
Force 690	
Force 850	

Table 4.1: Demonstrating the effect of applying different forces

The basic idea of Gourlays' algorithm can be summarized as follows: a single line of a score is interpreted as a list of onset times, which is sorted by time position; an onset time occurs whenever a note or rest begins in one of the voices. In order to determine the correct amount of space to put in between the onset times, springs are inserted between successive onset times. In order to stretch (or shrink) the line to its desired length, a force is exerted on the whole collection of springs. Each spring is then stretched (or shrunk) depending on the external force and the individual spring constant, which mainly depends on the spring duration[14] and on the duration of notes (or rests) being present at the time position of the spring. Hooks' Law describes, how a force F, an extent x and a spring constant c are related: $F = c \cdot x$. To avoid collisions of noteheads, accidentals, and other musical markup, rods are introduced, which determine the minimum stretch for one or more springs. The spring-rod-model generally agrees very well with Wanske's spacing-rules, which were defined at the beginning of this section. Almost all essential requirements for spacing are automatically fulfilled, when the model is used.
Table 4.1 shows a short musical example together with the associated springs and rods. The springs are shown above the line of music, the rods are shown below. It is demonstrated, how different forces are applied to the same line of music. When the applied force is zero, the rods prevent the notated symbols to collide horizontally. As the force increases, more and more springs are stretched further than the extent

[14]The "spring duration" is the temporal distance of the two onset times in between which the spring is placed.

of the rods. At Force 690, all springs are stretched further than the pre-stretch induced by the rods.

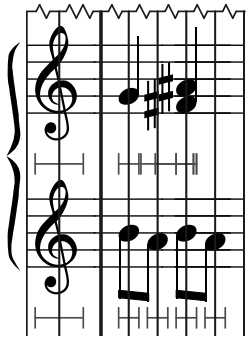


Figure 4.9: Overlap of notational elements

The spring-rod-model also works very well for multi-voiced music: because rods are created separately for each voice (or staff), notational elements of different voices/staves may overlap horizontally, as long as extent of the *combined* springs is at least as wide as the rod extent. Figure 4.9 shows, how two voices are spaced using the spring-rod-model. The accidentals of the chord in the first voice have the same horizontal position as the second eighth note of the second voice. This exactly matches common rules for music notation.

It will be shown later in this chapter, how the individual spring constants for a given line of music are calculated. In order to better understand the workings of the underlying physical model, lets assume that the spring constants have already been calculated. Then, a line of music can be spaced by exerting a force on the (sequential) collection of springs. Because in musical scores, all lines are usually justified, an algorithm is required, which calculates the force required to stretch a line of music to a given width.

To calculate how springs in series react to a certain force, an overall spring constant is determined using this formula:

$$c = \frac{1}{\sum_{i=1}^{n} \frac{1}{c_i}} \tag{4.1}$$

The overall spring constant can then be used in Hooks' Law as before: $x = \frac{F}{c}$. Figure 4.10 shows an example of three springs with three different spring constants. The applied force is $f = 15$, stretching the series of springs to an overall extent of 10cm. When the three springs are replaced by one spring and the same force of $f = 15$ is applied, then the new spring constant must be $c = \frac{15}{10} = 1.5$, which is exactly what equation (4.1) evaluates to for the 3 individual springs.

When dealing with the springs in a line of music, an additional fact has to be taken into account: as was shown before, springs can be pre-stretched by rods. In this case, a spring has a pre-stored extent and a pre-stored force. For example, consider a spring s with spring constant c that is pre-stretched to length x_p. Then the resulting pre-stretching force F_p is computed as

$$F_p = c \cdot x_p$$

If an external force F_e is applied to the spring s, the new extent can be calculated as:

$$x(F_e) = \begin{cases} x_p & \text{if } F_e \leq F_p \\ \frac{F_e}{c} & \text{if } F_e > F_p \end{cases}$$

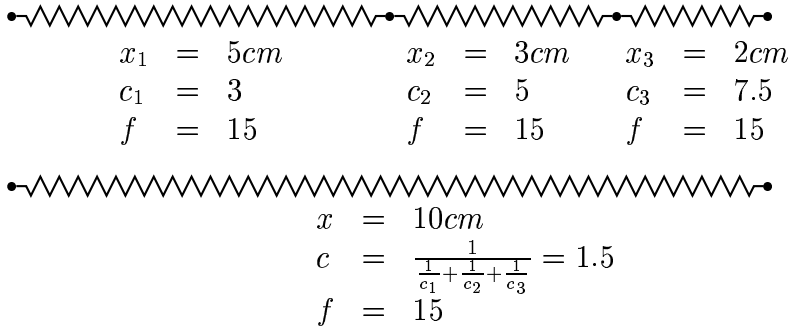


Figure 4.10: Connected springs

This means that the spring will only be stretched further by an external force F_e, if F_e is greater than the pre-stretching force F_p. Otherwise, the extent will just be the pre-stretched extent x_p.
Therefore, when dealing with a series of springs from a line of music, the overall spring-constant depends on the applied force and the pre-stretched springs: only those springs are further stretched, whose saved force is smaller than the applied force, and therefore, the overall spring constant must be computed using only these springs. The other springs just add their pre-stretched extent.

The Space-Force-Function

To efficiently compute the force required to stretch a line of music a so called space-force-function was defined and implemented during work on this thesis. This function calculates the force required to stretch a line of music to a given extent. Evidently, the pre-stretching of springs is taken into account. The space-force function being introduced here, is extensively used in line breaking and page filling, which will be described at the end of this chapter.
Given a set of springs $S = (s_1, \ldots, s_n)$ with respective spring constants c_i and pre-stretch x_i for $1 \leq i \leq n$, the space-force-function $\mathrm{sff} : \text{extent} \rightarrow \text{force}$ calculates the required force to stretch the springs to a given extent:

$$\mathrm{sff}(x) = \frac{x - x_{\min}}{c} = f \quad \text{with } c = \frac{1}{\sum\limits_{\substack{1 \leq i \leq n \\ c_i \cdot x_i \leq f}} \frac{1}{c_i}} \quad \text{and } x_{\min} = \sum_{\substack{1 \leq i \leq n \\ c_i \cdot x_i > f}} x_i \tag{4.2}$$

It is obvious that sff in equation (4.2) cannot be computed directly, because the exact values for c and $x_{\min}$, which are required to calculate sff can only be determined when the overall result for sff has already been computed. In order to compute the function, a pre-processing step is performed, which sorts the set of springs by their respective pre-stretching forces. Additionally, the sum of all pre-stretched spring extents is computed. Then, the calculation of sff simply requires a traversal of the

ordered set: Let $S = (s_1, \ldots, s_n)$ be a set of springs with their respective spring constants c_i and pre-stretching force f_i and pre-stretching extent $x_i = \frac{f_i}{c_i}$ (for $1 \leq i \leq n$). Let S be ordered, so that for all $i \leq j$ we have $f_i \leq f_j$. Algorithm 1 computes the required force for a given extent.

Algorithm 1 sff: Determining the required force for a given extent

Require: $x > 0$ {x is the extent}
Require: $\exists c = (c_1, \ldots, c_n)$ spring constants
Require: $\exists x = (x_1, \ldots, x_n)$ spring extents
Require: $\forall\, i, j, 1 \leq i \leq j \leq n : x_i \cdot c_i \leq x_j \cdot c_j$
  {Pre-calculate the minimum extent:}
  {This is the sum of all pre-stretched springs}
  $x_{\min} := \sum_{i=1}^{n} x_i$
  if $x \leq x_{\min}$ **then** {The extent is smaller than the pre-stretched springs;}
    return 0; {Therefore, the required force is 0}
  end if
  {Initialize the spring-constant}
  $c := c_1$;
  for $i := 1$ to n **do** {Each spring is taken from the ordered set}
    $x_{\min} := x_{\min} - x_i$; {The pre-stretched extent is subtracted}
    $f := \frac{x - x_{\min}}{c}$; {The force is calculated using the current spring-constant}
    if $i = n$ or $f \leq f_{i+1}$ **then** {We are at the very last spring or the force of}
      return f; {the next spring is bigger than the required force}
    end if
    {Adjust the spring constant for the next cycle}
    $c := \frac{1}{\frac{1}{c} + \frac{1}{c_{i+1}}}$;
  end for

The space-force-function is a strictly monotonic rising, piece-wise linear function. Figure 4.11 shows a sample space-force-function for a short line of music, which is stretched using three different forces. In the graph of the sff, it can be clearly seen how the individual notes have differing pre-stretching forces, depending on the spring-constant. The smaller valued notes have bigger spring constants, therefore, pre-stretching these notes requires more force than pre-stretching the notes with longer durations. When a force of 400 is applied (first line of music in Figure 4.11), the 16th-notes are just stretched marginally, while the 32nd-notes are not stretched at all. The 32nd-notes do not overlap only because of the existing rods.
The inverse function of the space-force-function, which determines an extent for a given force, can be calculated similar to the above procedure. In this case, the (ordered) set of springs is traversed until the line-segment of the sff is found, that matches the given force. Then, the concrete value is calculated using the spring constant for this section.

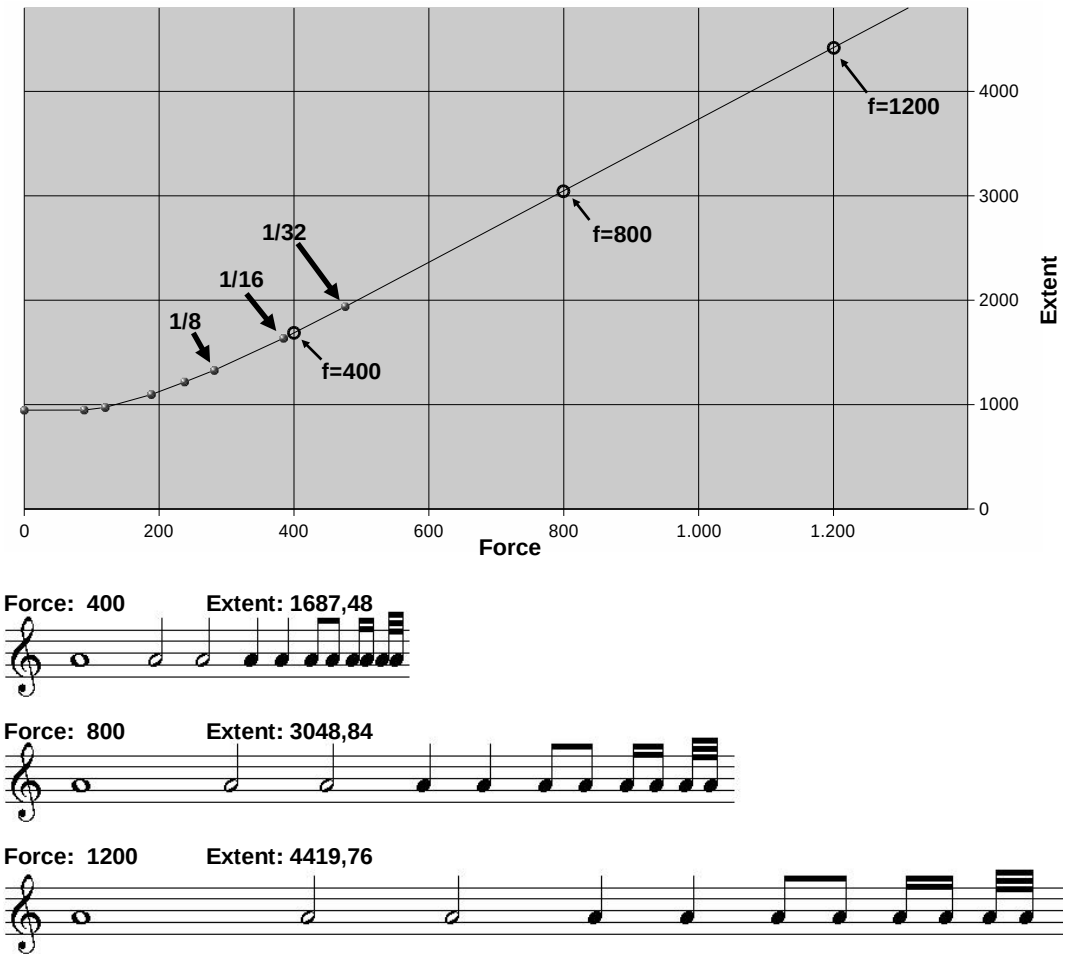


Figure 4.11: The Space-Force-Function

Gourlays' Algorithm for Spacing

This section gives a brief overview of Gourlays' algorithm for spacing a line of music [Gou87] and briefly discusses additional publications and notation systems based on it. Obviously, some of the details of Gourlays' algorithm are omitted to focus on the relevant issues being improved later.

Calculation of spring constants: The calculation of spring constants in Gourlays' algorithm, which will be modified by our improved algorithm, needs to be examined more closely. Basically, the space placed after a note (or rest) solely depends on the duration of the event: there is generally less space placed after a sixteenth note than after a whole note. A suitable way to calculate the actual amount of space for a given duration d is given by the formula

$$\text{space}(d) = \psi(d) \cdot \text{space}(d_{\min}) \qquad \text{where} \qquad \psi(d) = 1 + a \cdot \log_2\left(\frac{d}{d_{\min}}\right) \tag{4.3}$$

where a is some constant (usually in the range of 0.4 and 0.6), and $d_{\min}$ is some smallest duration for which the required space is predefined. In current notation software (like, for instance in Finale, LilyPond, etc.) quite a number of minor variations of this scheme are actually implemented (see for example [Gie01]), but the general principle of using a logarithmic function for calculating the required space is agreed upon widely. Tweaking the parameters for a and $d_{\min}$ results in slightly different spacing and it is usually a matter of personal (or publishers) taste, which values to use.

The formula for $\psi(d)$ is also used to determine the spring constants for a given line of a score: Let s be a spring between two successive onset times o_1 and o_2. The resulting spring duration d_s is simply $d_s = \text{timeposition}(o_2) - \text{timeposition}(o_1)$. Let d_i be the shortest duration of all notes (or rests) beginning or continuing at $\text{timeposition}(o_1)$. Then the spring constant c is calculated as

$$c = \frac{d_i}{d_s} \cdot \frac{1}{\psi(d_i) \cdot \text{space}(d_{\min})} \tag{4.4}$$

Gourlay introduced this formula in order to get optimal spacing even when rhythmically complex structures (like tuplets versus triplets) are present. In the case of monophonic pieces, the formula simplifies to

$$c_{\text{simple}} = 1 \cdot \frac{1}{\psi(d_s) \cdot \text{space}(d_{\min})}$$

because $d_i = d_s$. Using this simple spring constant and putting it into Hooks' Law with a force $F = 1$ and a duration of d_s equal to $d_{\min}$ stretches the spring to the extent

$$x = \frac{F}{c} = 1 \cdot \psi(d_{\min}) \cdot \text{space}(d_{\min}) = \text{space}(d_{\min})$$

which is exactly the expected result.

As indicated above, a number of variations of Gourlays' algorithm have been proposed. The newest publication on spacing a line of music is by Haken and Blostein [HB95]. They were the first to actually use the terms "springs" and "rods". Even though their terminology is different, the general ideas of Gourlay are present. The only significant difference lies in the handling of rods, which are called "blocking widths" by Gourlay. Haken/Blostein introduced an elegant way to pre-stretch the springs using a two-step process: first, only those rods are considered that only stretch one spring. Then the remaining rods (which all span more than one spring) are checked. In this phase, the required force to stretch a chain of springs to the desired rod length is calculated. The rod requiring the *maximum force* is applied first; other rods stretching the same springs can be discarded. This procedure is more efficient than Gourlays' original algorithm, which just iterates through all rods without pre-sorting.

GNU LilyPond, a musical typesetting system based on TEX [NN01], is also using Gourlays' algorithm for spacing a line of music. One interesting detail of LilyPond's

spacing is the fact that the value for $d_{\min}$ in equation (4.3) is determined separately for each measure. This sometimes leads to an uneven spacing of measures, which might be musically misleading as shown in Figure 4.12, where the general rhythm of measures one and two is quite similar, but the spacing is very uneven.[15]

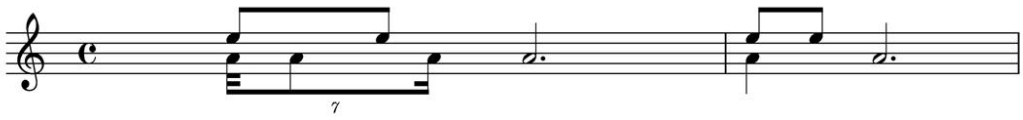


Figure 4.12: LilyPond spacing problem

A recently published dissertation on the automatic generation of scores [Gie01] uses the following formula

$$c = \frac{1}{\max\{\frac{d_s}{d} \cdot \psi(d)\} \cdot \text{space}(d_{\min})} \tag{4.5}$$

Where d is the duration of any note (or rest) beginning or continuing at the spring. When using the standard formula for ψ, the maximum for the term in equation (4.5) *always* occurs for the shortest duration (the term d_i in equation (4.4)). Therefore, equation (4.5) is similar to Gourlays' original proposal.

The spacing algorithms used by commercial notation software are usually not disclosed: therefore, a small number of spacing tests were performed with the two most popular systems Finale and Sibelius.[16] In the case of Finale, it seems like some variation of Gourlays' algorithm is used: the obtained spacing for rhythmically complex pieces is generally very good. Sibelius, on the other hand, created awkward spacing even for some rather simple examples. It seems like the used algorithm does not allow noteheads to overlap horizontally, even if they are in completely independent voices. Figure 4.13 shows, how Sibelius fails to space a rhythmically complex measure in an acceptable way: the spacing of the individual voices is completely irregular.

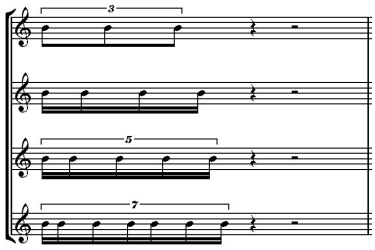


Figure 4.13: Spacing in Sibelius

Spacing Problems in Gourlays' Algorithm

Gourlays' algorithm for spacing a line of music calculates good spacing on a large body of musical scores. Because of its wide spread use, it was therefore somewhat

[15] Obviously, it can be argued, whether a spacing like this is desirable or not.

[16] For our tests we used Finale 2001b and Sibelius 1.4

surprising to detect that Gourlays' algorithm (and therefore all of its direct descendants) fails to produce good spacing in certain situations. The reason for these deficiencies stems from the fact that sequential individual notes with equal duration might be spaced unevenly because of simultaneous notes in other voices. As stated above, the coupling of note duration and spacing is very strong, therefore notes with equal duration should be spaced equally whenever possible.[17] Because these cases do occur in real scores (and also in computer generated music), a solution to the problem was searched for and found. This solution led to an improved algorithm for optimally spacing a line of music that automatically detects and corrects the above mentioned spacing errors. First, groups of notes with equal duration are determined in the score. Then, for each such group it is checked, whether the original Gourlay algorithm creates spacing errors. If this is the case, an alternate spacing is calculated and applied, if certain tolerance conditions are met.

Figure 4.14 shows, how the improved spacing algorithm (b) compares to Gourlays' original algorithm (a): The spacing of the triplet in the first voice is exactly equal when the improved algorithm is used. This effect is highly desirable, because the graphical appearance of the triplet now directly conveys that each individual triplet note has equal duration. When using the original spacing algorithm (a), the space between the first and the second note of the triplet is significantly greater than between the second and the third note, even though their duration is equal. We call this error a "neighborhood spacing error".

(a) (b)

Figure 4.14: Gourlays' original algorithm (a) compared to the improved spacing algorithm (b)

The reason for the unequal spacing can be seen rather easily: Figure 4.15 shows a rhythmically simpler example together with the created springs and their respective calculated spring constants. In Figure 4.15 (a) the first triplet note spans two springs (s_1 and s_2) with spring constants $c_1 = \frac{\frac{1}{16}}{\frac{1}{16}} \cdot \frac{1}{\psi(\frac{1}{16})} = \frac{1}{\psi(\frac{1}{16})}$ and $c_2 = \frac{\frac{1}{12}}{\frac{1}{48}} \cdot \frac{1}{\psi(\frac{1}{12})} = \frac{3}{\psi(\frac{1}{12})}$. When combining s_1 and s_2 into a combined spring $s_{1,2}$ the spring constant $c_{1,2}$

[17] This can be directly derived from Wanske's third spacing rule of section 4.5.1 on page 86. When dealing with scores that contain rhythmic complexity, this rule may have a rather low priority. Nevertheless, it should be applied wherever possible.

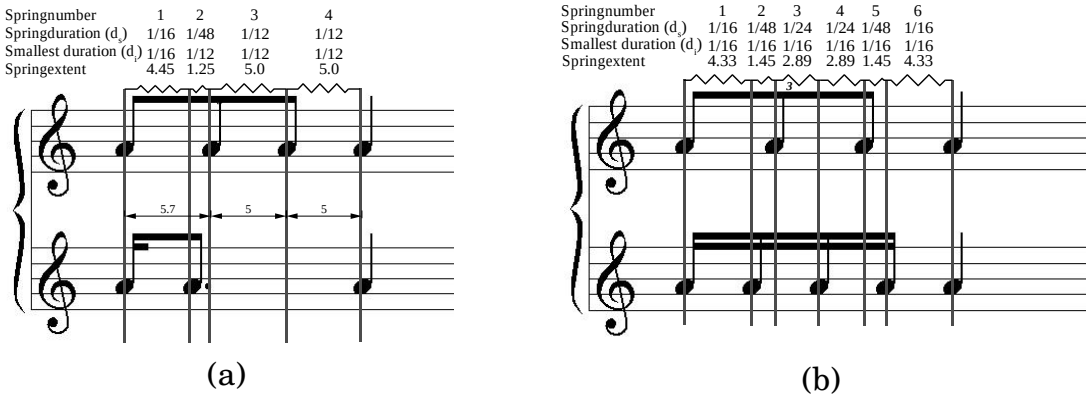


Figure 4.15: Spacing of Gourlays' algorithm including springs

is calculated using equation (4.1) as

$$c_{1,2} = \frac{1}{\frac{1}{c_1} + \frac{1}{c_2}} = \frac{1}{\psi(\frac{1}{16}) + \frac{1}{3}\psi(\frac{1}{12})}$$

It is obvious, that the spring constant $c_{1,2}$ is not equal to $c_3 = c_4 = \frac{1}{\psi(\frac{1}{12})}$. Therefore, when a force F stretches the line, the springs s_1 and s_2 are stretched to the extent $x_{1,2} = \frac{F}{c} = F \cdot (\psi(\frac{1}{16}) + \frac{1}{3}\psi(\frac{1}{12}))$ and the springs s_3 and s_4 are both stretched equally to $x_3 = x_4 = F \cdot \psi(\frac{1}{12})$. Clearly, $x_{1,2}$ and x_3 (and x_4) are not equal.
To understand, why this error does not occur in the standard 3 against 4 case (see Figure 4.15 (b)), one must understand how the fraction $\frac{d_i}{d_s}$ in equation (4.4) is supposed to work: consider Figure 4.15 (b), which differs from (a) by replacing the second voice with four sixteenth notes. Now, the value of d_i for each of the springs is $\frac{1}{16}$, so that each spring is stretched as a partial of a sixteenth-note spring. To clarify this, consider springs s_2 and s_3 of Figure 4.15 (b). The respective spring-constants are $c_2 = \frac{\frac{1}{16}}{\frac{1}{48}} \cdot \frac{1}{\psi(\frac{1}{16})} = \frac{3}{1} \cdot \frac{1}{\psi(\frac{1}{16})}$ and $c_3 = \frac{\frac{1}{16}}{\frac{1}{24}} \cdot \frac{1}{\psi(\frac{1}{16})} = \frac{3}{2} \cdot \frac{1}{\psi(\frac{1}{16})}$. When combining springs s_2 and s_3 the spring constant is

$$c_{2,3} = \frac{1}{\frac{1}{3}\psi(\frac{1}{16}) + \frac{2}{3}\psi(\frac{1}{16})} = \frac{1}{\psi(\frac{1}{16})}$$

The result is, that the springs s_2 and s_3 are stretched so that their sequential combination behaves just as a single spring for one sixteenth note. The same is true for springs 4 and 5. Subsequently, the springs s_1, $s_{2,3}$, $s_{4,5}$, and s_6 all have a spring constant of $\frac{1}{\psi(\frac{1}{16})}$ so that the sixteenth notes of the second voice of Figure 4.15 (b) are spaced evenly. Regarding the first voice of Figure 4.15 (b), the respective spring constants $c_{1,2}$, $c_{3,4}$, and $c_{5,6}$ are all equal to $\frac{3}{4} \cdot \frac{1}{\psi(\frac{1}{16})}$, so each note gets the same amount of space.
One other spacing problem of Gourlays' algorithm concerns lines of music that are spaced very loosely. Consider Figure 4.16, where a single line of music has been

stretched rather heavily. In this case, Gourlays' algorithm (a) results in unequal spacing of the eighth-notes in the second voice: because of the 32nd notes in the first voice, the first eighth-note of the second voice is followed by much more space than the following ones, which are all spaced equally. The improved algorithm (b) distributes the space evenly.

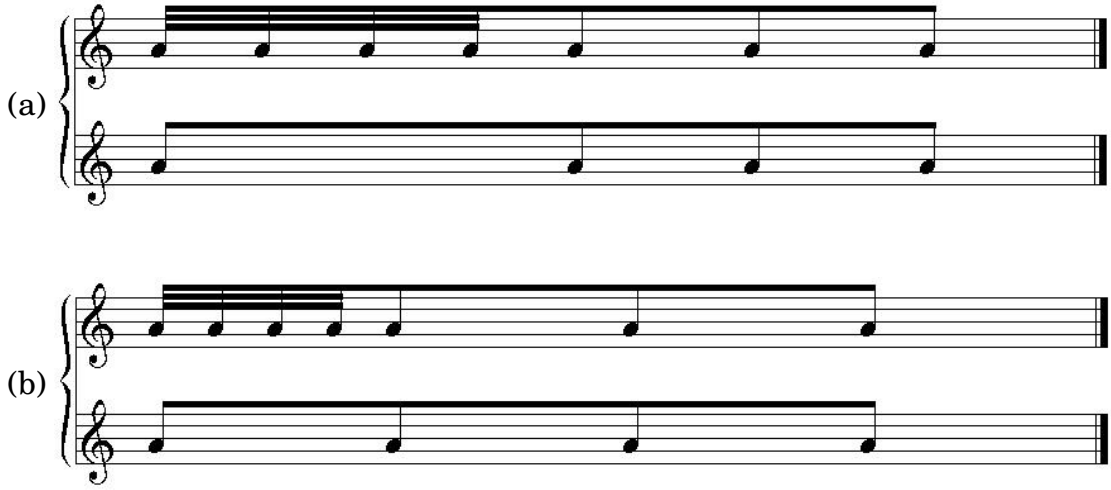


Figure 4.16: Loose spacing: Gourlay algorithm (a) and improved algorithm (b)

An Improved Spacing Algorithm

It can be deduced from the last section that, for solving the neighborhood spacing problem, an algorithm must automatically detect those regions of springs, where spacing errors might occur and then set an appropriate value for d_i only for the relevant regions. This is exactly what our proposed improved spacing algorithm does, which is now described in detail:

1. The original algorithm of Gourlay for the given line of music is carried out. For each spring, the value $\frac{1}{d_i}$ from equation (4.4) is saved, instead of directly calculating and saving the spring constants.

2. For each voice the location of neighborhoods of notes with the same duration are determined.

3. Each of the calculated neighborhoods from step (2) is checked for neighborhood spacing errors. This is done by averaging over the values d_i for each spring covering a note in the neighborhood. If the average value differs for any note of the neighborhood, a spacing error is detected.

4. For all error regions of step (3), it is checked to see whether any other neighborhood reaches into it; if this is the case, then the boundary of the error-region

is extended to also cover the new neighborhood. This step is necessary so that the fixing a spacing error in one of the error-regions does not introduce a new spacing error in another voice.

5. For the finally determined error-regions of step (4), three spring constants are calculated: the spring constant for the original Gourlay algorithm, the spring constant for using an average of the d_i, and a spring constant which uses the minimum d_i for the whole region.

6. The difference of the original Gourlay constant and the average constant is determined. If it is within a tolerance band (less than 0.05), then the average value for d_i is taken for the region. Otherwise, the difference of the original constant and the minimum-d_i-constant is calculated. If this difference is within another tolerance band (less than 0.17), then the minimum duration of the region is chosen as d_i. Otherwise the original Gourlay value for d_i is chosen for the region.

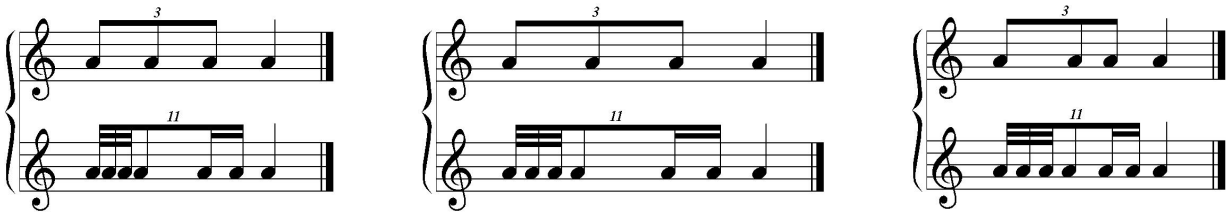
7. Before a line is finally spaced, an additional step is performed to avoid the wrong spacing of loosely spaced lines. This is done by determining, whether any note with the smallest note duration of the whole line is followed by more than two noteheads of white space using the calculated spring constant. If this is the case, the smallest note duration for the whole line is taken as a value for d_i and the spring constants for the line are recalculated accordingly.[18]

The values for the tolerance bands for step 6 of the algorithm have been chosen through various experiments. Figure 4.17 shows an example for taking the average (a), the minimum (b) and the original Gourlay value (c). Although the neighborhood spacing error in the first voice is removed in cases (a) and (b), choosing the average value results in too little space for the short notes in the beginning of the second voice. Choosing the minimum duration (b) results in a spacing, which is too wide. In this example, the original Gourlay algorithm (c) gives the best result, although the neighborhood spacing error in the first voice is clearly visible. The tolerance bands are indications, when to tolerate a neighborhood spacing error so that the overall spacing remains acceptable.
To see, how the algorithm solves the neighborhood spacing problem of the last section (see Figure 4.15 (a)), we give a trace of the improved algorithm.

1. The original Gourlay algorithm is used to determine the values $\frac{1}{d_i}$ for each spring: $16, 12, 12, 12$. These values can be found in Figure 4.15 (a).

2. The first voice has a neighborhood list of $(1, 3, 4, 5)$ which means, that the notes covering springs 1 up to 3 (not including the last value) and 3 to 4 and 4 to 5 are successive notes having the same duration (in our example this is $\frac{1}{12}$). There is no neighborhood list for the second voice.

[18] This step was performed to obtain the result of Figure 4.16 (b).



(a) (b) (c)

Figure 4.17: Showing the effect of choosing different relative duration values

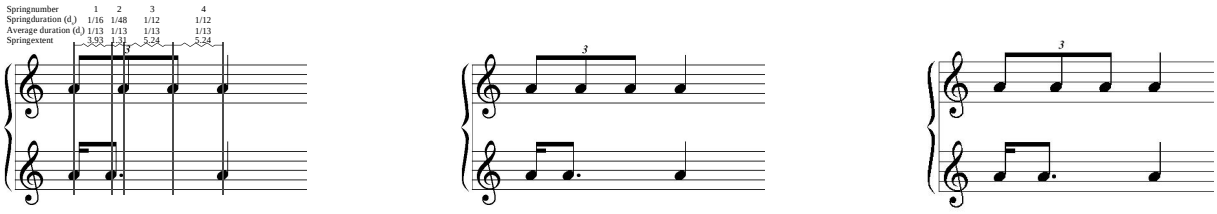
3. The neighborhood list $(1, 3, 4, 5)$ is checked for spacing errors: the average for the first note (going from springs 1 up to 3) is $\frac{16+12}{2} = 14$, the average for the second and third note is both 12. Therefore, a spacing error is detected.

4. Because there are no more neighborhoods, the error region reaches from spring 1 up until spring 5.

5. The three spring-constants are calculated: $s_{\text{gourlay}} = 1.81665$, $s_{\text{minimum}} = 1.60325$, $s_{\text{average}} = 1.81631$.

6. The difference $|s_{\text{gourlay}} - s_{\text{average}}| = 0.00034$ is within the tolerance band. Therefore, the average value for $\frac{1}{d_i} = \frac{16+12+12+12}{4} = 13$ is chosen. Spacing is done, as if the smallest duration is a 13th note.

The resulting spacing can be seen in Figure 4.18 (a) and (b), which also shows a comparison to the old Gourlay algorithm (c).



(a) Improved algorithm with springs

(b) Improved algorithm

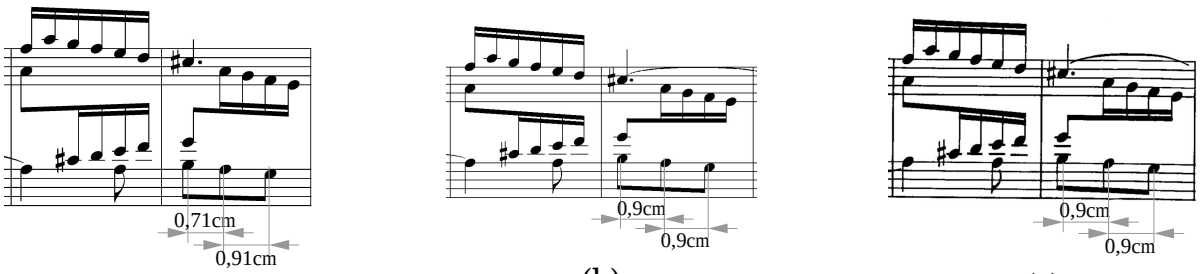
(c) Gourlay algorithm

Figure 4.18: Correction of neighborhood spacing errors by the improved algorithm

To show that the algorithm not only concerns somewhat constructed examples, Figure 4.19 shows a two-measure excerpt from a fugue by J.S.Bach (BWV 856) as it is spaced by Finale (a), the improved algorithm (b) and in the Henle Urtext edition (c), which has been spaced by an expert human engraver [vI70]. It is clearly visible, that (b) and (c) strongly emphasize on the equal spacing of the eighth-notes in the bass voice of the second measure, whereas the spacing algorithm of Finale[19] spaces

[19]Finale spaces this example similar to Gourlays' algorithm.

the first eighth note tighter than the following two. Of course it can be argued, whether the spacing of (b) and (c) is better than the spacing in (a); it is surely a matter of personal taste and also of musical semantics which spacing is preferred. The point made here is that any spacing algorithm should be adjustable to accommodate personal preferences.



(a) (b) (c)

Figure 4.19: Bach fugue BWV 856: Finale (a), improved algorithm (b), hand engraver (c)

4.5.2 Line breaking

Line breaking is the process of determining the optimum positions where to break the lines of music so that the resulting score looks esthetically pleasing, which depends largely on the following:

1. The individual lines of the score are neither spaced too tight nor too lose.
2. If the piece covers more than 1 page, the individual lines should be created so that the score covers all pages (including the last one) completely.
3. The locations where page turns occur are musically sensible: positions with less activity should be preferred over passages where a lot of notes are being played.

While the last of these points is far from being solved by any automatic system for music notation, the other two can be tackled. As Hegazy and Gourlay have pointed out in [HG87], the process of optimal line breaking in music is strongly related to line breaking in conventional text typesetting. The most relevant research in this area has been carried out by Donald Knuth [Knu98] and its results are part of the TEX typesetting system. In music typesetting the work of Hegazy and Gourlay is the first to cover the issue. Their ideas are derived from Knuth, but as will be shown in the following section, the problem is a little more complex in music typesetting than it is when setting text. Although the issue of page filling (second item from the above list) is related to optimal line breaking, there exists no publication on the

subject, even though the music notation system SCORE by Leland Smith [Smi97] contains an (unpublished) optimal page fill algorithm. While working on this thesis, an algorithm for optimal page fill was developed. Because it is an extension to the optimal line breaking algorithm proposed by Hegazy and Gourlay, we will first give an overview of Knuths' original algorithm for line breaking in text, then the Hegazy-Gourlay algorithm for optimal line breaking in music is being presented. The new optimal page fill algorithm will be presented in a separate section below.

Line Breaking in Text Paragraphs or Learning from TEX

Because the ideas of finding the optimum line breaks in text form the basis for deciding on the optimum line breaks in music, this section gives a short overview of the fundamental ideas applied there. First, a (simplified) mathematical description of the problem is given; then, the basic steps to solve the stated problem are shown. The actual algorithm given by Knuth is more complex, because it has been fine-tuned to cover a great variety of problems found in setting texts. For understanding the basic ideas, the following description should be sufficient.

A paragraph may be seen as a sequence of n words ($n > 0$), $\{w_1, \ldots, w_n\}$. Each word w_i has a (natural) length $l_i > 0$. Each word is separated from the next by white-space. To get justified output, the space between words can be stretched or compressed.[20] For simplicity reasons, we assume that all lines of the paragraph ought to have equal length given by D. The task is to find a sequence of indices $B = \{b_0, b_1, \ldots, b_k\}$ that determine which word ends on which line: w_{b_1} is the last word of the first line, w_{b_2} is the last word of the second line, and so on. b_0 is always equal to zero and b_k is always equal to n (this means, that the last word of the last line is the last word of the paragraph). It should be clear that for all i with $0 \leq i < k$ we have $b_i < b_{i+1}$ (B is a strictly monotone rising, ordered list) and that for all i with $1 \leq i \leq k$ we have $b_i \in \{1, \ldots, n\}$. The sequence B breaks the lines of the paragraph in an optimal way, if the amount of stretching or compression of the space between words is minimized.

Let s be the natural amount of space between words. Let $L_{i,j}$ be the sum of the lengths of the words from w_i up to w_j: $L_{i,j} = \sum_{k=i}^{j} l_k$. Putting the words w_i to w_j in one line with natural word spacing requires a length of $L_{i,j} + (j - i)s$. In order to evaluate how much stretching or compression of the natural space s is needed to set the words w_i until w_j in one line, a penalty function $P_{i,j}$ is defined as follows:

$$P_{i,j} = \begin{cases} |D - L_{i,j} - (j - i)s| & D \geq L_{i,j} \\ \infty & D < L_{i,j} \end{cases} \tag{4.6}$$

The penalty is zero, if the words can be set without stretching or compression of s. The penalty becomes ∞, if there is no space left between the words of the line.[21]

[20] Knuth's algorithm deals with stretching and compression of space differently. His algorithms also obviously deals with hyphenation.

[21] Knuths' penalty function is more complex; for reasons of simplicity, the above definition suffices.

The objective is to find a break sequence $B_{\min} = \{b_0, b_1, \ldots, b_k\}$ so that the sum of penalties of all k lines of the paragraph is minimized:

$$\forall B, B \text{ is Break Sequence } : P(B_{\min}) = \sum_{i=0}^{k-1} P_{w_{b_i+1}, w_{b_i+1}} \leq P(B)$$

Given n words there are 2^{n-1} possible ways to break them into a paragraph.[22] It is therefore prohibitive to simply check all possible combinations. In reality, only a few break sequences give good results, meaning that they have a low overall penalty. Note that it is not sufficient to solve the problem by simply using a first-fit algorithm, which would simply begin at the first line and makes choices as it sets each line: this approach can easily lead to line breaks that do not result in an overall minimum penalty; sometimes it is essential to set some earlier lines with a higher penalty so that later lines can be set with a lower penalty and thus reducing the overall penalty of the paragraph.

For the algorithm, Knuth introduced the concept of "feasible breakpoints"; a "feasible breakpoint" is a possible entry b_i in the final sequence $B_{\min}$ with the restriction that there exists a way, to set the paragraph from the very beginning to the word w_{b_i} so that the individual penalties of all the individual lines of the thus created subparagraph are within an allowed penalty range. When the algorithm runs, a list of "active breakpoints" (called *active list*) is maintained: this list contains those feasible breakpoints that might be chosen for future breaks.

Algorithm 2 shows a pseudo-code for Knuths' original optimal line break algorithm. The algorithm commences by adding the very beginning of the paragraph to the active list.[23] Then, subsequently all potential breakpoints (for our simplified case, these are just the breaks between words) are checked with respect to each breakposition in the active list. If there exists a way to set the line from an active breakpoint a to a potential breakpoint b so that the penalty of that line is within the allowed range, b is added to the active list. The algorithm also remembers, which of the possibly multible existing active breakpoints a gives the lowest overall penalty when setting the paragraph up to w_b using one active breakpoint a. The best predecessor is remembered so that the break sequence can be retrieved later. If a potential breakpoint b is encountered, for which an entry a in the active list returns an infinite penalty (this means, that the words from a to b can not be placed on one line), then a is removed from the active list.

The algorithm finishes by chosing the feasible breakpoint for the last word of the paragraph that has the lowest overall penalty. Because for each feasible breakpoint the predecessor is known, all previous breakpoints can be subsequently accessed.

[22] For a fixed number of lines k, there are $\binom{n-1}{k}$ ways to break the sequence of words. If the number of lines can vary from 1 to n, the resulting formula is $\sum_{k=0}^{n-1} \binom{n-1}{k} = 2^{n-1}$ (according to the binominal theorem).

[23] This location is not really a breakpoint; it is treated as a breakpoint so that the rest of the algorithm must not deal with the first line as a special case.

Algorithm 2 Optimal line breaks in text

```
Require: n ≥ 1 {Number of words}
  activelist := new list<Entry>;
  feasiblebreaks := new array<list<Entry>>;
  Entry := new Entry;
  Entry.pos := 0;
  Entry.penalty := 0;
  Entry.predecessor := -1;
  activelist.Add(Entry);
  {We go through all words}
  for i := 1 to n do
    feasiblebreaks[i] := LIST();
    for all a ∈ activelist do
      penalty := P_{a.pos+1,i} + a.penalty; {using penalty function (4.6)}
      {if the penalty gets too high, remove a from active list}
      if penalty = ∞ then
        activelist.Remove(a);
      end if
      if penalty < penaltyrange then
        Entry := new Entry;
        Entry.pos := i;
        Entry.penalty := penalty;
        Entry.predecessor := a.pos;
        feasiblebreaks[i].Add(Entry);
      end if
    end for
    {We only save the best entry}
    feasiblebreaks[i].RemoveAllButBestEntry();
    activelist.Add(feasiblebreaks[i].First());
  end for
  {Now we can construct the optimal break sequence B}
  {The sequence is created backwards}
  B := new list<int>;
  {Take the last entry added to the active list}
  a := activelist.Last();
  while a.predecessor > 0 do
    B.AddHead(a.pos);
    a := feasiblebreaks[a.predecessor].First();
  end while
  B.AddHead(0);
  return B;
```

The algorithm makes use of the principle of Dynamic Programming [Bel57, Pre00], which solves an optimization problem by solving a series of carefully devised subproblems. Each solution is subsequently obtained by combining the solutions of one or more of the previously solved subproblems. There are two places, where Dynamic Programming is applied in the line breaking algorithm: first, the calculation of $L_{i,j}$ (the length of a subsequence of words) does not need to be recalculated for every combination of an active breakpoint a and a potential breakpoint j. It is sufficient to just maintain a global variable $L_{1,j}$ that stores the length of all words from the beginning up until w_j. For each active breakpoint a, the length $L_{1,a}$ is stored. Then $L_{a,j}$ can be easily calculated as $L_{a,j} = L_{1,j} - L_{1,a}$. The second place, where Dynamic Programming is applied, is where the best previous breakpoint a is remembered when a feasible breakpoint is added to the active list. This can be done, because the subparagraph from the beginning up until w_b using the breakpoint a has the smallest total penalty. The line breaking of the remaing words (the lines after w_b) does not affect the previous subparagraph. Knuth puts it into the following words: "The optimum breakpoints for a paragraph are always optimum for the subparagraphs they create". This is even true if there is a way to set the paragraph from the beginning to w_b with a different number of lines using only feasible breakpoints. In this case, only one previous active breakpoint a gives the lowest overall penalty – the number of lines for the subparagraph is automatically chosen so that the global optimum is maintained. This very fact is very important when comparing text setting to music setting: if the number of lines of a paragraph must be variable, active breakpoints must be stored for each different line number therfore increasing running time and storage space. Also, even if the total number of lines does not matter, the described mechanism only works if the remaining lines all have the same width. Otherwise, it might be important which breakpoint occurs on which line in order to get the global minimum penalty. The running time of Knuths' algorithm is $O(n \cdot \alpha)$, where α is the maximum number of entries in the active list, which is just the maximum number of words in one line of text. As indicated above (and also described in Knuth), the running time increases, if the total number of lines needs to be adjustable and also if the individual lines of the paragraph have differing widths.

Optimal Line Breaking in Music Scores

Finding optimal line breaks in music has been previously discussed by [HG87]. Their work is very closely related to Knuths' algorithm, which has been described in the previous section. In order to adapt Knuths' algorithm to be usable for music, some specific musical issues must be considered. Instead of dealing with a collection of words, we now deal with a collection of system slices. A system slice (or slice for short) is a horizontal area of the score which cannot be further broken appart. For a first understanding of the problem, it is sufficient to treat system slices simply

as the measures of a score.[24] For each system slice, a separate space-force-function exists, which desribes, how this particular system slice will react to a stretching-force. Assuming that there exists a "natural" or "optimum" force f_{opt}, which results in esthetically pleasing lines of music, we can determine the "natural" extent for each slice by computing the respective space-force-function using the optimum force as the input. In order to determine the overall penalty for a line made up from a collection of system slices (compare to equation (4.6)), the space-force-functions for the participating system slices must be merged. Then the force required to stretch the merged space-force-functions to the given line length must be computed and the deviation with respect to the optimum force must be determined.

Let $S = (s_1, \ldots, s_n)$ be an ordered set of slices. A penalty function $\phi : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ takes two numbers i and j and calculates the badness of the line beginning at location s_{i+1} going to s_j. The objective is to find a sequence $B_{\min} = (b_0, b_1, \ldots, b_m)$ for which the value:

$$P(B_{\min}) := \sum_{i=0}^{m-1} \phi(s_{b_i}, s_{b_{i+1}})$$

is minimal. Note that $b_0 = 0$ for every sequence B. The resulting number of lines is then m.

In order to calculate the sequence $B_{\min}$ that minimizes P, Algorithm 3, which is an adaption of Knuths' original algorithm, is used.

Algorithm 3 is quite similar to Knuths' algorithm. Instead of maintaining an active list, an entry is stored for each slice, which saves the optimum predecessor and the overall penalty of breaking at that particular slice. The algorithm automatically leads to justified lines (including the last one), because the penalty function returns smaller penalties for full lines.

Just as the text break algorithm, the line break algorithm for music makes use of the principles of Dynamic Programming [Bel57], because for each slice, only the best way to break the previous slices is remembered. The running time of the algorithm is bound by the number of slices (n, the first loop), and the maximum number of slices per line (the maximum value for $j - i$, where $\phi(i, j) \neq \infty$). We denote this parameter α. For all practical purposes, we can probably assume that $\alpha < 10$ for most scores. Because the algorithm does not contain an explicit value for α, it works for any value found in real scores. The overall running time is then $O(n \cdot \alpha \cdot O(\phi))$. This shows that the running time of ϕ is a crucial part of the optimal line break algorithm.

The actually implemented algorithm has to deal with certain musical aspects: when a line of music is broken, the following line gets a clef and a key-signature. It is therefore important to leave room (horizontal space) for these notation elements.

[24] As there are instances in music notation, where line breaks do occur at positions which are *within* a measure, or there are even scores with no measure information, the general algorithm just deals with measure-independent slices.

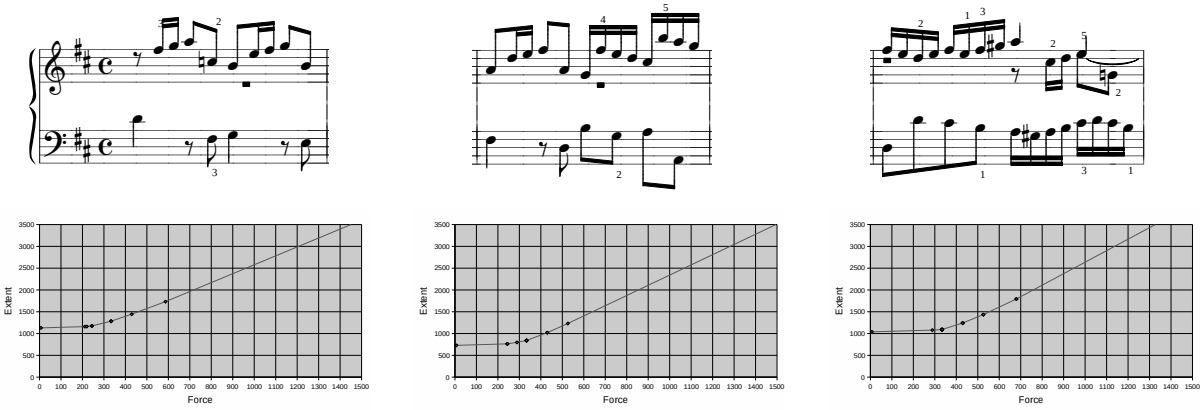
Algorithm 3 Optimal line breaks in music

```
E := new array<Entry>;
E[0].penalty := 0;
E[0].predecessor = 0;
for i = 1 to n do
  E[i].penalty := ∞;
  E[i].predecessor = -1;
end for
for i = 0 to n − 1 do
  if E[i].penalty < ∞ then
    prevpenalty := E[i].penalty;
    for j = i to n do
      newpenalty := φ(i, j); {Using equation (4.7)}
      if newpenalty + prevpenalty < E[j].penalty then
        E[j].penalty := newpenalty+ prevpenalty;
        E[j].predecessor := i;
      end if
      if newpenalty ≥ ∞ then
        break; {End the inner loop}
      end if
    end for
  end if
end for
{Now we can retrieve the Break Sequence}
B := new list<int>;
int k := n;
while E[k].predecessor > 0 do
  B.AddHead(k);
  k := E[k].predecessor;
end while
B.AddHead(0);
return B;
```

Because the clef and the key-signature can change within a piece, a pointer to the current clef and key has to be stored with each potential break location.
The penalty function ϕ is calculated as follows:

$$\phi(s_i, s_j) = |\,\mathrm{sff}_{s_i,s_j}(lineextent) - f_{\mathrm{opt}}| \tag{4.7}$$

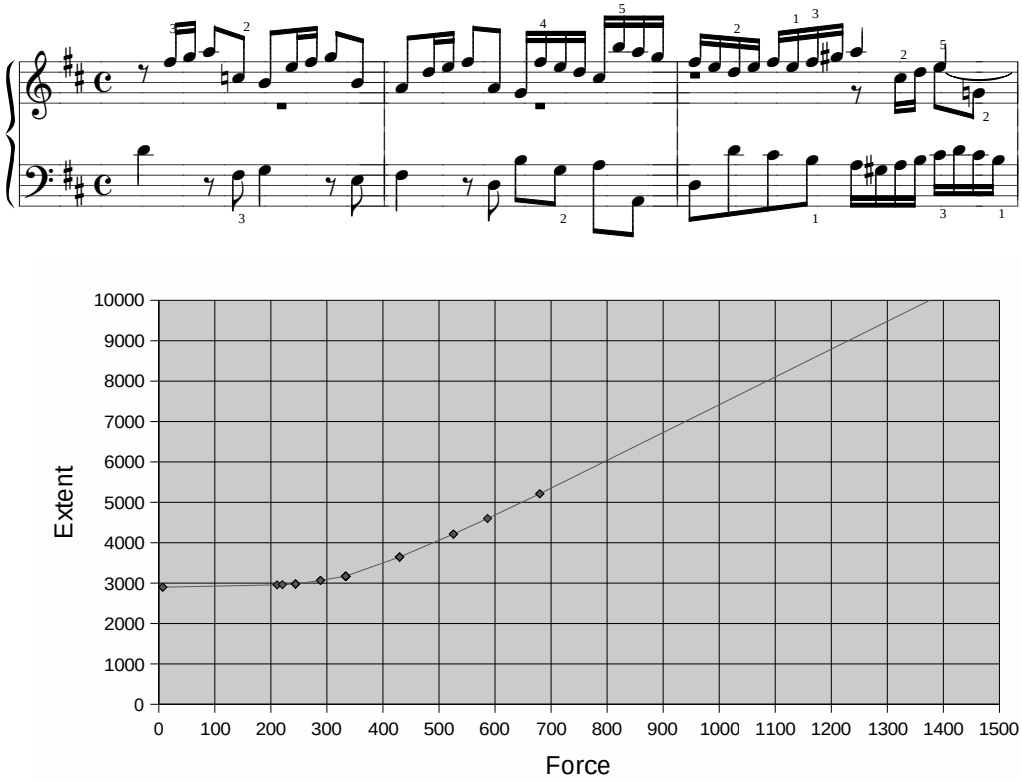
where sff_{s_i,s_j} is just the merger of all sff's from slice s_{i+1} up to slice s_j, $lineextent$ is the desired width of a line and f_{opt} is a constant value set by the user (and dependent on personal taste). Merging the sff's is a somewhat expensive operation: the merging of two space-force-functions sff_1 and sff_2 has a running time of $O(\max\{|\,\mathrm{sff}_1\,|, |\,\mathrm{sff}_2\,|\})$, where $|\,\mathrm{sff}\,|$ is the number of springs in the space-force-function. If α slices are merged, each containing a maximum of β springs, the overall cost is $O((\alpha-1)\cdot\beta)$. Together with the formula from above, we get a running time for Algorithm 3 of $O(n\cdot\alpha\cdot\alpha\cdot\beta)$. Because most of the lines will later not be used, and a complete merger of the sff's is so expensive, a quicker (and less exact) mechanism was actually implemented: Because a sff is a piece-wise linear function, it is possible to identify a *slope* for each linear section. This slope directly corresponds to the added spring constants of those springs that are active at the respective force interval. In order to efficiently approximate the merger of different sff's, the spring-constant for the linear segment at the optimum force is saved. This saved spring-constant approximates the function in the neighborhood of the optimum force. An approximate merging of different sff's is then possible by just determining the overall spring-constant using equation (4.1), which then just requires α multiplications. Figure 4.20 demonstrates, how this optimization compares to the exact merging of sff's. The upper part of Figure 4.20 shows three measures of a Bach piece. Each measure is treated as a separate slice with its own sff. The spring constants at the optimum force are shown below the scores. For the approximation, the individual spring constants are simply multiplied using equation (4.1). To determine the approximated force required to stretch the three measures to a desired width of 4280, Hooks' Law is used with the approximated values. This results in an approximated force of 533.43. The lower part of Figure 4.20 shows the actual merged space force function for the complete three measure line. The actual force required to stretch the system is 536.3, so the difference from the approximate value is less than one percent! For all practical purposes, it is therefore sufficient to work with the approximated values. The running time of ϕ is thus reduced to α multiplications and additions.
To finally demonstrate, how the optimal line break algorithm works on a 27 measure long Bach fugue, consider Figures 4.21 and 4.22. The 27 measures are broken into 12 lines covering two and a half pages. The individual lines are spaced evenly and the overall appearance is quite good. The overall penalty for all lines is 1777.37, which gives an average penalty of $1777.37/12 = 148.11$ for each line. The user-selected optimum force for this particular calculation was set to 800. Figure 4.22 shows the calculated array of entries. The nodes in the graph represent ends of measures together with the overall penalty, where a line break may occur.



$$c_{\text{opt}} = 0.488976 \qquad c_{\text{opt}} = 0.427088 \qquad c_{\text{opt}} = 0.379205$$
$$x_{\text{opt}} = 533 \qquad x_{\text{opt}} = 0 \qquad x_{\text{opt}} = 0$$

$$c_{\text{app}} = \frac{1}{\frac{1}{0.488976}+\frac{1}{0.427088}+\frac{1}{379205}} = 0.142363$$
$$x_{\text{app}} = 533$$

Force required for width 4280: $(4280 - 533) \cdot 0.142353 = 533.43$



$$c_{\text{opt}} = 0.145318$$
$$x_{\text{opt}} = 534$$

Force required for width 4280: $\text{sff}(4280) = 536.3$

Figure 4.20: Exact and approximate merging of sff's

The edges in the graph depict penalties associated with a line beginning at the end of the measure in node 1 and ending at the end of measure of node 2. As an example consider the lines beginning at the very bottom of Figure 4.22: The first line of the piece begins at end of measure 0 and goes to the end of measure 2 (penalty: 188.106) **or** measure 3 (penalty: 291.758) **or** measure 4 (penalty: 835.235). Figure 4.22 shows the optimum path and the first fit path, which is not the same: in order to demonstrate that the first fit algorithm leads to a bigger overall penalty, the entries for measure 25 and 27 are each shown twice. The actual algorithm only saves the entries for the optimum line breaking path. Additionally, when calculating the optimal line break, only one predecessor is saved for each node. Figure 4.22 shows more then one predecessor to demonstrate that the algorithm tries different break combinations before the optimum solution is determined. To determine the breaking sequence, the algorithm begins at the top of Figure 4.22 in the node labeled "27/1777.37". The dashed edges are subsequently followed to produce the sequence $27, 25, 22, 20, 18, 16, 13, 11, 9, 7, 5, 3, 0$. Reversing this sequence shows, which measures are endings of lines.

4.5.3 Optimal Page Fill in Music

As indicated before, music is usually printed so that a complete piece completely fills the pages it is being printed on. This is different in book printing for example, where the last paragraph may end almost anywhere on the page. Even though one professional music notation program (SCORE by Leland Smith) has an integrated page fill algorithm, nothing on this subject has been published so far. The most popular notation programs (such as Finale, or Sibelius) do not even offer an optimal line breaking algorithm, although the results published by Hegazy and Gourlay are fairly easy to implement. During the work on this thesis, an optimal page filling algorithm was developed. This new algorithm is an extension of the optimal line breaking algorithm of the previous section. It computes optimally filled pages with lines of music that are esthetically pleasing. The new algorithm is also scalable so that it can be adopted to either run with maximum speed or to produce optimum output.
A definition of the problem can be given similar to before: Let $S = (s_1, \ldots, s_n)$ be a set of ordered (system) slices. Each slice has an associated height h_i, which determines, how much vertical space the slice requires on a page. The height function is defined as follows:

$$\text{height}(i, j) := \max\{\text{height}(s_k) \mid i < k \leq j\} \tag{4.8}$$

Where $\text{height}(s_k)$ is simply the height of slice s_k.[25] A slice also has an associated sff (just as in the case of optimal line breaks). The objective is to find a break se-

[25] In certain (rare) situations, equation (4.8) might lead to false results: there might be scores, where multiple staves require different heights within one line of music. As a first estimation for the algorithm, equation 4.8 is sufficient.

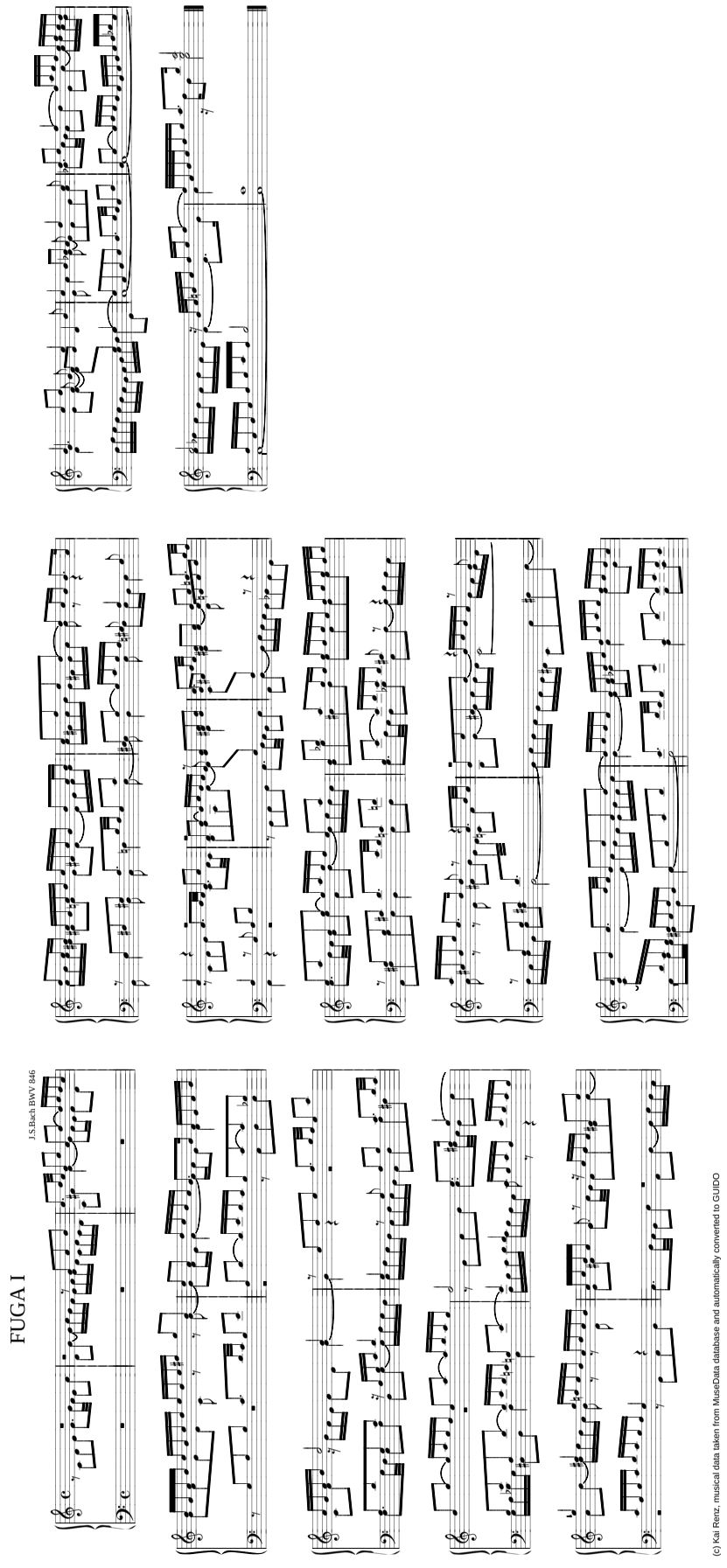


Figure 4.21: A Bach fugue broken with the optimal line fill algorithm

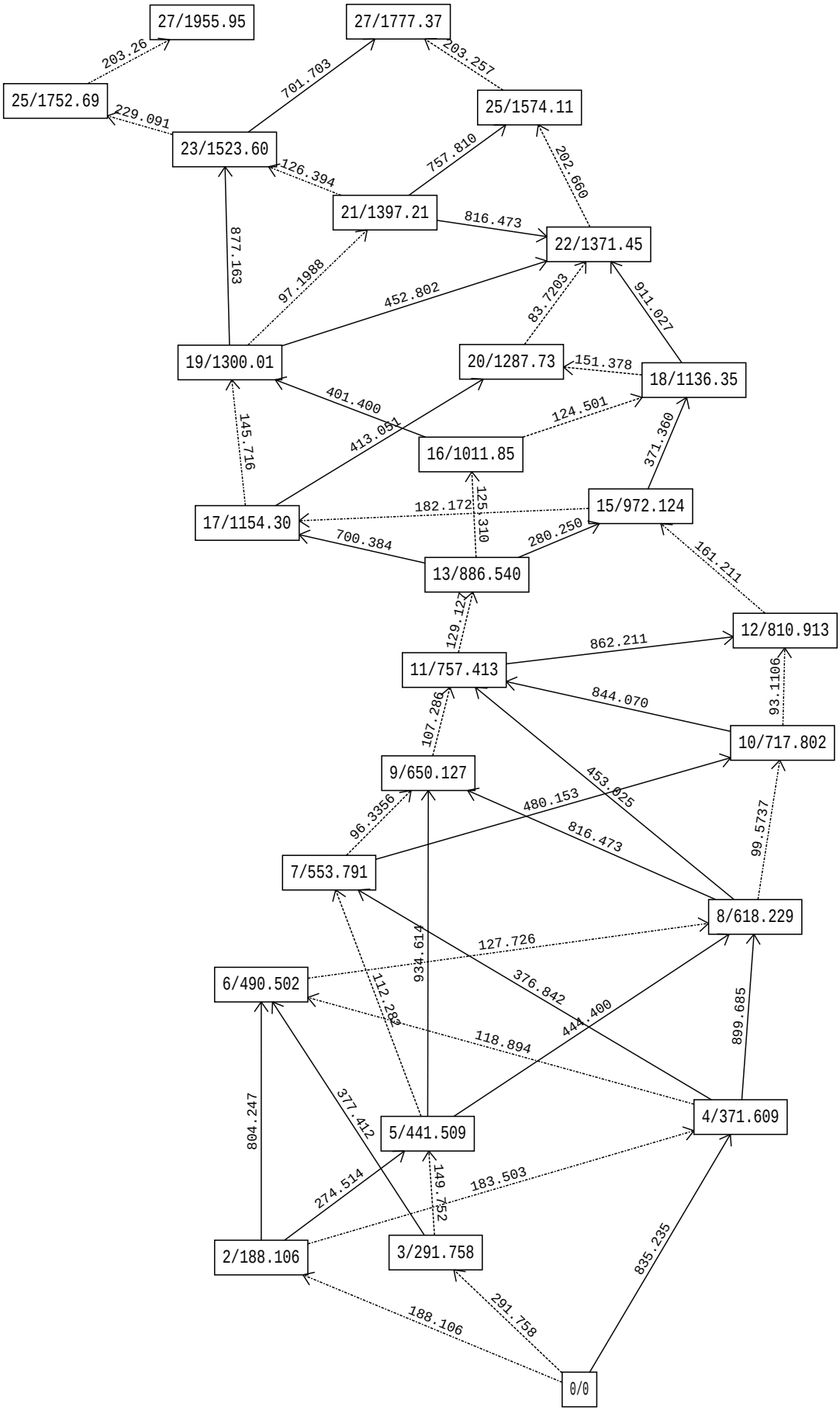


Figure 4.22: Partial calculation graph for the optimal line break of the Bach fugue

quence $B_{\min} = (b_0, b_1, \ldots, b_m)$ where each b_i contains three parameters: a slice number (referred to as $b_i.pos$ below), a position on a page (referred to as $b_i.height$ below), and a flag, which is set, if the slice is the location of a page break (referred to as $b_i.pagebreak$); as before, $B_{\min}$ should minimize the penalty function P

$$\forall B, B \text{ is Break Sequence } : P(B_{\min}) = \sum_{i=0}^{k-1} P_{w_{b_i.pos+1}, w_{b_i.pos+1}} \leq P(B)$$

and additionally the following four conditions must be met:

1. $\forall 1 \leq i < m$ and $b_i.pagebreak =$ **false** :
   $b_{i+1}.height = b_i.height + \text{height}(b_i.pos, b_{i+1}.pos)$
2. $\forall 1 \leq i < m$ and $b_i.pagebreak =$ **true** :
   $b_{i+1}.height = \text{height}(b_i.pos, b_{i+1}.pos)$
3. $\forall 1 \leq i < m : b_i.pagebreak =$ **true** $\Leftrightarrow$
   $b_i.height + \text{height}(b_i.pos, b_{i+1}.pos) > pageheight$
4. $b_m.height \geq 0.75 \cdot pageheight$

The first condition means that the height of each break location increases as long as no page break is encountered. The second condition means that the height is reset if a page break is encountered. The third condition states that a page break may *only* occur, if the following line does not fit on the current page. The last condition requires the final measure to end on the last 75 percent of the page.
The algorithm works by dividing a page into a distinct number of "page areas", which are called slots. Figure 4.23 shows an example page that is divided into 12 different slots. There are three *allowed* ways to break the first eight measures of the Bach fugue (BWV 846) into lines. This means that each breaking sequence of Figure 4.23 contains only lines of music whose individual penalties are within the (user defined) penalty range. On the bottom of Figure 4.23, only two lines are required to set the eight measures. Here, measure eight ends in slot 5. The overall penalty of the two lines is 1734.92, which is a very high penalty, which is not surprising considering how tight the two lines are spaced. In the middle of Figure 4.23, the end of measure eight ends in slot 7. The penalty associated with this particular breaking sequence is 796.896. At the top of Figure 4.23 the end of measure eight ends in slot 9. The penalty associated with this setting is 618.229. When dealing with optimal page fill, it is important to remember each one of the different possibilities to break the music together with the associated slot. To meet this goal, the algorithm needs to deal with the *height* of slices and lines. Using the required height of a collection of slices, the output slot can be computed. For each slot only the best way break sequence is saved – this is where Dynamic Programming is used again.
The maximum number of slots, which is denoted β, directly determines how fast the algorithm runs. The running time is bound by $O(n \cdot \alpha \cdot \beta)$. Essentially, the algorithm

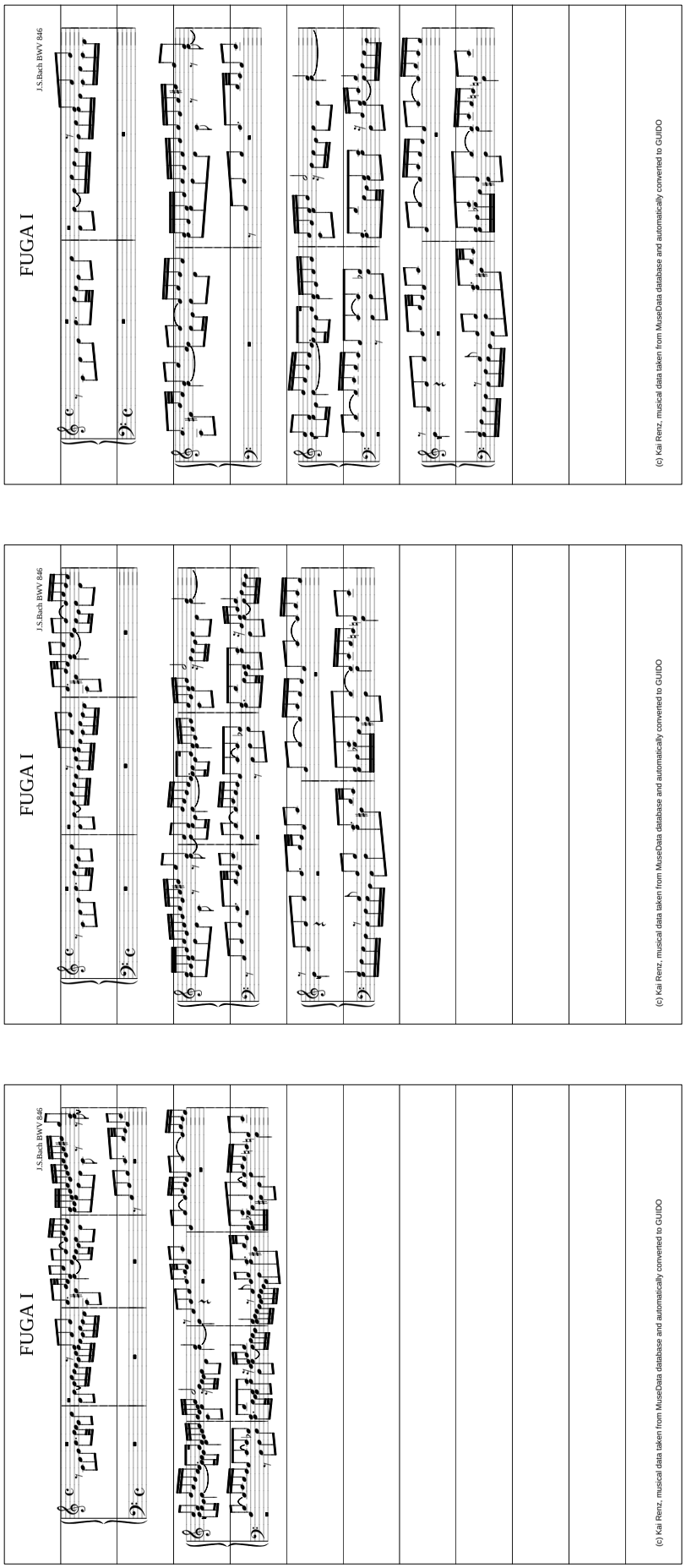


Figure 4.23: A page divided into 12 slots

carries out the optimum line fill algorithm β times. To determine the optimum break sequence that also fills the last page completely, the algorithm simply picks a small overall penalty value from the last slice which ends on the lower part of the page. Going backwards from this entry reveals the locations for line- and page breaks. Note, that the algorithm determines the number of pages automatically; this is similar to the optimal line break algorithm, which determines the number of lines automatically.

The user can manipulate the algorithm by numerous ways: either by specifying direct line- or page-breaks and by discouraging or encouraging certain breaks. These flags are directly used by the penalty function and result in optimum break sequences, which also consider user input.

Algorithm 4 is an extension of Algorithm 3. For each slice, there are now β slots reserved for entries that save the predecessor (slice *and* slot), the accumulated penalty and the accumulated height. The algorithm iterates through all slices (loop for $i = 0$ to $n-1$). For each slice, all β entries (loop for $j = 0$ to β) are taken as a starting point for future lines. This is quite similar to the optimal line breaking algorithm. The optimal page fill algorithm does not only compute the penalty of a potential line beginning at end of slice i and going up until slice k, but it also computes the required height. If the accumulated height of the previous ending slice (this is called *prevheight* in the algorithm) added to the height of the current line is bigger than the page height, then the new line is placed on a new page. The accumulated height is used to determine in which slot the calculated break position should be placed. Only if the calculated penalty is smaller than the slots' current value, the new value replaces the old calculation; this is again an application of Dynamic Programming. If the penalty of the potential lines becomes too large, the next slot of the current slice is calculated. When the algorithm has finished, the calculated entry array is used to retrieve the page break sequence that fills all pages. The procedure to retrieve the sequence is shown in Algorithm 5. The algorithm first looks at the lower part of the last slice: it finds the minimum penalty for the lowest quarter of the page. If no value is found (this mostly happens, if less than one page of music is being set), then the lowest overall penalty is determined – this entry is similar to the optimal line break algorithm. Finally, the break sequence is determined by going backwards from the last entry.

In order to demonstrate, how the optimum page fill algorithm works, Figure 4.24 shows the Bach fugue (BWV 846) of the previous section automatically being set on exactly two pages. Figure 4.25 shows the calculated entryarray grid. The measures of the finally chosen path are highlighted. Note that the optimum line break path of the previous section (requiring 3 pages) is also present in the calculated array. Figure 4.25 also shows that the first fit approach leads to a bigger overall penalty than the optimal line break algorithm.

Algorithm 4 Optimal page fill algorithm

```
entryarray := new array<Entry>[n+1][β + 1]; {Create Entry array}
entryarray[0][0].penalty := 0;
entryarray[0][0].predecessorslice := 0;
entryarray[0][0].predecessorslot := 0;
for i := 1 to n do
  for j := 0 to β do
    entryarray[i][j].penalty := ∞;
    entryarray[i][j].predecessorslice := -1;
    entryarray[i][j].predecessorslot := -1;
    entryarray[i][j].height := -1;
  end for
end for
for i := 0 to n − 1 do
  for j := 0 to β do
    if entryarray[i][j].penalty < ∞ then
      prevpenalty := entryarray[i][j].penalty;
      prevheight := entryarray[i][j].height;
      for k := i to n do
        ispagebreak := 0;
        newpenalty := φ(i, k); {Using equation (4.7)}
        newheight := height(i, k);
        if prevheight + newheight > pageheight then
          prevheight := 0;
          ispagebreak := 1;
        end if
        newslot := (int) ( ((prevheight+newheight)/pageheight) · β );
        if newpenalty + prevpenalty < entryarray[k][newslot].penalty then
          entryarray[k][newslot].penalty := newpenalty + prevpenalty;
          entryarray[k][newslot].predecessorslice := i;
          entryarray[k][newslot].predecessorslot := j;
          entryarray[k][newslot].height = newheight + prevheight;
          entryarray[k][newslot].ispagebreak := ispagebreak;
        end if
        if newpenalty > ∞ then
          break; {Break the inner loop}
        end if
      end for{for k := i to n}
    end if{if entryarray[i][j].penalty < ∞}
  end for{for j := 0 to β}
end for{for i := 0 to n − 1}
return entryarray;
```

Algorithm 5 Retrieval of the page break sequence

```
Require: n = number of slices;
Require: β = number of slots;
Require: entryarray[n][β] calculated by algorithm 4
  kminpage := β+1; {Determine minimum penalty for lower part of the page}
  minpagepenalty := ∞;
  for j := (int) (0.75 · β) to β do
    if entryarray[n][j].penalty < minpenalty then
      kminpage := i;
      minpagepenalty := entryarray[n][i].penalty;
    end if
  end for
  if kminpage > β then {There is no entry at the end of the page}
    kmin := β + 1; {Determine the minimum penalty}
    minpenalty := ∞;
    for j := 0 to β do
      if entryarray[n][j].penalty < minpenalty then
        kmin := j;
        minpenalty := entryarray[n][j].penalty;
      end if
    end for
    index := kmin;
  else
    index := kminpage;
  end if
  B := new list<int>;
  k := n;
  while entryarray[k][index].predecessorslice > 0 do
    B.AddHead(k);
    prevslice := entryarray[k][index].predecessorslice;
    index := entryarray[k][index].predecessorslot;
    k := prevslice;
  end while
  B.AddHead(0);
  return B;
```

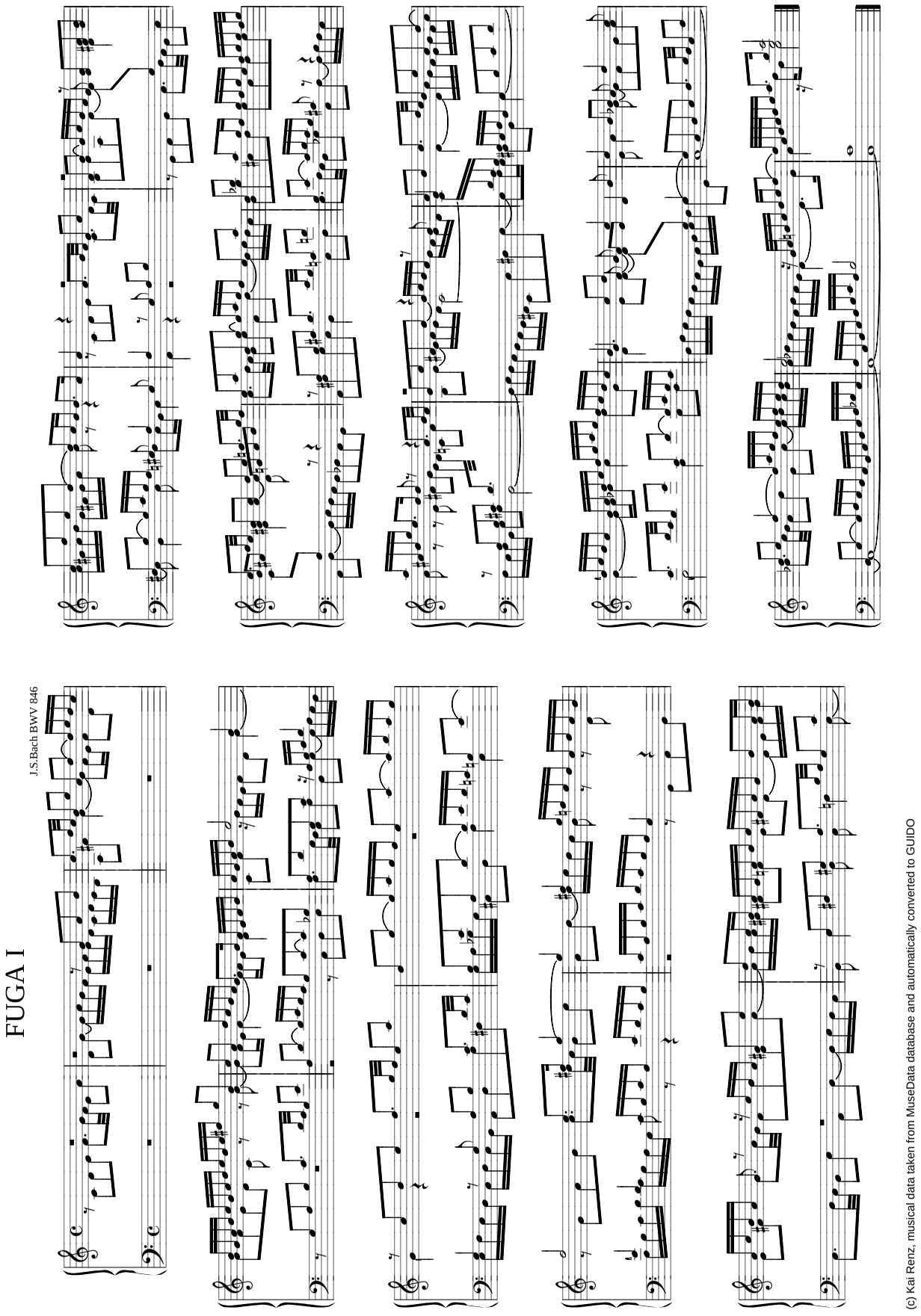


Figure 4.24: The Bach fugue (BWV 846) set by the optimum page fill algorithm

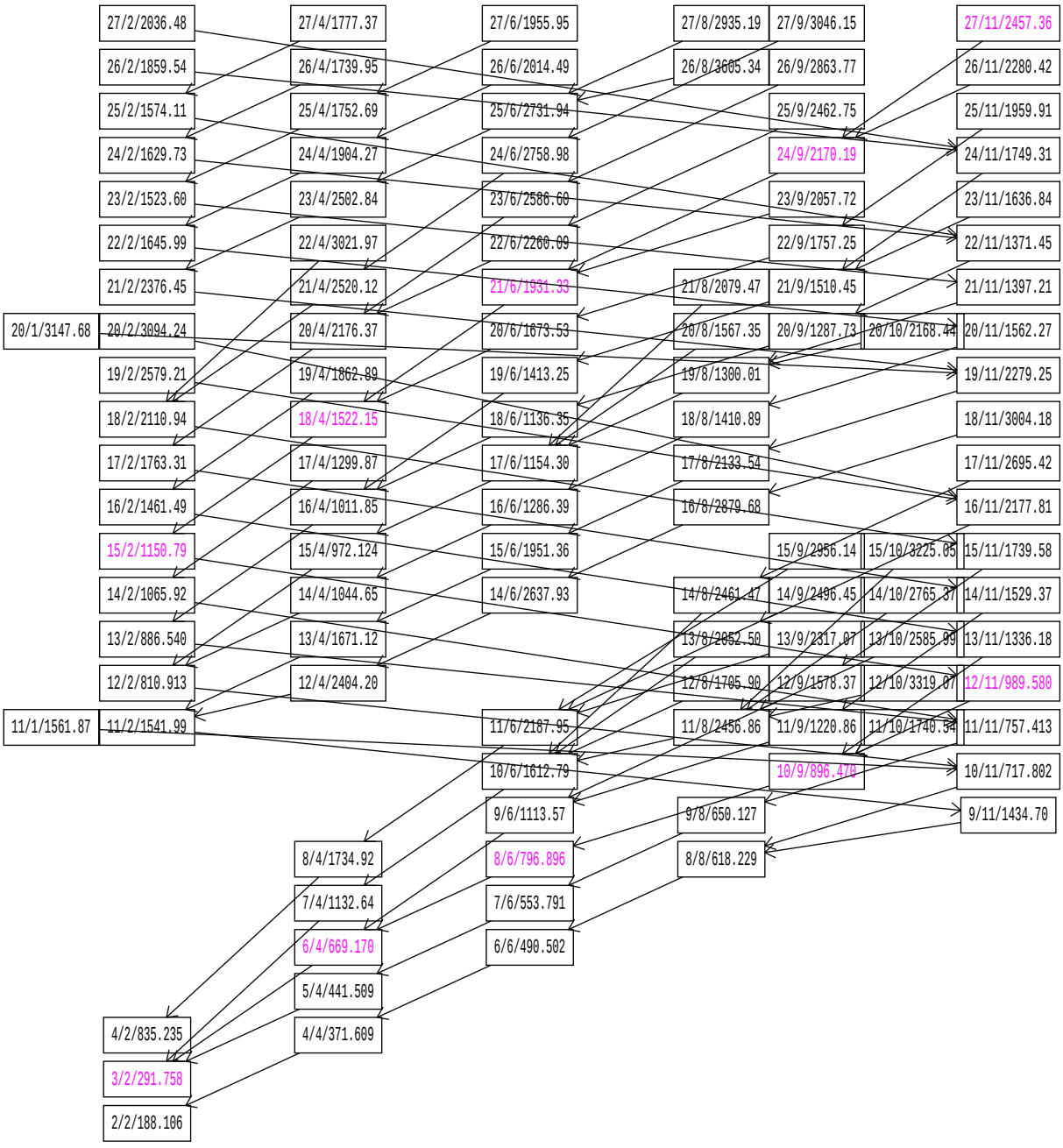


Figure 4.25: The calculated graph for the optimum page fill of the Bach fugue (BWV 846)

Further Thoughts on Line Breaking

Some of the assumptions used when describing the optimal line break and page fill algorithm must be reconsidered: mainly the assumption that slices are simply treated as measures does not hold for all types of music. There are scores, where breaks in between measures are necessary and lead to much better scores than it is the case if only complete measures can be broken. Additionally, some music does not contain measures, or different voices/staves can contain different meters and therefore different bar lines. Then, the question of detecting suitable break locations gets complicated. In rare cases, music can be constructed that contains *no* possible break locations, because there are *no* common onset times for all given voices (see also the AutoBreaks-algorithm on page 73).
Another issue that has to be mentioned is the question of how the slices are merged and how rods are treated that lie at slice boundaries. In the case of bar lines, the issue is pretty straight forward. If other break locations are considered, it must be asserted that several slices can be merged into one line without visible disruptions.
Finally, Knuth's line break algorithm has been fine tuned over a period of time by carrying out user experiments. For example, Knuth mentions how loose lines being followed by tight lines disturb the visual appearance. When dealing with optimal line breaks, the fine-tuning of the penalty function ϕ and the setting of the optimum force and the allowed penalty for individual lines must be carefully adjusted. This certainly requires some experience with a large body of scores.

4.6 Conclusion

This chapter dealt with the process of converting an arbitrary GUIDO description into a matching graphical representation of a score. First, it was shown that some of the required music notation algorithms can be directly described as GUIDO to GUIDO transformations. Then, those music notation algorithms, which operate directly on the Abstract Representation were presented. In the following, the graphical requirements for any conventional music notation system were exemplified and the Graphical Representation (GR), which is an object-oriented hierarchical structure representing all visual elements of a score was introduced. Subsequently, the conversion of a GUIDO description (and its corresponding AR) into the matching GR was explained. Finally, an improved algorithm for spacing a line of music was presented. Additionally, a completely new algorithm for optimal page fill was introduced, which makes optimal use of all pages required by the score.
The algorithms and procedures described in this chapter show that the creation of a score is a complicated task that strongly depends on the underlying data structure. In order to create "good" music notation, great care has to be taken when designing the overall structure and the available manipulation routines. Overall, the creation of conventional scores can be managed (at least partially) by using clever

algorithms.
Obviously, some issues, which are essential for creating esthetically pleasing music notation, have not been discussed, although they have been – at least partially – implemented in the current system. These issues concern subjects like a completely automatic collision detection (and prevention) or, for example, the exact and complete description of how to calculate beam-slopes for any conceivable case. While some of these techniques have been described elsewhere (see for example [Gie01]), some have not been dealt with in the accessible literature. A system capable of automatically avoiding all sorts of collisions of musical symbols has not been yet realized, although it seems that progress using "intelligent" approaches is possible; a fruitful endeavor might be the use of random local search algorithms in conjunction with evaluation functions to avoid local collisions of symbols. It is nevertheless a futile task to estimate if such a system will be available in the near or more distant future.

Chapter 5

Applications

The previous chapters dealt with the algorithms and data structures involved during the conversion of arbitrary GUIDO descriptions into conventional scores. This chapter now presents the notation system, which has been designed and implemented during work on this thesis. All of the ideas presented in the previous chapters are incorporated in the so called GUIDO Notation Engine (GNE). The objective of the GNE is the conversion of an arbitrary GUIDO description into a graphical score. The GNE is realized as a platform independent library, which offers well defined interface functions that can be used by applications, which require conventional music notation to be displayed. The GNE has been compiled as a Dynamic Link Library (DLL) for Windows and as a shared object library for Linux. As the complete source code is written using ANSI C++, it is easily possible to compile the notation engine on other systems. Currently, three different output mechanisms for a graphical score exist when using the GNE: a score can be retrieved as a GIF picture (which is a bitmap format), or it can be drawn directly into a window (using operating system dependent drawing routines), or a so called GUIDO Graphic Stream (GGS) can be retrieved, which is a platform independent way to describe a rendered score. The first two retrieval methods are currently available only when using the Windows-DLL, while the third approach is available using Linux or Windows.
At the moment, the GNE is used in three different application contexts: in connection with a standalone notation viewer, as an online service using standard Internet services, and as a (prototypical) interactive notation editor based on Java. These applications together with extensions made possible through them, are presented in this chapter.

5.1 The GUIDO NoteViewer

The GUIDO NoteViewer is an application for displaying and printing musical scores, which are created from arbitrary GUIDO descriptions. The application is currently

running on Windows PCs. As the name suggests, the GUIDO NoteViewer is *not* a notation editor. The NoteViewer reads GUIDO descriptions and creates a screen display or a printer output of the rendered score. The user can zoom in and out of the score; it is possible to scroll within the window and to change the displayed page number. The GUIDO NoteViewer is somewhat similar to the Acrobat Reader, which displays PDF documents, but does not allow them to be changed interactively.
Figure 5.1 shows a screen shot of the GUIDO NoteViewer. The GUIDO description of the score in the lower window is visible in a text-editor.

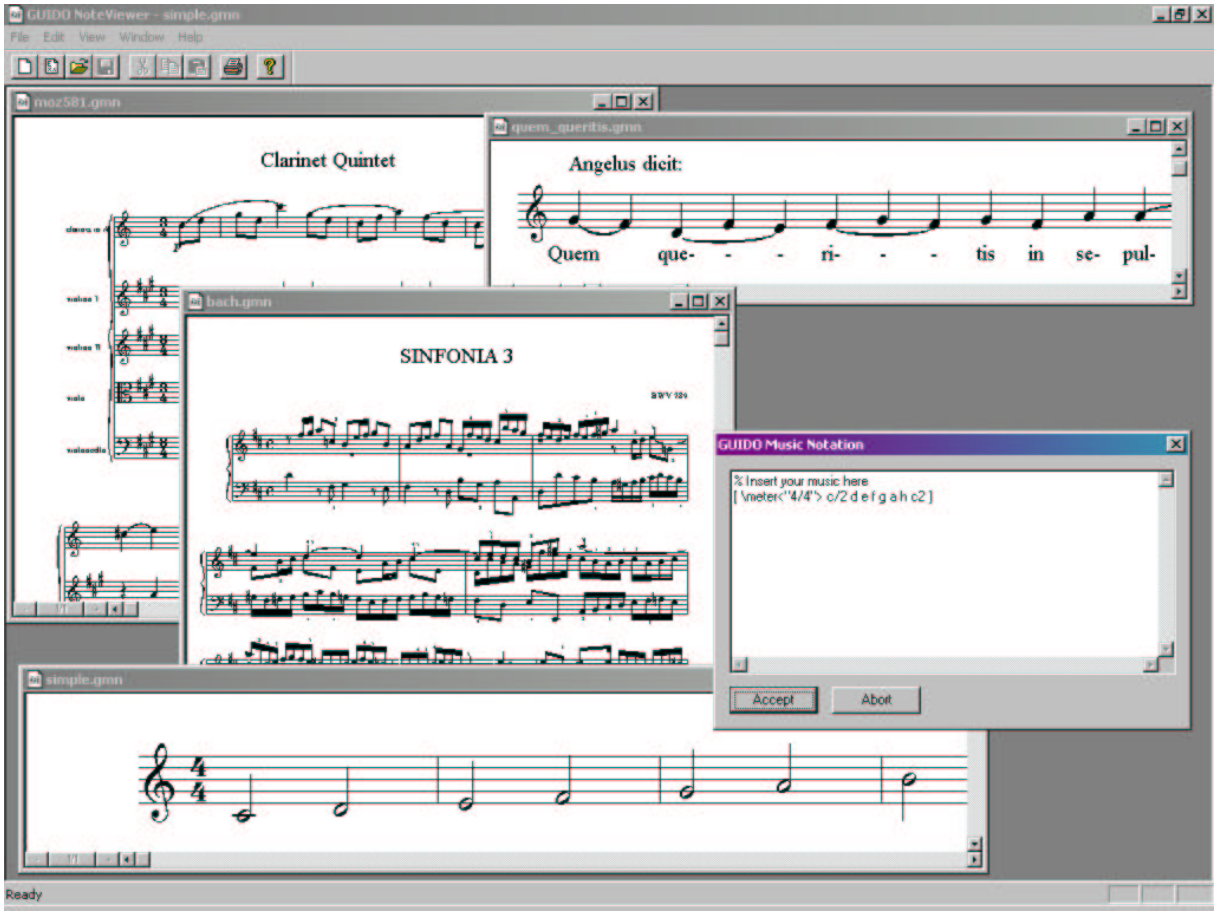


Figure 5.1: The GUIDO NoteViewer

5.1.1 Architecture of the GUIDO NoteViewer

The GUIDO NoteViewer is a combination of several components, which are shown in Figure 5.2. Within the main module, which is called `gmnview.exe`, flow control is handled through the "Flow control"-unit. User input consists of either a GUIDO description or a MIDI file. If the input is a MIDI file, it is first converted into

a GUIDO description using the `midi2gmn` component, which has been realized by J. Kilian [Kil99]. The flow control unit then sends the GUIDO description to the GNE (called nview32.dll), where it is prepared and converted into an internal score representation, as it was shown in the previous chapters. The flow control unit also creates a window on the screen, in which the GNE directly draws the score using functions supplied by the operating system. All user interaction (like, for example, the zooming or changing of the page number) is handled by the flow control unit, which sends formatting information to the GNE, which then redraws the score.

Another component deals with audio playback using MIDI: the flow control unit sends a GUIDO description to the `gmn2midi` module, which creates a MIDI file. This MIDI file is then played using routines provided by the operating system.

A text editor window within `gmnview.exe` can be used to edit the textual GUIDO description of any displayed score. If the GUIDO description is changed by the user, the flow control unit resends it to the GNE, which then redraws the score in the respective score window. The changed description may also be saved in a file.

Printing is handled similar to screen-display: all the printing commands are generated by the GNE through functions provided by the operating system.

The communication between the flow control unit and the individual modules follows a well defined interface, which is partly described in the following subsection.

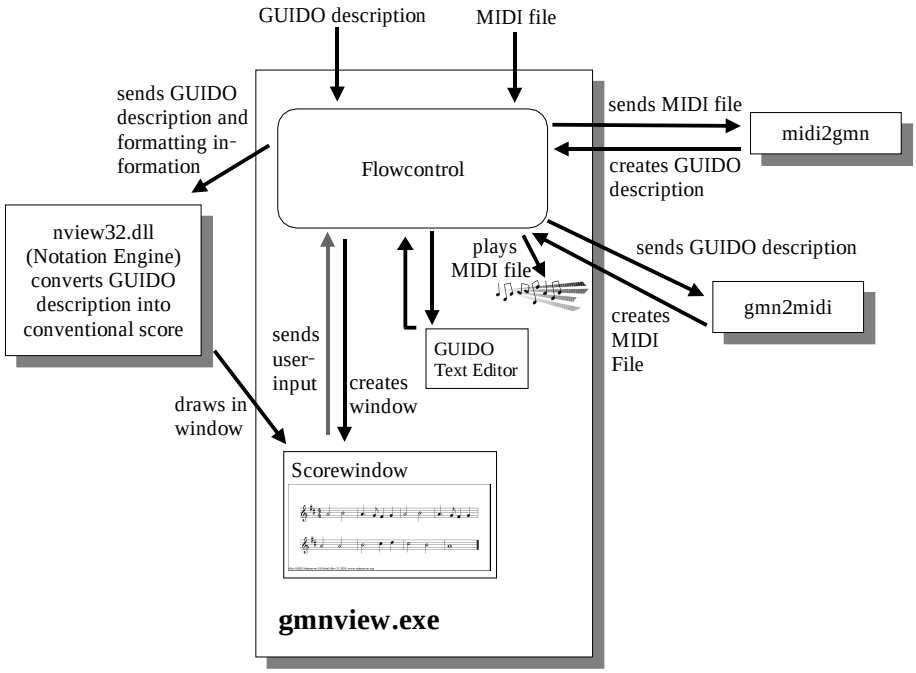


Figure 5.2: The Architecture of the GUIDO NoteViewer

5.1.2 Interface Functions of the GNE

There are more than twenty interface functions that can be used to interact with the GNE. In the following, only the most important ones are shortly described:

- `nview_parse` This function is used to start a new conversion process: a filename for a GUIDO description file is provided; the file is parsed by the GNE and converted into an internal score-representation. Finally, an integer-handle is returned to the caller. This handle is then used in all of the following routines to uniquely specify the process.

- `FreeHandle` This routine is called, when a process can be terminated (this happens, if a user closes a score-window). In this case, the GNE frees the internal storage space for the process, which is identified through the unique handle.

- `OnDraw` This function is used for score display and printing. Additional formatting information is supplied by the caller: a zoom-factor, scrolling-parameters, a page-number, the size and a handle of the display-window. Additionally, a flag can be set to trigger printing or Postscript-output. The GNE then draws (or prints) the score within the window (or on paper).

- `getNumPages` This routine can be used to determine the number of score pages that were created by the conversion process. This is important for handling the scrolling and printing of individual pages.

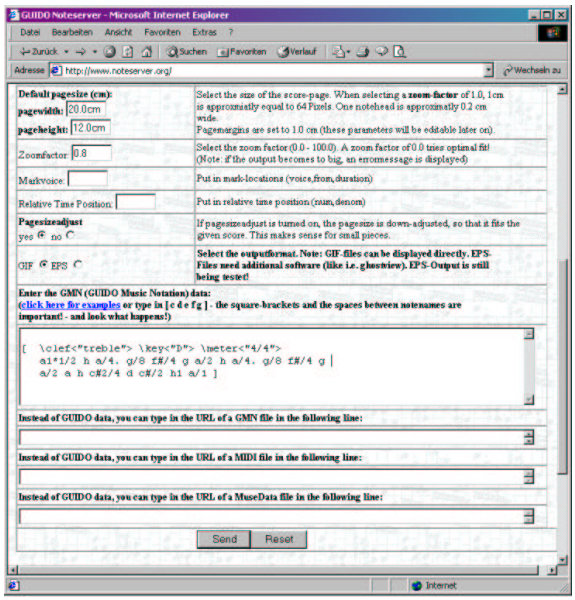
- `exportMusic` This routine retrieves the transformed GUIDO description: as was shown in the previous chapters, the conversion of an arbitrary GUIDO description into a score can be described as a GUIDO to GUIDO transformation. `exportMusic` returns the *final* GUIDO description, which is the result of all transformation routines contained in the GNE.

Several more interface functions are provided that are used in different application-contexts, which will be described later in this chapter.

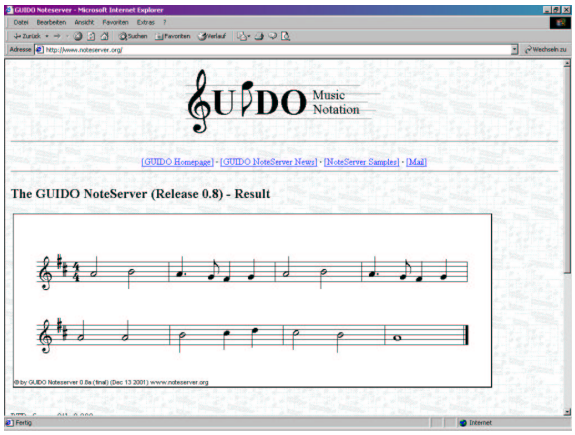
5.2 The GUIDO NoteServer

Two aspects of GUIDO Music Notation make it an ideal candidate for online usage: first, it is a text-based language, which means it can be created very easily either by hand or automatically; second, it is adequate in the sense that a simple score has a simple GUIDO description. The second feature also implies that simple GUIDO descriptions can often be understood intuitively. As an almost natural consequence, an online score notation service based on GUIDO Music Notation was developed. This application, which was realized using the GNE, is the GUIDO

NoteServer. The GUIDO NoteServer is a free online service, which converts GUIDO descriptions into images of conventional scores, which can be displayed in a standard Internet-browser [RH98, RH01]. Figure 5.3 shows a screen shot of the GUIDO NoteServer: on the left, the default input page is shown; on the right, the resulting score image is displayed.



(a) Input Page



(b) Result Page

Figure 5.3: The GUIDO NoteServer

The GUIDO NoteServer is realized as a collection of Perl-scripts [WCS97] and several executables. The notational logic is encapsulated in the same Dynamic Link Library (nview32.dll) that is used within the GUIDO NoteViewer. Figure 5.4 shows the basic architecture of the GUIDO NoteServer. First, a GUIDO description is created using one of the methods shown at the top: several converters can be used to transform different music representation formats into GUIDO descriptions, for example converting MIDI files or the MuseData format. The GUIDO description is sent to the NoteServer using CGI, which is a client-server-protocol supported by almost any web server and browser. Within the NoteServer, the GUIDO description is converted into a score by using the GNE. The formatting parameters are first evaluated using Perl-scripts; then, the necessary executables are called. Finally, the output is created and the result is sent back to the calling client.

Several access modes for the GUIDO NoteServer exist, all of which will be described in the following.

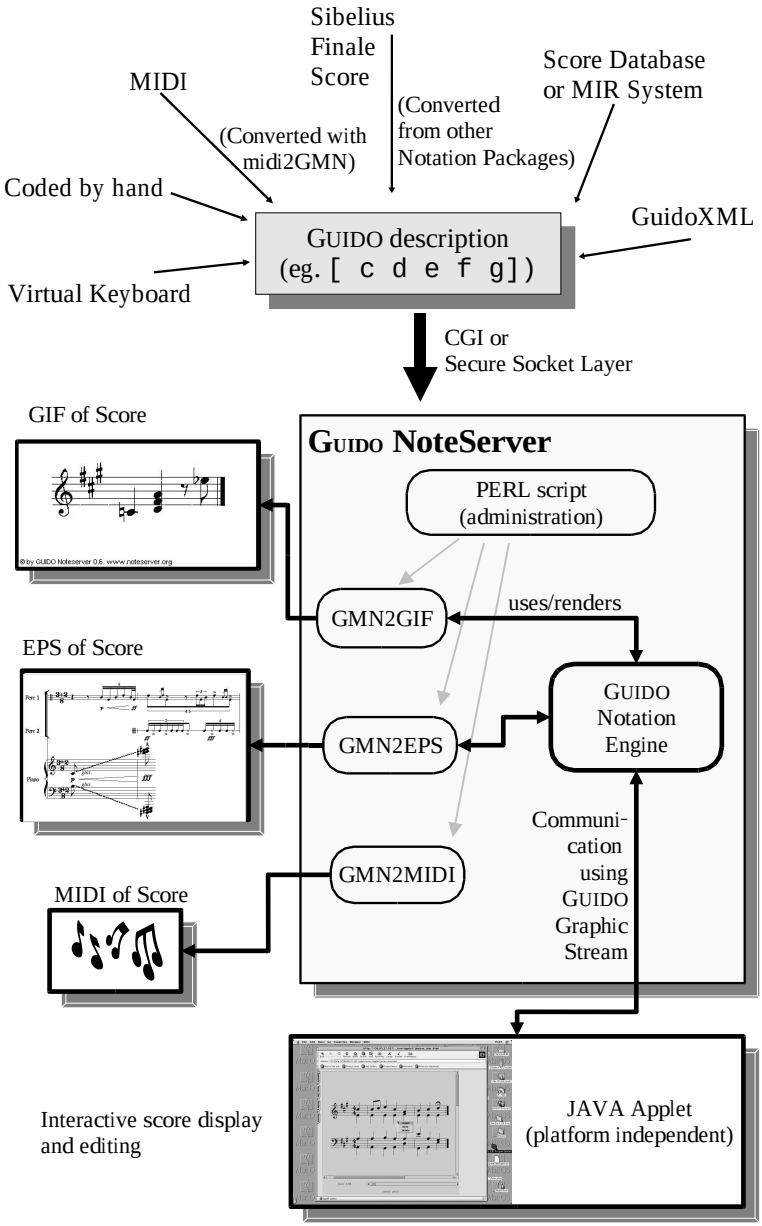


Figure 5.4: Architecture of the GUIDO NoteServer

5.2.1 Browser-Based access

The simplest access to the GUIDO NoteServer is the usage of any standard web browser. As was shown on the left of Figure 5.3, the NoteServer web page allows the entry of a GUIDO description together with additional formatting parameters (like, for example, the zoom factor). The user sends the GUIDO description by pressing the send button. Then, the image of the respective score is returned as a GIF image, which is directly shown in the browser window. It is also possible to listen to the music matching the score by playing the automatically created MIDI file. Because only standard Internet protocols are used, the browser-based access is *completely platform independent*. The text-based entry is obviously useful for creating simple scores "on-the-fly". Sometimes, scores need to be embedded within a web pages or within other applications. This can be be realized by using the CGI-based access, which will be described next. The standard browser-based access is internally implemented using CGI, but the details of this are hidden from the user.

5.2.2 CGI-Based access

A more advanced way of accessing the GUIDO NoteServer is the usage of CGI (Common Gateway Interface) [cgi02], a well defined method of interaction between clients and servers on the Internet using HTML. The NoteServer consists of three main services, all of which are realized as Perl-scripts, which can all be accessed using CGI:

- **noteserv**: this script gets a GUIDO description with additional formatting parameters and returns a complete web page including additional links (for example to the MIDI file to realize audio-output) and for accessing different pages.

- **gifserv**: this script gets a GUIDO description and additional formatting parameters and simply returns an image of the score. This service is very useful for third party applications, that do not need a complete web environment, but just want to include the score-image within their own environment.

- **midserv**: this script takes a GUIDO description and returns a MIDI file, which can then be directly played on the client computer using a standard soundcard.

CGI-based access is realized through especially formatted URLs (Universal Resource Locators), which are just the standard web addresses. In the case of the NoteServer, a typical CGI-URL looks like this:

```
http://www.noteserver.org/scripts/salieri/gifserv.pl?defpw=16.0cm&
defph=12.0cm&zoom=1.0&crop=yes&mode=gif&gmndata=%5B%20c%20d%20e%20%20%5D%0A
```

The first part is a standard web address; after the question mark, a series of name-value-pairs are separated by ampersands. In the URL above, the following variables are used:

- `defpw=16.0cm` This sets the default page width to 16 cm.
- `defph=12.0cm` This sets the default page height to 12 cm.
- `zoom=1.0` This sets the zoom factor to 1.0.
- `crop=yes` This sets the page-adjust-flag. This ensures, that only that portion of a page that contains the score is returned as the image.
- `mode=gif` This ensures, that a GIF-image is returned (and not a postscript-file).
- `gmndata=%5B%20c%20d%20e%20%20%5D%0A` This is just an encoding for the GUIDO description [c d e].

There are some additional variables, which are used in other application-contexts. The creation of CGI-URLs is greatly facilitated by Perl-routines; therefore, it is quite simple to use the CGI-based access mode. Internally, every single application using the GUIDO NoteServer is using the CGI-based-access mode.

5.2.3 Usage of JavaScript

In order to facilitate the usage of the CGI-based access and to make it possible to easily include images of music scores in web pages, some routines have been written using JavaScript [Fla98], a scripting language that can be used in web pages, which are written using HTML, and which are interpreted by the client browser. Figure 5.5 shows the HTML source code including two (relatively simple) JavaScript functions, which are used to dynamically generate the score picture shown of the right. Note that the GUIDO description is directly embedded in the HTML source code of the page: by simply changing the GUIDO description, the score being displayed changes as well. When looking at the HTML source of Figure 5.5, one aspect has to be pointed out: the GUIDO description must be syntactically changed to follow the syntax of JavaScript. Each backslash in the original GUIDO description is duplicated; otherwise, JavaScript would interpret the backslash as a control-character (similar to the C-programming language).
The following functions are provided for the use of JavaScript:

- `GetGIFURL` This function takes a GUIDO description and returns the URL that converts the given description into a GIF-file.
- `GetMIDIURL` This function takes a GUIDO description and returns the URL that converts the given description into a MIDI file.

```
<html> <head>
<title>How to embed Music Notation in WEB pages
     using JAVAScript</title>

<script language=''JavaScript''>
// this functions takes a GMN-string and returns the URL
// that converts it into a GIF-file
function GetGIFURL(gmnstring,zoom,pagenum)
{
      gmnstring = escape(gmnstring);
      gmnstring = gmnstring.replace(/\//g,"%2F");

      var string = "http://" + noteserveraddress +
        "/scripts/salieri" + versionstring +
        "/gifserv.pl?";

      if (!zoom) {  zoom = "1.0"; }
      if (!pagenum) { pagenum = "1"; }

      string = string + "pagewidth=21";
      string = string + "&pageheight=29.7";
      string = string + "&zoomfactor=" + zoom;
      string = string + "&pagesizeadjust=yes";
      string = string + "&outputformat=gif87";
      string = string + "&pagenum=" + pagenum;
      string = string + "&gmndata=" + gmnstring;

      return string;
};
// This function takes a GUIDO string, accesses the
// NoteServer (address specified as a constant above)
// and then embeds the GIF-Image in the document.
function IncludeGuidoStringAsPict(gmnstring,zoom,pagenum)
{
   if (!zoom) zoom = "";
   if (!pagenum) pagenum = "";

   document.write("<img src=" +
              GetGIFURL(gmnstring,zoom,pagenum) + ">");
};
</script>
</head>
<body>
<script>IncludeGuidoStringAsPict(
   '[ \\clef<"treble"> _/8 g g g \\bar e&/2 ]',
   '0.6');
</script>
</body> </html>
```


Figure 5.5: Using JavaScript for embedding music notation

- `IncludeGuidoStringAsPict` This function takes a GUIDO description and then creates the HTML code necessary to embed the given description as a GIF-image at the current position in the document.

- `IncludeGuidoStringAsMIDI` This function creates a hyper link to access the MIDI file that matches the given GUIDO description. The hyper link is placed at the current document position.

- `IncludeGuidoStringAsApplet` This function creates the HTML code necessary to place a Java-Scroll-Applet (which will be described in the next section) at the current location in the document.

For each one of the above mentioned JavaScript routines, there exists an additional function, that takes a GUIDO-URL instead of a GUIDO description. A GUIDO-URL is just a standard web address, which points to a valid GUIDO description. Using this mechanism it is possible to easily create a score display for GUIDO based music databases.

5.2.4 The Java Scroll Applet

Because sometimes the scores that need to be embedded in web pages become quite large, another approach using Java has been implemented. The developed Java Applet uses the GUIDO NoteServer to display the image of a score. The image is scrollable and it is further planned to realize an adjustable zoom factor. Figure 5.6 shows the Java Applet in a web page.

The shown Java Applet uses version 1.1.8 of Java, therefore, it runs on *almost any platform*.[1] The basic structure of the scroll Applet is quite simple: the CGI-based access to the NoteServer is used to retrieve a GIF-image of the score. This GIF-image is then displayed within the scrollable area of the window. Because Java offers very easy to use routines for accessing Internet content, the realization of the scroll Applet was fairly easy. Because of Java security issues, the Applet *must* be stored on the same machine as the GUIDO NoteServer. Although it is possible to adjust the security level for an Applet, it is generally very important to ensure, that only trusted Applets are allowed access to other computers on the Internet.

[1] Newer versions of Java (beginning from 1.2) do not run on Macintosh Computers with OS 9 and before. Using the older Java version ensures that the scroll applet can be used by almost all Internet users.

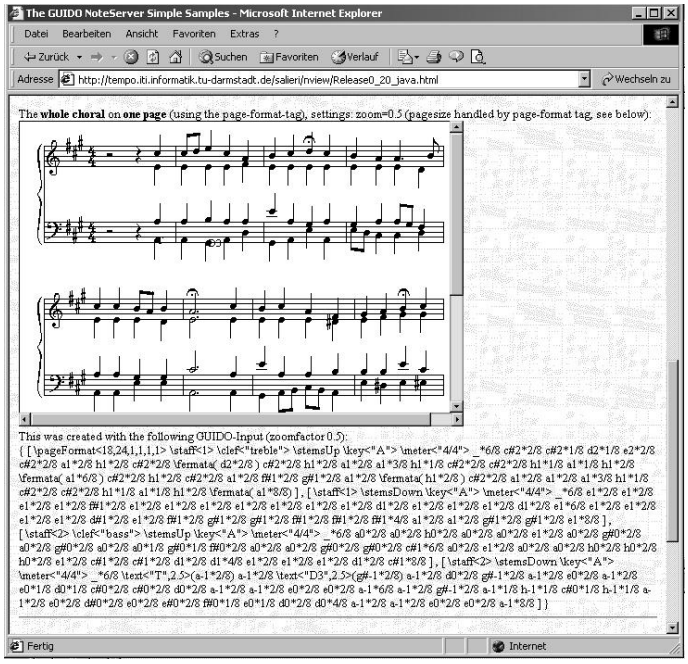


Figure 5.6: The Java Applet for score display

5.2.5 The Java Keyboard Applet

In order to facilitate the learning of GUIDO, a "virtual" keyboard was realized as a Java Applet. Figure 5.7 shows the Java Keyboard Applet in a standard browser. Whenever one of the "keys" on the virtual piano is pressed (by clicking it with the mouse), the score display and the GUIDO description below is updated. Recently, an enhancement was realized, which triggers an audio-feedback when a key is pressed.

Basically, the keyboard Applet has a similar internal structure as the scroll Applet from above. Obviously, the continuous update of the display and the GUIDO description requires some more internal logic; nevertheless, the part, which is concerned with the score display is exactly the same.

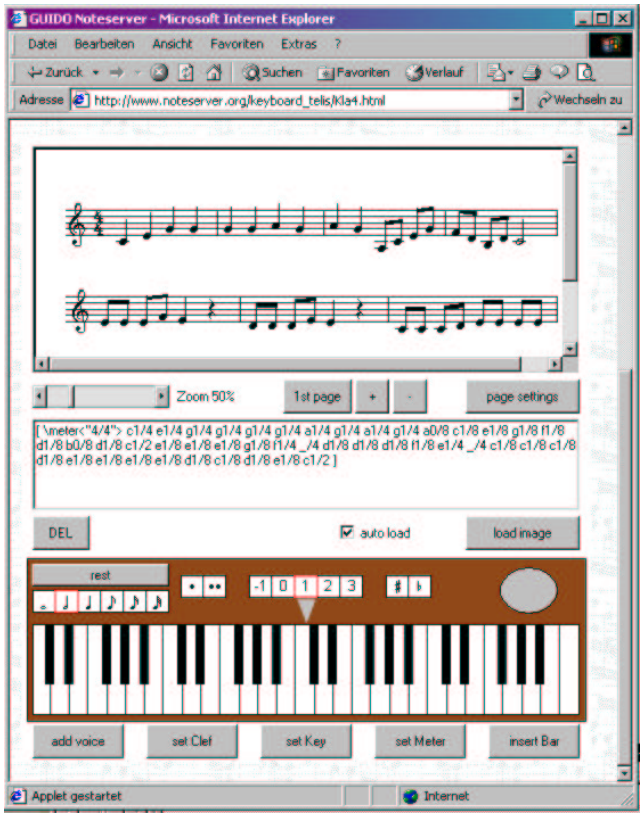


Figure 5.7: The Java Keyboard Applet

The keyboard Applet is an excellent tool for learning how to write music using GUIDO. Because both the score and the GUIDO description are shown in parallel, and because the text can be edited directly, the intuitive nature of GUIDO can be easily seen. By supplying buttons for the most common musical markup (like clef, key, and meter), the user directly learns, how tags are used within GUIDO descriptions.

5.3 Java-based Music Notation and Editing

The notation applications presented in this chapter so far are all music notation *viewers*. They are *static*, as the score notation is presented without the ability to directly edit any of the graphical elements. The displayed score can only be changed by editing the underlying GUIDO description.
In this section, the (prototypical) Java-based music notation *editor* is presented. The fundamental idea was the creation of a platform-independent notation editor that is also usable for online applications within web browsers. Figure 5.8 shows the realized Java application for displaying and editing music notation. All graphical elements shown are movable; new notes can be added.

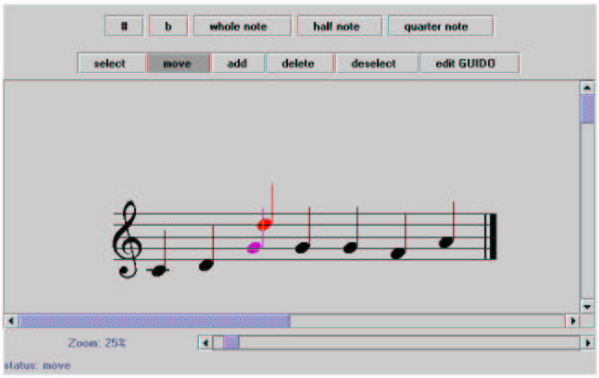


Figure 5.8: Java-based music notation and editing

The general architecture of the Java-based music notation editor is shown in Figure 5.9. The layout is a standard client-server-application; in this case, the client is the Java-based music notation editor and the server is the GUIDO Notation Engine. Communication between the client and the server is done using a newly defined protocol language, which is called GUIDO Graphic Stream. This protocol language will be described in the next subsection. Additionally, a “FontServer” has been developed: this server is needed to create images of musical symbols that are needed for score display. Because the Java version being used for the development of the editor does not allow the usage of arbitrary fonts, the FontServer had to be developed to guarantee platform independent notation. The FontServer not only creates

the image of the symbol, but also returns information on the bounding-box, which is needed for hit-testing when reacting to user input.
Two interesting features have been (partially) implemented in the notation editor: hit-testing and editing-constraints. Hit-testing is the process required when evaluating, which element needs to respond to a mouse-click. In the current implementation, a grid is used to identify those objects that might be candidates for a hit. Editing-constraints define allowed movements for musical symbols: a note head, for example can only be vertically placed directly on a staff-line or in the space in between. All other vertical positions are *not allowed*. Therefore, when moving a note head, it must be ensured that the final location is at an allowed position.

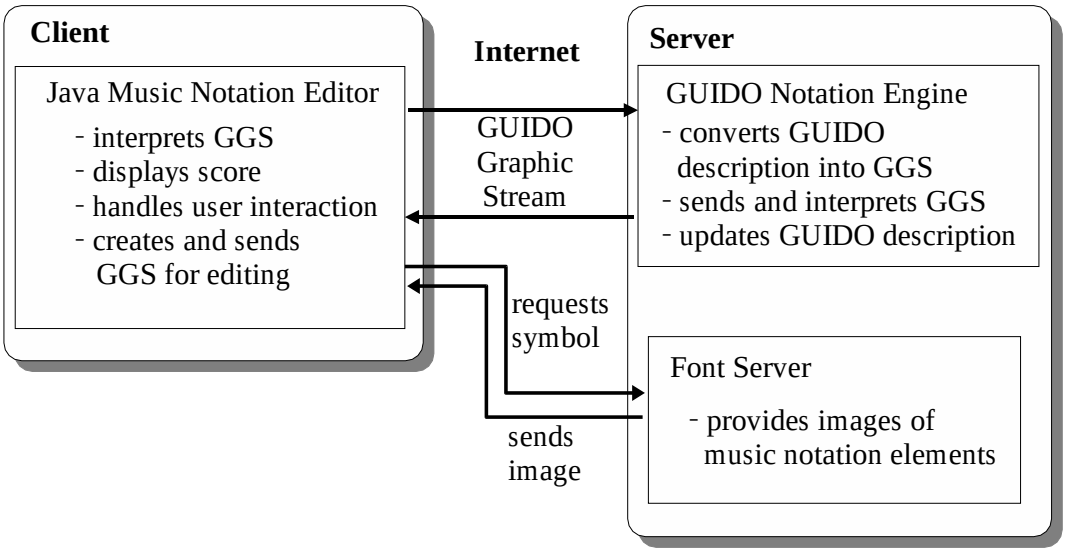


Figure 5.9: Architecture of the Java-based music notation editor

5.3.1 The GUIDO Graphic Stream Protocol

The GUIDO Graphic Stream (GGS) is a text based protocol language for describing the graphical elements of a score. GGS is not nearly as powerful as generalized graphical descriptions languages (like Postscript, or Portable Document Format). Its focus lies on a rather simple description of the position of elements visible on a score page. Even though GGS is a simple format, it can not only be used to visually describe the score, but also to transmit editing instructions. For a quick understanding of GGS consider the text and the score being shown in Figure 5.10. The individual parts of the GGS-text are linked to the score by arrows.
Each GGS-command looks like a GUIDO tag. It begins with a backslash followed by a command name and optional parameters. One very important point of the GGS is

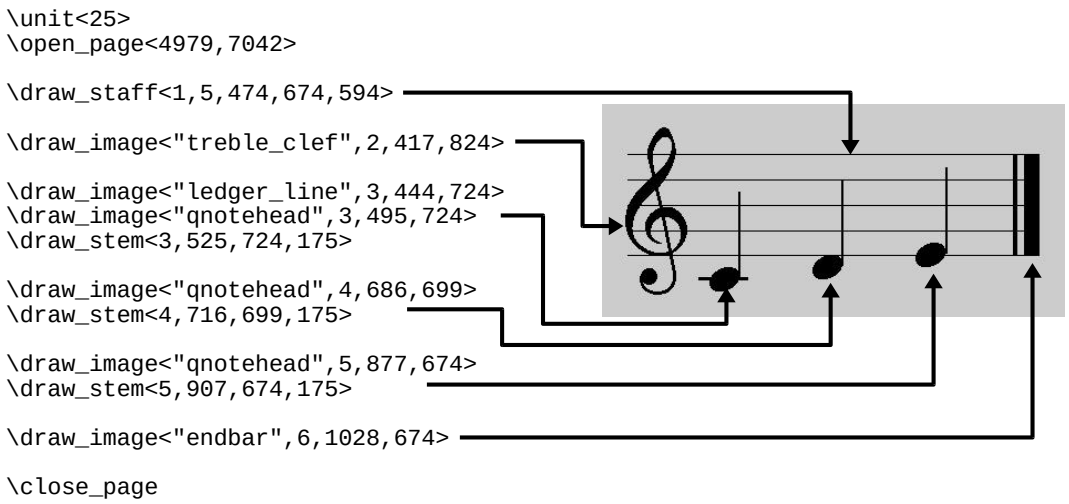


Figure 5.10: The GUIDO Graphic Stream for a score

the use of unique (integer) identifiers, which are used to group and identify individual graphical elements. Different GGS-commands may use the same id to specify that all graphical elements using this id are treated as a unit. This can be seen in Figure 5.10, where the individual notes are created from note heads and stems, which share the same id. The id is also used for sending editing instructions from the client to the server. An editing instruction might look like `\move<4,725,599>` which instructs the server to move the element with the id 4 to the given position.
Currently, only a small subset of the GGS is actually implemented in the GUIDO Notation Engine; it is planned to realize a complete GGS output for all implemented notational elements. The fully implemented GGS would directly result in a *completely platform independent* music notation viewer (and editor).
The conversion of a GGS description into another document oriented format, like (encapsulated) Postscript or PDF, is very easy, because it merely requires a one to one conversion of GGS-commands into other drawing commands.

5.4 A Musical Database System and Music Information Retrieval

GUIDO Music Notation has been used as a music representation language in the context of Music Information Retrieval (MIR) [HRG01]. MIR applications are concerned with the retrieval of musical data from musical databases. Music Information Retrieval distinguished between audio based and structure based music representation. Audio-MIR works on concrete audio files (like WAV or mp3), while structure based MIR currently mostly uses MIDI for describing music. In [HRG01]

it was shown that GUIDO is an ideal candidate for structure based MIR. The (prototypical) system, which was developed, allows the search for a musical melody (or fragments of such) in a body of scores, which are stored as GUIDO descriptions in a database. The search is specified using an enhanced form of GUIDO. If a match is found in the database, the score can be retrieved, where the location of matches are highlighted. This highlighting is made possible by an enhancement of the GUIDO NoteServer: the interface was extended to offer the possibility to mark certain fragments of a score. An example of such a highlighted score can be seen in Figure 5.11. On the left, the search phrase [g f e d] can be seen; on the right, a score is displayed, where the found search pattern is shown in red.

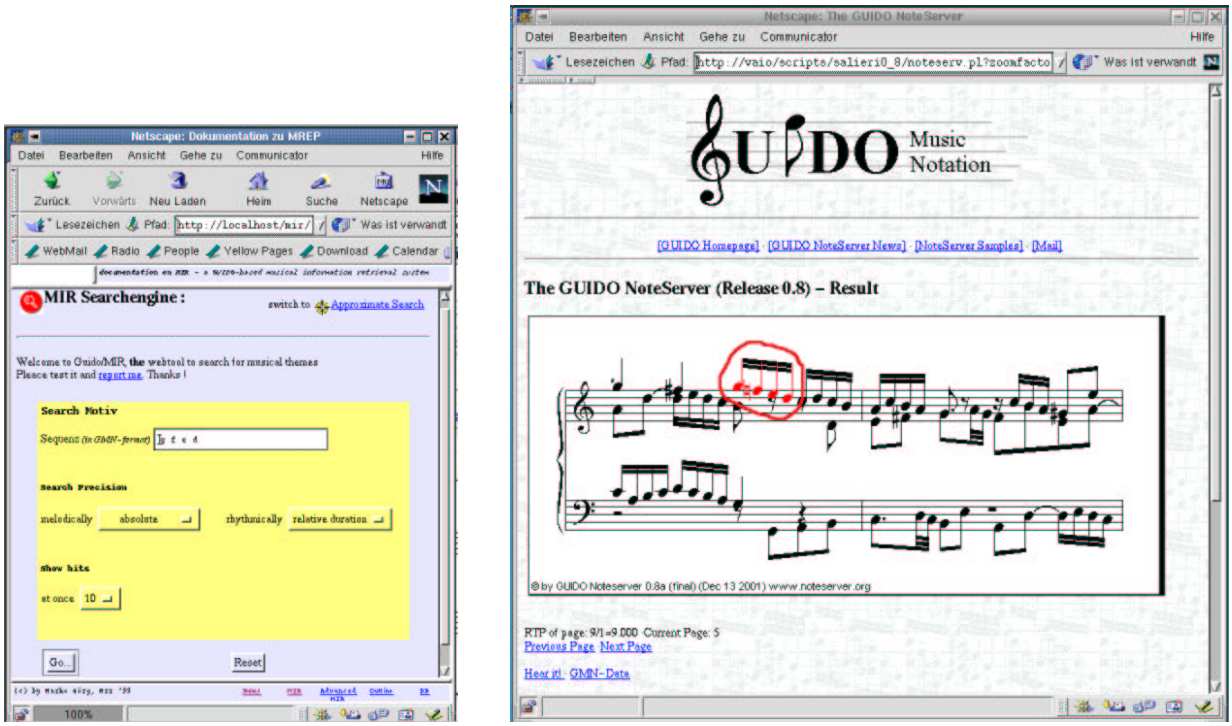


Figure 5.11: Music Information Retrieval

By using the GUIDO NoteServer to display the result of the query, all of the present features can be fully exploited. The user can listen to the music and scroll to other parts of the score. It is also possible to embed the image of the score on other web pages or in research papers.

5.5 Conclusion

The applications presented in this chapter give an overview of the variety of music notation applications made possible through the use of GUIDO Music Notation.

Even though the developed applications are used by a growing number of people around the word, it became clear during the work on this thesis that the development of a *complete* notation system is an enormous, never-ending task. In spite of the incomplete nature of the music notation system, the general idea of using GUIDO and especially the online aspect of music notation is interesting for many users. Because online music notation *and editing* has not been presented elsewhere, it seems like a fruitful endeavor to further improve the GUIDO Graphic Stream and the Java-based editor. A more complete notation editor could be easily used for online music education tools. Overall, the collection of applications proves that GUIDO Music Notation and the GUIDO Notation Engine are a solid foundation for music notation applications.

Chapter 6

Conclusion

Many music notation systems have been developed since computers are being used for music processing. While most systems are capable of automatically creating nicely formatted scores of simple music, a completely automatic system for complex music has not yet been build. The reason for that lies in the complexity of a musical score, where different, sometimes contradictory constraints have to be met. Very often when setting complex music, some typesetting rules have to be explicitly broken in order to get a reasonable result. Currently, human interaction is indispensable when producing high-quality scores.

In this thesis, the internals of a new music notation system were presented; it differs from other systems by its underlying music representation language. Using GUIDO Music Notation as the input language for the implemented music notation system created insights in the general procedures which are necessary when automatically creating conventional scores. Because GUIDO is an adequate, intuitive text-based language, the creation of musical data, represented as a GUIDO description, is fairly easy. Nevertheless, the process of automatically creating a graphical score from an arbitrary GUIDO description is far from trivial.

As GUIDO descriptions are human readable plain text, the conversion into a graphical score requires a number of steps. First, it was shown that GUIDO descriptions can be converted into a so called GUIDO Semantic Normal Form (GSNF). The definition of the GSNF was very helpful for defining an object-oriented computer-suited representation for GUIDO, which is called Abstract Representation. A GUIDO description in GSNF is converted into a one-to-one corresponding instance of the Abstract Representation. This instance is used for storing, manipulating, and traversing the musical data. The Abstract Representation has been designed and implemented in C++ using object-oriented design patterns, therefore the data structure and operations on the data are both part of the class library.

Once the GUIDO description has been converted into the one-to-one corresponding instance of the Abstract Representation, a number of music notation algorithms are executed. These music notation algorithms enhance a given GUIDO description by analyzing the input and applying common musical typesetting rules. All im-

plemented music notation algorithms are GUIDO to GUIDO transformations: they take a GUIDO description as their input and return an enhanced GUIDO description. Most of the music notation algorithms add formatting information which is automatically computed from the input using commonly accepted musical typesetting rules. As music typesetting rules are sometimes discussed contradictory in different publications, all algorithms can be exchanged quite easily.

Once the music notation algorithms have been performed on the Abstract Representation, the contained data is used to create an instance of the so called Graphical Representation, which is closely related to a graphical score. The conversion into the Graphical Representation requires the creation of graphical notational elements visible in the score. Additionally, those notational elements have to be identified that require an equal horizontally position, because they have the same onset time. A suitable way to deal with the issue is the usage of the spring-rod-model, which uses springs and rods to describe the layout of a line of music. Once the elements have been associated with springs, a line of music can be stretched to a desired extent by applying a "force" on the virtual springs. To prevent horizontal collisions, rods are used to pre-stretch the springs. The spring-rod-model has been first used for spacing lines of music in 1987 and produces quite good results, which closely match scores from human engravers. The issue of calculating spring constants for this model is not easy, and it was shown in this thesis, that it can be improved, especially if rhythmically complex interactions of voices are concerned.

After the Graphical Representation has been created from the Abstract Representation, another set of music notation algorithms needs to identify optimal line- and page-break positions. These depend on the spacing of individual lines and also on the degree of page-fill of the last page (in music, the last page of a score should be completely full). An optimal line-breaking algorithm for music based on the line breaking algorithm for TEX has been previously published by [HG87], but there has been no publication on optimally filling pages of a score. In this thesis a new optimal page fill algorithm was designed and implemented. The new algorithm is an extension of the optimal line breaking algorithm; it makes use of dynamic programming and is scalable so it can be adapted for different requirements.

The developed music notation system is used within a couple of applications, which were also implemented during the work on this thesis. The stand-alone GUIDO NoteViewer can be used to create graphical scores from GUIDO descriptions, which can also be printed. The online GUIDO NoteServer is a client-server application on the Internet that converts a GUIDO description into a picture of a score. This free service can be accessed using any standard web browser. Because GUIDO is an intuitive and powerful music representation language, it is used by a continuously growing number of people and research groups. The developed GUIDO based music notation tools are also being used and accepted widely by users around the world.

Outlook

A music notation system is never complete in the sense that every single notation feature required in any conceivable score is already implemented. As music notation is continuously evolving to represent new musical ideas, there will probably never be a single system that covers all needs.
Many features of conventional music notation have not been implemented in the presented music notation system. Some of these features can be implemented rather easily by merely using diligence (for example the addition of different note shapes). Others are quite complicated and require a lot of additional research: these include automatic collision detection and prevention. The framework of the implemented music notation provides a solid ground for further research. Additionally, the underlying music representation language GUIDO is a powerful notation interchange format. Therefore, any enhancement to the implemented music notation algorithms can be directly coded into a GUIDO description, which can then be read by any GUIDO compliant music notation software.
Another interesting topic for further research is the enhancement of the prototypical Java based music notation editor, which uses the GUIDO Graphic Stream protocol. Because Java is platform independent, this could provide the basis for a truly interchangeable music notation system. Nevertheless, a lot of additional work is necessary to create a fully functional music notation editor.

Bibliography

[ARW98] V. Abel, P. Reiss, and R. Wille. Mutabor ii - ein computergesteuertes musikinstrument zum experimentieren mit stimmungslokigen und mikrotönen. Technical Report 34, Universität Mainz, Musikinformatik und Medientechnik, 1998.

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 1957.

[Bel01] Alan Belkin. NIFF Homepage at http://www.musique.umontreal.ca/personnel/Belkin/NIFF.doc.html, 2001.

[BH91] Dorothea Blostein and Lippold Haken. Justification of printed music. *Communications of the ACM*, 34(3):88–99, March 1991.

[Blu89a] Friedrich Blume, editor. *Musik in Geschichte und Gegenwart*, volume 9, chapter Notensatz (music notation), pages 1595–1667. Deutscher Taschenbuch Verlag, Bärenreiter Verlag, Kassel, Basel, London, 1956,1989.

[Blu89b] Friedrich Blume, editor. *Musik in Geschichte und Gegenwart*, volume 5, chapter Guido von Arezzo, pages 1071–1078. Deutscher Taschenbuch Verlag, Bärenreiter Verlag, Kassel, Basel, London, 1956,1989.

[Blu89c] Friedrich Blume, editor. *Musik in Geschichte und Gegenwart*, volume 9, chapter Notendruck (music printing), pages 1667–1695. Deutscher Taschenbuch Verlag, Bärenreiter Verlag, Kassel, Basel, London, 1956,1989.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 1991.

[BSN01] P. Bellini, R. Della Santa, and P. Nesi. Automatic formatting of music sheets. In *Proceedings of the First International Conference on WEB Delivering of Music – WEDELMUSIC*, pages 170 – 177, 2001.

[Byr84] Don Byrd. *Music Notation by Computer*. PhD thesis, Department of Computer Science, Indiana University, 1984.

[cgi02] CGI: Common Gateway Interface. http://www.w3.org/CGI/, 2002.

[CGR01] Gerd Castan, Michael Good, and Perry Roland. Extensible Markup Language (XML) for Music Applications: An Introduction. In Walter B. Hewlett and Eleanor Selfridge-Field, editors, *The Virtual Score*, number 12 in Computing in Musicology, chapter 6. The MIT Press and CCARH, 2001.

[Eck00] Robert Eckstein. *XML – kurz & gut*. O' Reilly, 2000.

[eng98] Webpage on Hand-Enraving. http://www.henle.de/englisch/info/notenstich.htm, G. Henle Verlag, 1998.

[Fla98] David Flanagan. *JavaScript: The Definitive Guido, Third Edition*. O' Reilly & Associates, Cambridge, 1998.

[Fri00] Martin Friedmann. *GUIDO to MIDI Converter*. Computer Science Department, Darmstadt University of Technology, 2000.

[Fri01] Martin Friedmann. *MuseData to GUIDO Converter*. Computer Science Department, Darmstadt University of Technology, 2001.

[FS00] Martin Fowler and Kendall Scott. *UML konzentriert. Eine strukturierte Einführung in die Standard-Objektmodellierungssprache. 2., aktualisierte Auflage*. Addison-Wesley, 2000.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1994.

[Gie01] Martin Gieseking. *Code-basierte Generierung interaktiver Notengraphik*. PhD thesis, Universität Osnabrück, 2001.

[GNU] The GNU Project Webpage: http://www.gnu.org.

[Gou87] John S. Gourlay. Spacing a Line of Music. Technical report, Ohio State, 1987.

[Gra97] Cindy Grande. The Notation Interchange File Format: A Windows-Compliant Approach. In Eleanor Selfridge-Field, editor, *Beyond MIDI – The Handbook of Musical Codes*, chapter 31, pages 491–512. The MIT Press, 1997.

[Had48] Karl Hader. *Aus der Werkstatt eines Notenstechers*. Waldheim-Eberle-Verlag, Wien, 1948.

[Ham98] Keith Hamel. NOTEABILITY - A COMPREHENSIVE MUSIC NOTATION EDITOR. In *Proceedings of the 1998 International Computer Music Conference*, pages 506–509, University of Michigan, Ann Arbor, Michigan, USA, 1998. International Computer Music Association.

[HB95] Lippold Haken and Dorothea Blostein. A New Algorithm for Horizontal Spacing of Printed Music. In *Proceedings of the 1995 International Computer Music Conference*, pages 118–119, Banff Centre for the Arts, Banff, Canada, 1995. International Computer Music Association.

[Hew97] Walter B. Hewlett. MuseData: Multipurpose Representation. In Eleanor Selfridge-Field, editor, *Beyond MIDI – The Handbook of Musical Codes*, pages 402–450. The MIT Press, 1997.

[HG87] Wael A. Hegazy and John S. Gourlay. Optimal line breaking in music. Technical Report OSU-CISRC-8/87-TR33, The Ohio State University, 1987.

[HH96] Holger H. Hoos and Keith Hamel. Basic GUIDO Specification. Technical report, Darmstadt University of Technology, 1996.

[HHR99] Holger H. Hoos, Keith A. Hamel, and Kai Renz. Using Advanced GUIDO as a Notation Interchange Format. In *Proceedings of the 1999 International Computer Music Conference*, pages 395–398, Tsinghua University, Beijing, China, 1999. International Computer Music Association.

[HHRK98] Holger H. Hoos, Keith A. Hamel, Kai Renz, and Jürgen Kilian. The GUIDO Notation Format – A Novel Approach for Adequatley Representing Score-Level Music. In *Proceedings of the 1998 International Computer Music Conference*, pages 451–454, University of Michigan, Ann Arbor, Michigan, USA, 1998. International Computer Music Association.

[HHRK01] Holger H. Hoos, Keith Hamel, Kai Renz, and Jürgen Kilian. Representing Score-Level Music Using the GUIDO Music-Notation Format. In Walter B. Hewlett and Eleanor Selfridge-Field, editors, *The Virtual Score*, volume 12 of *Computing in Musicology*, chapter 5. The Center for Computer Assisted Research in the Humanities and the MIT Press, 2001.

[HKRH98] Holger H. Hoos, Jürgen Kilian, Kai Renz, and Thomas Helbich. SALIERI – A General, Interactive Computer Music System. In *Proceedings of the 1998 International Computer Music Conference*, pages

385–392, University of Michigan, Ann Arbor, Michigan, USA, 1998. International Computer Music Association.

[Hoo] Holger H. Hoos. The GUIDO Parser Kit. http://www.salieri.org/guido.

[How97] John Howard. Plaine and Easie Code: A Code for Music Bibliography. In Eleanor Selfridge-Field, editor, *Beyond MIDI – The Handbook of Musical Codes*, pages 362–372. The MIT Press, 1997.

[HR78] David Halliday and Robert Resnick. *Physics*. John Wiley & Sons, 1978.

[HRG01] Holger H. Hoos, Kai Renz, and Marko Görg. GUIDO/MIR – An Experimental Musical Information Retrieval System Based on GUIDO Music Notation. In *Proceedings of the 2nd Annual International Symposium on Music Information Retrieval (ISMIR)*, pages 41–50, 2001.

[Hub99] Matthias Huber. Entwurf eines objektorientierten Frameworks für die abstrakte Repräsentation und Visualisierung der konventionellen Musiknotation. Studienarbeit (Termpaper), Darmstadt University of Technology, 1999.

[Hur97] David Huron. Humdrum and Kern: Selective Feature Encoding. In Eleanor Selfridge-Field, editor, *Beyond MIDI – The Handbook of Musical Codes*, chapter 26, pages 375–401. The MIT Press, 1997.

[HwDCF+97] Walter B. Hewlett, Eleanor Selfridge-Field with David Cooper, Brent A. Field, Kia-Chuan Ng, and Peer Sitter. MIDI. In Eleanor Selfridge-Field, editor, *Beyond MIDI – The Handbook of Musical Codes*, chapter 2, pages 41–72. The MIT Press, 1997.

[HYT95] Hypermedia/Time-Based Structuring Language (HyTime), ISO document 10744, 1995.

[Kil99] Jürgen Kilian. MIDI2GMN – Converting MIDI to GUIDO Music Notation. Technical report, TU Darmstadt, 1999.

[Kil02] Jürgen Kilian. *Converting MIDI to GUIDO*. PhD thesis, Darmstadt University of Technology, 2002. to be published.

[Knu98] Donald E. Knuth. *Digital Typography*, chapter Breaking Paragraphs Into Lines, pages 67–155. Center for the Study of Language and Information, Leland Stanford Junior University, 1998.

[MNN99] Adrian Mariano, Han-Wen Nienjuys, and Jan Nieuwnhuizen. *Mudela 1.0.7 / LilyPond 1.1.8 Reference Manual*, January 1999.

[msd02] Articles on the document view model. http://msdn.microsoft.com, 2002.

[NN01] Han-Wen Nienjuys and Jan Nieuwenhuizen. *GNU LilyPond – The music typesetter*, March 2001. The newest tutorial and reference manual available at http://www.lilypond.org and downloaded at January, 21st, 2002.

[Pre00] Bruno R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. John Wiley & Sons, 2000.

[Rea79] Gardner Read. *Music Notation – A Manual of Modern Practice*. Taplinger Publishing Company, New York, 1979.

[Ren00] Kai Renz. Design and Implementation of a Platform Independent GUIDO Notation Engine. In *Proceedings of the 2000 International Computer Music Conference*, pages 469–472, Berlin, Germany, 2000. Berliner Kulturveranstaltungs GmbH, International Computer Music Association.

[Ren02] Kai Renz. An Improved Algorithm for Spacing a Line of Music. In *Proceedings of the ICMC 2002*, 2002.

[RH98] Kai Renz and Holger H. Hoos. A WEB-based Approach to Music Notation using GUIDO. In *Proceedings of the 1998 International Computer Music Conference*, pages 455–458, University of Michigan, Ann Arbor, Michigan, USA, 1998. International Computer Music Association.

[RH01] Kai Renz and Holger H. Hoos. WEB delivery of Music using the Guido NoteServer. In *Proceedings of the First International Conference on WEB Delivering of Music – WEDELMUSIC*, page 193. IEEE Computer Society, 2001.

[Rob98] Robert Robson. *Using STL*. Springer, 1998.

[Ros87] Ted Ross. *Teach Yourself the Art of Music Engraving & Processing*. Hansen House, Miami Beach, Florida, 1987.

[Sch97] Bill Schottstaedt. Common Music Notation. In Eleanor Selfridge-Field, editor, *Beyond MIDI – The Handbook of Musical Codes*, chapter 16, pages 217–221. The MIT Press, 1997.

[Sch98] Bill Schottstaedt. Common Music Notation, website. http://ccrma-www.stanford.edu/CCRMA/Software/cmn, 1998.

[SF97a] Eleanor Selfridge-Field, editor. *Beyond MIDI – The Handbook of Musical Codes*. The MIT Press, Cambridge, Massachusetts, London, England, 1997.

[SF97b] Eleanor Selfridge-Field. DARMS, Its Dialects, and Its Uses. In Eleanor Selfridge-Field, editor, *Beyond MIDI – The Handbook of Musical Codes*, chapter 11, pages 163–174. The MIT Press, 1997.

[SGM86] Standard Generalized Markup Language, ISO document 8879, 1986.

[SMD95] Standard Music Description Language (SMDL), Draft, ISO document 10743, 1995.

[Smi97] Leland Smith. SCORE. In Eleanor Selfridge-Field, editor, *Beyond MIDI – The Handbook of Musical Codes*, pages 252–282. The MIT Press, 1997.

[SN97] Donald Sloan and Steven R. Newcomb. HyTime and Standard Music Description Language: A Document-Description Approach. In Eleanor Selfridge-Field, editor, *Beyond MIDI – The Handbook of Musical Codes*, chapter 30, pages 469–490. The MIT Press, 1997.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1991.

[TME99] Daniel Taupin, Ross Mitchell, and Andreas Egler. *MusiXTEX– Using TEX to write polyphonic or instrumental music*, version t.93 edition, April 1999.

[vI70] Otto von Irmer, editor. *J. S. BACH – Das Wohltemperierte Klavier*, volume 1. G. Henle Verlag, München, 1970.

[Vin88] Albert C. Vinci. *Die Notenschrift – Grundlagen der traditionallen Musiknotation*. Bärenreiter-Verlag, Kassel, 1988.

[W3C02] HyperText Markup Language Home Page. World Wide Web consortium; http://www.w3c.org/Markup, 2002.

[Wan88] Helene Wanske. *Musiknotation – Von der Syntax des Notenstichs zum EDV-gesteuerten Notensatz*. B. Schott's Söhne, Mainz, 1988.

[WCS97] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programmieren mit Perl*. O' Reilly, 1997.

Complete *Advanced* GUIDO Description for Bach Sinfonia 3

The following is the complete *Advanced* GUIDO description of one page of Bach's Sinfonia 3 (BWV 789). The score page is shown in Figure 1 on page 148.

```
% The first page of a BACH Sinfonia, BWV 789
% Most of the element-positions are specified
% using Advanced GUIDO tags. The layout has
% been very closely copied from the URTEXT
% edition by the Henle-Verlag.
% This example has been prepared to show, that
% Advanced GUIDO is capable of exact score-
% formatting.
%

{ [ % general stuff
    \pageFormat<"a4",lm=0.8cm,tm=4.75cm,bm=0.1cm,rm=1.1cm>
    \title<"SINFONIA 3",pageformat="42",textformat="lc",dx=-2cm,dy=0.5cm>
    \composer<"BWV 789",dy=1.35cm,fsize=10pt>

   % indent of 1.05 cm for the first system.
     \systemFormat<dx=1.05cm>

% voice 1

    \staff<1,dy=0.85cm> % next staff is 0.85 cm below
    \staffFormat<style="5-line",size=0.9375mm>
    % barlines go through whole system
    \barFormat<style="system">
% measure 1 (voice 1)
     \clef<"treble"> \key<"D"> \meter<"C"> \stemsUp
     \restFormat<posy=-1hs> _/8
      \beam( \fingering<text="3",
                dy=8hs,dx=-0.1hs,fsize=7pt>(f#2/16) g)
      \beam( \stemsUp<5hs> a/8
             \fingering<text="2",dx=1hs,dy=11.5hs,fsize=7pt>( \stemsUp<8hs> c )
           )
      \beam( \stemsUp h1 e2/16 f#)
      \beam( \stemsUp<5hs> g/8 \stemsUp<8.5hs> h1 \stemsUp)
      \bar
% measure 2 (voice 1)
     \beam( a1/8 d2/16 e) \beam( f#/8 a1)
     \beam( \stemsUp<12hs> g/16
          \stemsUp \fingering<text="4",fsize=7pt,dy=9hs,dx=-0.2hs>(f#2) e
           \stemsUp<8hs> d \stemsUp)
     \beam( \stemsUp<12hs> c#  \stemsUp
           \fingering<text="5",fsize=7pt,dy=9hs>(h) a
           \stemsUp<8hs> g \stemsUp ) \bar
% measure 3 (voice 1)
```

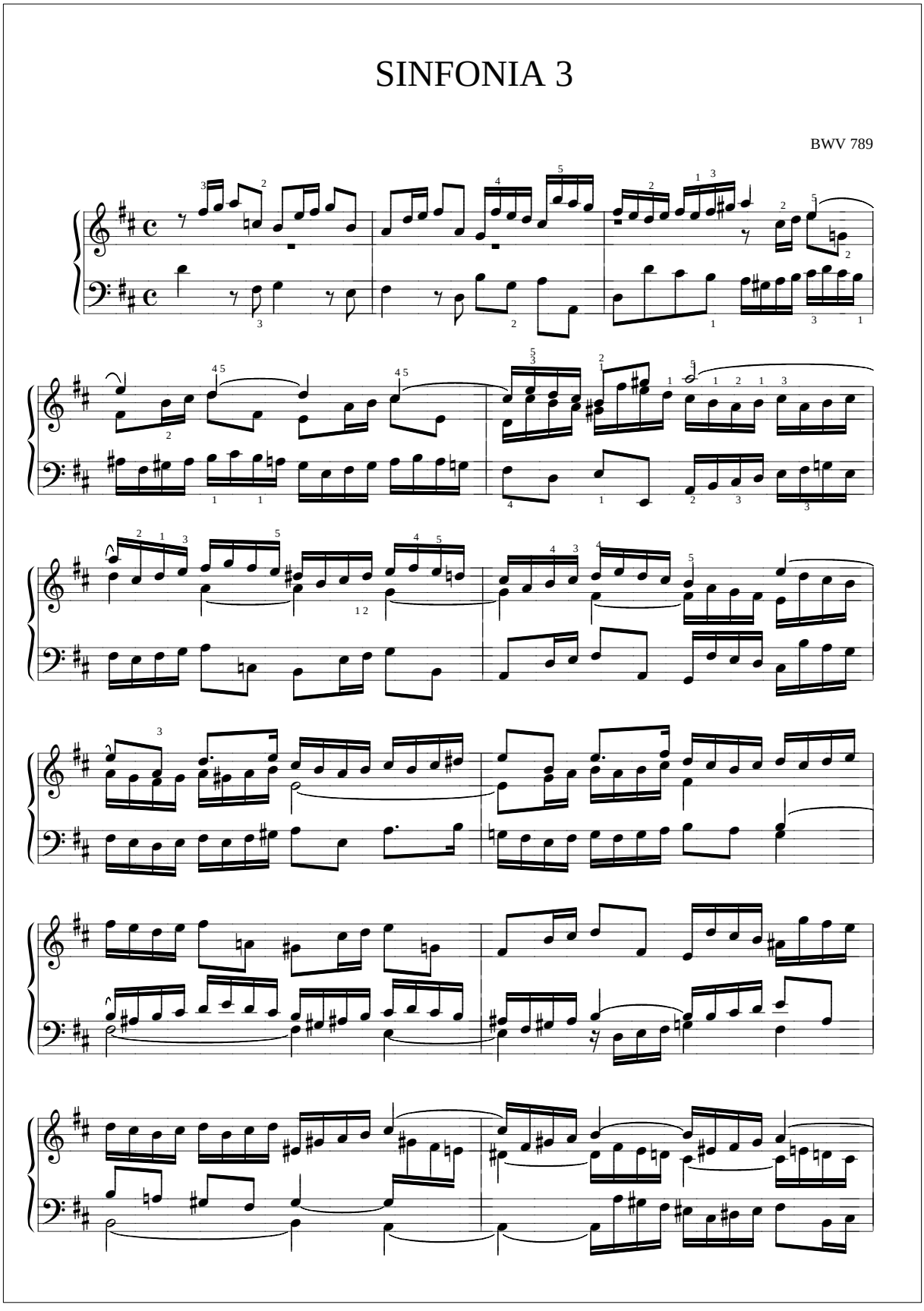


Figure 1: First page of Bach's Sinfonia 3 (BWV 789)

```
     \beam( \stemsUp<5.5hs> f#2/16 \stemsUp e
            \fingering<text="2",fsize=7pt,dy=10hs>( d ) \stemsUp<5.5hs> e
            \stemsUp)
     \beam( \stemsUp<6hs> f# \stemsUp
           \fingering<text="1",fsize=7pt,dy=11hs>(e)
           \fingering<text="3",fsize=7pt,dy=11hs>(f#)
           \stemsUp<6.5hs> g# \stemsUp)
     \stemsUp<4.75hs> a/4  \stemsUp
     \slurBegin<dx1=2hs,dy1=3hs,dx2=0hs,dy2=2hs,h=2hs>
      \fingering<text="5",fsize=7pt,dy=6hs>(  \stemsUp<4.5hs> e/4  )
      \bar \newSystem<dy=4cm>
% measure 4 (voice 1)
      \staff<1,dy=0.98cm>
          e/4 \slurEnd
         \tie<dy1=2.4hs,dx2=-1hs,dy2=2.4hs,h=1.75hs>(
         \fingering<text="4 5",dy=8hs,fsize=7pt>( \stemsUp<5hs> d) d )
         \tieBegin<dy1=2hs,dy2=2hs,dx2=0hs>
         \fingering<text="4 5",dy=8hs,fsize=7pt>(c#)
 %measure 5 (voice 1)
          \bm( \stemsUp<8hs> c#/16 \stemsUp
               \tieEnd \fingering<text="5",dy=11hs,fsize=7pt>(e) d
               \stemsUp<7.5hs> c# \stemsUp )
          \bm( \fingering<text="2",dy=12.5hs,fsize=7pt>(
                  \stemsUp<8hs> h1/8 ) \acc( \stemsUp<5hs> g#2 \stemsUp))
          \slurBegin<dy2=2hs,dx2=0hs>
          \fingering<text="5",dy=5hs,fsize=7pt>( \stemsUp<4hs> a/2)
          \newSystem<dy=4.15cm>
% measure 6 (voice 1)
          \staff<1,dy=1.08cm>
          \bm( \stemsUp<5.5hs> a/16 \stemsUp
               \slurEnd \fingering<text="2",dy=13hs,fsize=7pt>( c# )
               \fingering<text="1",dy=11.25hs,fsize=7pt>(d)
               \stemsUp<6hs> \fingering<text="3",dy=9.5hs,fsize=7pt>(e)
               \stemsUp)
          \bm( \stemsUp<7hs> f# \stemsUp g f# \stemsUp<7.5hs>
               \fingering<text="5",dy=11hs,fsize=7pt>(e) \stemsUp )
          \bm( \stemsUp<6hs> d# \stemsUp h1 c#2 \stemsUp<6hs> d# \stemsUp )
          \bm( \stemsUp<7hs> e \stemsUp \fingering<text="4",dy=9hs,fsize=7pt>(f#)
               \fingering<text="5",dy=9.5hs,fsize=7pt>(e)
               \stemsUp<7.5hs> d \stemsUp )
% measure 7 (voice 1)
         \bm( \stemsUp<6hs> c# \stemsUp a1
                 \fingering<text="4",dy=10hs,fsize=7pt>(h)
                 \stemsUp<6hs> \fingering<text="3",dy=9.25hs,fsize=7pt>(c#2)
                 \stemsUp)
         \bm( \stemsUp<7hs>
              \fingering<text="4",fsize=7pt,dy=9.25hs>(d)
              \stemsUp e d \stemsUp<7hs> c# \stemsUp )
         \stemsUp<5.5hs> \fingering<text="5",fsize=7pt,dy=8hs>(h1/4)
         \tieBegin<dy1=2.5hs,dx2=0,dy2=1.5hs,h=1.25hs>
         \stemsUp<4.5hs> e2
          \newSystem<dy=4.25cm>
% measure 8 (voice 1)
         \staff<1,dy=1.02cm>
          \bm( e/8 \tieEnd \stemsUp<6.5hs>
               \fingering<text="3",fsize=7pt,dy=12hs>(a1) \stemsUp)
                \dotFormat<dy=1hs>
           \bm( d2/8. e/16)
           \bm(c# h1 a h )
           \bm( c#2 h1 c#2 d# )
% measure 9 (voice 1)
            \bm( e/8 h1)
            \dotFormat<dy=0>
            \bm(e2/8. f#/16)
            \bm(d c# h1 c#2)
```

```
            \bm(d c# d e)
             \newSystem<dy=3.9cm>
% measure 10 (voice 1)
           \staff<1,dy=1.48cm>
                \bm( \stemsDown f# e d e)
                \bm(f#/8 \acc(a1))
                 \bm(g# c#2/16 d)
                 \bm(e/8 \acc(g1))
% measure 11 (voice 1)
                \bm( \stemsUp f# h/16 c#2)
                \bm(d/8 f#1)
                \bm( \stemsUp<13hs> e/16 \stemsUp d2 c#
                    \stemsUp<9hs> h1 \stemsUp )
                \bm( \stemsDown<5.5hs> a# \stemsDown g2 f#
                   \stemsDown<8.5hs> e \stemsDown)
               \newSystem<dy=4.45cm>
% measure 12 (voice 1)
                \staff<1,dy=1.09cm>
         \bm(d/16 c# h1 c#2 )
         \bm(d h1 c#2 d)
         \bm( \stemsUp<9.5hs> e#1 \stemsUp g# a
            \stemsUp<6hs> h \stemsUp )
         \tieBegin<dy1=3.2hs,dx2=0,dy2=3.2hs> c#2/4
% measure 13 (voice 1)
         \bm(c#/16 \tieEnd f#1 g# a)
         \tieBegin<curve="up"> h/4
         \bm( h/16 \tieEnd e# f# g#)
         \tieBegin<dx1=2hs,dy1=2.8hs,dx2=0hs,dy2=-2.1hs,h=1.1hs>
            a/4
          \newPage
% Here, the new page begins ....
% measure 14 (voice 1)
          a/16 \tieEnd
% .....
     ] ,
  [ % voice 2
    \staff<1>
% measure 1 (voice 2)
      \restFormat<posy=-8hs,dx=2.5cm> _/1
% measure 2 (voice 2)
       _/1
% measure 3 (voice 2)
      \restFormat<posy=-2hs> _/2 \restFormat<posy=-5hs> _/8
      \stemsDown \beam( \fingering<text="2",fsize=7pt,dy=6hs>(c#2/16)
              \stemsDown<7.5hs> d \stemsDown )
    \beam( \stemsDown<8hs> e/8 \stemsDown
             \stemsDown<5hs> \acc<dx=0.02cm>(
              \fingering<"2",fsize=7pt,dy=-3hs,dx=0.8hs>( g1 ) )
             \stemsDown )
% measure 4 (voice 2)
        \bm( \stemsDown<4.5hs> f#1/8 \stemsDown
                  \fingering<text="2",fsize=7pt,dy=-6.5hs>(h/16)
                    \stemsDown<6hs> c#2 \stemsDown)
       \bm( \stemsDown<7hs> d/8 \stemsDown<4.25hs> f#1 \stemsDown )
       \bm( \stemsDown<5hs> e \stemsDown a/16 \stemsDown<7hs> h \stemsDown)
       \bm( \stemsDown<7hs> c#2/8 \stemsDown<4.5hs> e1 \stemsDown )
% measure 5 (voice 2)
    \bm( \stemsDown<5hs> d/16 \stemsDown
           \fingering<text="3",fsize=7pt,dy=11hs>(c#2) h1
              \stemsDown<7hs> a \stemsDown )
     \bm( \stemsDown<6hs>
            \fingering<text="1",fsize=7pt,dy=12.25hs>( g# )
                  \stemsDown f#2 e \stemsDown<8.5hs>
                  \fingering<text="1",dy=5hs,fsize=7pt>(d)
                     \stemsDown)
```

```
     \bm( \stemsDown<8hs> c# \stemsDown
           \fingering<text="1",fsize=7pt,dy=7hs>(h1)
           \fingering<text="2",fsize=7pt,dy=8hs>( a )
           \fingering<text="1",fsize=7pt,dy=7hs>(
               \stemsDown<7.5hs> h \stemsDown) )
    \bm( \stemsDown<9hs> \fingering<text="3",dy=6hs,fsize=7pt>( c#2 )
            \stemsDown a1 h \stemsDown<8.75hs> c#2 \stemsDown )
% measure 6 (voice 2)
      d2/4 \tieBegin<dx1=0,dy1=-3.25hs,dx2=-2hs,dy2=-3.25hs,h=-1hs> a1
          \fingering<text="1 2",fsize=7pt,dx=1.3cm,dy=-4hs>(a) \tieEnd
     \tieBegin<dx1=0,dy1=-2.25hs,dx2=-2hs,dy2=-2.25hs,h=-1hs> g
% measure 7 (voice 2)
     g \tieEnd  \tieBegin<dx1=-0.5hs,dy1=-2hs,dx2=-2hs,dy2=-2hs,h=-1hs> f#
     \bm( f#/16 \tieEnd a g f# )
     \bm( e d2 c# h1 )
% measure 8 (voice 2)
     \bm( a1/16 g f# g)
     \bm( a g# a h )
     \tieBegin<dx1=0.25hs,dy1=-1.5hs,dx2=-2hs,dy2=-1.5hs,h=-2hs> e1/2
% measure 9 (voice 2)
      \bm( e/8 \tieEnd g/16 a)
     \bm( h a h c#2)
     f#1/4 \staff<2>
           \tieBegin<dx1=2hs,dy1=3hs,dx2=0hs,dy2=4hs,h=1.2hs>
           \stemsUp h0
        \bar
% measure 10 (voice 2)
        \bm(  h/16 \tieEnd a# h c#1)
          \bm(d e d c#)
          \bm(h0 g# \acc(a#) h)
          \bm(c#1 d c# h0)
% measure 11 (voice 2)
          \bm( a# f# g# a#)
          \tieBegin<curve="up"> h/4
          \bm(h/16 \tieEnd h c#1 d)
          \bm(e/8 a#0)
% measure 12  (voice 2)
          \bm( h \acc(a) )
          \bm(g# f#)
          \tieBegin g#/4
          \bm(g#/16 \tieEnd \staff<1> \stemsDown
                          \acc(g#1/16) f# \acc(e) )
% measure 13 (voice 2)
          \tieBegin<dx1=0.5hs,dy1=-1.9hs,dx2=-2hs,dy2=-1.9hs,h=-1hs> d#/4
          \bm(d#/16 \tieEnd f# e d)
          \tieBegin<dx1=0hs,dy1=-1.9hs,dx2=-2hs,dy2=-1.9hs,h=-1hs> c#/4
          \bm(c#/16 \tieEnd \acc(e) \acc(d) c#)
      ],
[ % voice 3
    \staff<2>
    \staffFormat<"5-line",size=0.9375mm>
% measure 1 (voice 3)
     \clef<"bass"> \key<"D"> \meter<"C">
      d1/4  \restFormat<posy=-3hs> _/8 \stemsDown
         \fingering<text="3",dy=-7hs,fsize=7pt>( f#0 ) g/4 _/8 e
% measure 2 (voice 3)
      f#/4 _/8 d \beam( h \fingering<text="2",fsize=7pt,dy=-8hs>(g))
           \beam(a a-1)
% measure 3 (voice 3)
     \bm( \stemsDown<7hs>d0 \stemsDown d1 c#
              \fingering<text="1",fsize=7pt,dy=-10hs>(
         \stemsDown<9hs> h0 \stemsDown ) )
   \beam( \stemsDown<8.25hs> a/16 \stemsDown g# a
       \stemsDown<9.25hs> h  \stemsDown )
    \bm( \fingering<text="3",fsize=7pt,dy=-10hs>(
```

```
              \stemsDown<8hs> c#1 \stemsDown )
             d c#
            \fingering<text="1",fsize=7pt,dy=-9hs>(
                \stemsDown<7hs> h0 \stemsDown ))
% measure 4 (voice 3)
     \bm( \stemsDown<8.25hs> a# \stemsDown f# g#
            \stemsDown<8.25hs> a# \stemsDown)
     \bm( \stemsDown<7hs> \fingering<text="1",fsize=7pt,dy=-9hs>( h )
            \stemsDown c#1 \fingering<text="1",fsize=7pt,dy=-9hs>( h0 )
            \stemsDown<6.25hs> \acc(a) \stemsDown )
     \bm( \stemsDown<9hs> g# \stemsDown e f#
            \stemsDown<9hs> g#  \stemsDown )
      \bm( \stemsDown<8hs> a \stemsDown h a
            \stemsDown<7.25hs> g \stemsDown )
% measure 5 (voice 3)
     \bm( \stemsDown<7hs>
            \fingering<text="4",fsize=7pt,dy=-7hs>(f#/8 )
                  \stemsDown<6.25hs> d \stemsDown)
    \bm( \stemsUp<6hs>
           \fingering<text="1",fsize=7pt,dy=-5hs>(e)
           \stemsUp<11hs> e-1)
    \bm( \stemsUp<8.75hs>
           \fingering<text="2",fsize=7pt,dy=-1hs>(a/16) \stemsUp h
           \fingering<text="3",fsize=7pt,dy=-3hs>(c#0 ) \stemsUp<6hs> d \stemsUp )
    \bm( \stemsDown<7hs> e \stemsDown
           \fingering<text="3",fsize=7pt,dy=-7.5hs>(f#) \acc(g)
           \stemsDown<7hs> e \stemsDown )
% measure 6 (voice 3)
        \bm(f# e f# g)
        \bm(a/8 \acc(c))
        \bm(h-1 e0/16 f#)
        \bm(g/8 h-1)
% measure 7 (voice 3)
        \bm( \stemsUp a d0/16 e)
        \bm(f#/8 a-1)
        \bm( \stemsUp<12hs> g/16 \stemsUp f#0 e \stemsUp<8hs> d )
        \bm( \stemsDown c# h a g)
% measure 8 (voice 3)
        \bm(f# e d e)
        \bm(f# e f# g#)
        \bm(a/8 e)
        \dotFormat<dy=1hs>
        \bm(a. h/16)
% measure 9 (voice 3)
        \bm( \acc(g) f# e f#)
        \bm( g f# g a)
        \bm( h/8 a)
        g/4
% measure 10 (voice 3)
        \tie<dx1=0.25hs,dy1=-1.5hs,dx2=-2hs,dy2=-1.5hs,h=-2hs>( f#/2 f#/4 )
        \tieBegin<dx1=0,dy1=-1.25hs,dx2=-2hs,dy2=-1.25hs,h=-1.5hs> e
% measure 11 (voice 3)
        e \tieEnd
        \restFormat<posy=-4.8hs>
        _/16 \bm(d e f#)
        \acc( g/4 ) f#
% measure 12 (voice 3)
        \tie<dx1=0,dy1=-2.2hs,dx2=-2hs,dy2=-2.2hs,h=-2hs>( h-1/2 h/4)
        \tieBegin<dx1=0.2hs,dy1=-1.9hs,dx2=-2hs,dy2=-1.9hs,h=-2hs> a/4
% measure 13 (voice 3)
        a/4 \bm( a/16 \tieEnd a0 g# f#)
        \bm(e# c# d# e#)
        \bm(f#/8 h-1/16 c#0)
% measure 14 (voice 3)
] }
```

Curriculum vitae

1970	born in Darmstadt, Germany
1976-1980	Schillerschule, Darmstadt
1980-1983	Lichtenberschule, Gymnasium, Darmstadt
1983-1987	Georg-Büchner-Schule, Gymnasium, Darmstadt
1987-1988	Northridge Highschool, Middlebury, Indiana, USA
1988-1990	Lichtenbergschule, Gymnasium, Darmstadt Allgemeine Hochschulreife
1990-1991	Städtische Kliniken Darmstadt, Zivildienst
1991-1993	Technische Universität Darmstadt Studium der Physik, Abschluss Vordiplom
1993-1997	Technische Universität Darmstadt Studium der Informatik, Abschluss Diplom
1997-2002	Technische Universität Darmstadt Fachgebiet Automatentheorie und Formale Sprachen Wissenschaftlicher Mitarbeiter

Erklärung

Hiermit erkläre ich, die vorliegende Arbeit zur Erlangung des akademischen Grades Dr.-Ing. mit dem Titel “Algorithms and Data Structures for a Music Notation System based on GUIDO Music Notation” selbstständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Ich habe bisher noch keine Promotionsversuche unternommen.

Darmstadt, 26. August 2002, Kai Renz, geb. Flade