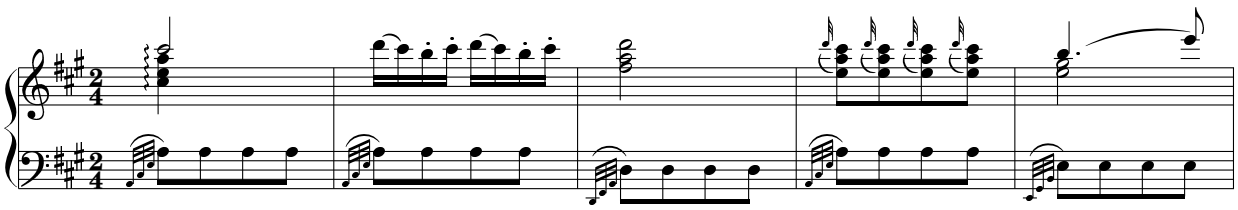
TRANSFORMING XML INTO MUSIC NOTATION



TRANSFORMING XML INTO MUSIC NOTATION

A Thesis
in TCC402

Presented to

The Faculty of the
School of Engineering and Applied Science
University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Computer Science

by

Baron Schwartz

April 10, 2003

On my honor as a University student, on this assignment I have neither given nor received unauthorized aid as defined by the Honor Guidelines for Papers in TCC Courses.

Approved _____ (Technical Advisor)
Worthy Martin

Approved _____ (TCC Advisor)
Betsy Mendelsohn

Preface

Several years ago, I was trying to notate some music to send to my brother. I thought that a search of the Internet would find a free notation editor, but there were no good free programs for creating notation. There were several for displaying it, but they made me install all sorts of things on my computer (and reboot — I had not yet discovered GNU/Linux), and did not do what I wanted. Frustrated, I searched for music markup languages and found nothing. I could not understand why no one had met this seemingly obvious need.

I quickly had the idea to create my own XML format, so I sent a message to my advisor. His reply directed me to Perry Roland, and I learned that my initial impression — "no one has done this before" — was the opposite of the truth. As my interest turned into a thesis project, my research has impressed upon me what a difficult problem this is, and how much better off we will all be when it is finally solved.

This thesis is based on the MEI format as of December 2002. Much has changed since then, but I needed to ignore it in order to finish my work. I apologize for any information that is out of date or inaccurate.

I owe a debt of gratitude to Perry Roland and Worthy Martin for their help and time. I am fortunate to work with two people so intelligent, knowledgeable, and patient. My thanks also go to Professor Peter Norton. I hope my writing reflects his teaching. A special thanks to my family, especially my brother Nathaniel, and to the friends that I haven't seen much since I started working on this project, including Matt, Kris, Dana, and Wendy and Ian.

Baron Schwartz

March 2003

Contents

List of Figures

Acronyms

API Application Programmer Interface. A defined interface for interacting with some program from another program.

CMN Common Musical Notation.

CSS Cascading Style Sheets. A visual formatting language for HTML and other languages.

DARMS Digital Alternate Representation of Musical Scores. A file format for representing musical scores.

DOM Document Object Model. One way to manipulate XML files.

DTD Document Type Definition. A structural definition of an XML language such as MEI.

EPS Encapsulated PostScript. A "bounded" PostScript image.

GNU GNU's Not Unix. Free, high-quality software.

HTML Hypertext Markup Language. The most common markup langage for Web pages.

MEI Music Encoding Initiative. An XML format for encoding musical information.

MIDI Musical Instrument Digital Interface. A common interchange format for musical information.

PDF Portable Document Format. A universal document format.

SAX Simple API for XML. An alternative to the DOM.

SGML Standardized General Markup Language. A general markup language that is the parent of XML, HTML, and many other languages.

SMDL Standard Music Description Language. An SGML language for describing music.

SQL Structured Query Language. A language for querying relational databases.

SVG Scalable Vector Graphics. A language for defining graphics, often called "the XML version of PostScript."

TEI Text Encoding Initiative. An encoding framework for texts.

URL Uniform Resource Locator. An electronic "address" for a resource such as a file.

WC3 World Wide Web Consortium. A standards body that has standardized many common Web technologies.

XHTML Extensible Hypertext Markup Language. The XML version of HTML.

XML Extensible Markup Language. A structured syntax for markup languages.

XSLT Extensible Stylesheet Language: Transformations. A language tailored for transforming XML documents into other forms.

Abstract

This thesis project tested whether an XML (Extensible Markup Language) encoding of musical data can be transformed into printed music notation. The encoding format is called MEI, the Music Encoding Initiative, and is intended to become a framework for encoding musical data to enable storage, retrieval, and transmission. Because music notation requires a superset of the information needed for most other purposes, successful transformation into music notation indicates that the MEI format represents enough information about musical data to be useful for other purposes. The thesis also analyzes the design of the MEI format, and suggests design techniques, such as variations on stylesheet languages, that may result in a more flexible, extensible format.

Part I

Thesis

Chapter 1

Introduction

This thesis project examines a new XML file format designed to encode musical data. The XML format, known as MEI,[1] is intended to support generic information retrieval and usage. The thesis research demonstrates the format's feasibility and suggests directions for further research.

1.1 Background

Computers are natural tools for manipulating musical information. For example, there is software for composing scores and tablature, creating MIDI files, analyzing music, and cataloging, indexing, and searching libraries of works. These uses all imply storage and retrieval of musical information. Some tools use databases to store musical information, but most use files. Unfortunately, files created for one purpose may not be usable for others. For instance, a file that stores bibliographical information is probably not useful for reproducing printed notation faithfully. The files may be incompatible even when the uses are similar; one format for printed scores is unlikely to be readable by another program. This is overwhelmingly true of the many file formats that exist. Because each fails to address some need adequately, and because the formats are not designed to be extensible, programmers have reinvented the wheel dozens of times, leading to vast bodies of computer-encoded knowledge that cannot be shared effectively. It is often possible to translate between formats, but unless the data is trivial, some information is usually lost.

[1] MEI stands for Musical Encoding Initiative and is pronounced "may." The format was formerly known as MDL, or Music Description Language. See [32] for details.

The need for a universal format is self-evident; clearly, one would like to encode data once and use it many times for many purposes, as shown in Figure 1.1. However, this does not describe a universal file format completely. Any such format must at least address the majority of its users' needs "out of the box," be easy to use without modification, be easy for software developers to work with, and be extensible to other purposes easily. Perry Roland, a researcher at the University of Virginia's Digital Library project, intends to develop MEI into such a format. MEI uses XML, a universal data-interchange syntax, to define a musical-data encoding. MEI's ability to store musical information in a universally accessible way is mostly untested.

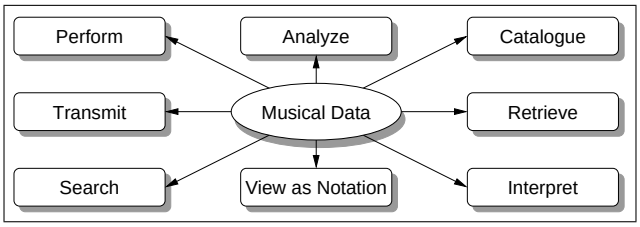


Figure 1.1: Musical data should be usable for many purposes. In this example, the oval represents the encoded musical data; rectangles represent uses of that data. The arrows indicate that universally encoded data can be used for many purposes.

1.2 Problem Definition, Scope, and Success Criteria

The thesis proposal identified two areas of research:

- Can Perry Roland's MEI format be used to generate printed music notation?
- How is it possible to separate information about musical content from information about its presentation?

The first question is a feasibility test for a subset of possible uses. In practice, notation is a very important use for musical data; most existing file formats are designed to encode music notation. Simply transforming a MEI file into printed music, which is fairly straightforward with a special scripting language called XSLT, can answer the question. It is easy to process the resulting text file into the industry-standard Postscript format with the Mup music publication program, creating high-quality printable files.[2] This demonstrates that MEI files can encode enough information

[2]Adobe PostScript is a stack-based page description language that describes shapes with arbitrary precision.

about music to generate a printed view of it.

The second question is beyond the scope of an undergraduate thesis, but specific issues are small enough to address. These include defining "content and presentation" as it applies to MEI, suggesting a standard means of extending the MEI format, and because music notation is such an important use of MEI, suggesting a way to think about music notation's visual layout. The intent is to avoid mistakes that have kept other formats from being universally useful.

The success criteria for the research are informal. To answer the first question above, it is only necessary to demonstrate that a MEI-to-PostScript transformation is possible. The second question is too large to answer fully, so I chose to define the question more clearly, and include additional thoughts as appendices to this thesis.

1.3 Rationale

This project helps solve an important, difficult problem. It is important because the lack of a universal musical data format limits the use of musical data. It is difficult because of the size and complexity of the problem.

Musicians, software developers, publishers, and researchers continue to waste a great deal of time, money, and effort on formats that cannot be used effectively for anything but the purposes for which they were designed. In the end, the user bears the costs. The lack of a good way to exchange music over the Internet, for example, means that people often purchase it by mail. Customers can purchase some music in Adobe's Portable Document Format (PDF), but these files can only be viewed and printed. There is no way to embed music notation in a web page or view it in a browser. Plug-ins exist for some browsers, but they are not widely supported, and the user is forced to download and install the plug-in, which uses a proprietary format itself [59, 53].

Not only does the user pay too much for what he or she can do, but many things are impossible to do. There are no truly excellent programs for writing music notation at a reasonable price, for example. Thus, the end user has for decades been paying too much for too little functionality.

The problem is difficult because music is complicated [51], and because there is a great deal of it to encode. One of the first bibliographical projects, RISM, identified the need to catalog 1.5 million works more than 50 years ago [10, p. 318].

MEI is one step towards a good format. Once the format exists, developers can use it to create the tools. This project helps develop the MEI format by showing that it is useful for a single purpose, that of creating printed music notation.

1.4 Social Impact

This thesis research will impact society by either encouraging or discouraging development and adoption of the MEI format. In the short term, it will only impact academics and researchers, but in the long term it could affect many organizations, corporations, and individuals.

Because of the academic nature of the MEI project at this point, this project will only affect researchers working on this and similar projects. Given that the MEI project and format are new and experimental, very few people are involved with it at present, and only Mr. Roland has invested significantly in it. This limited deployment and involvement constrains the format's impact in the short term, and by extension, this project's impact.

Assuming that the format is developed actively and eventually adopted as a standard, this project could eventually impact more of the research and development community that centers around musical information storage, retrieval, and analysis. It could also affect developers interested in writing applications for interactively creating music notation, if they consider using MEI — either as a file format for storing generated notation files, or as an interchange format. This project could also affect users and developers of other programs if MEI becomes a common interchange format.

Any format that emerges as a standard will have great social and financial impact. The ability to store musical data in a single type of file and use it for any purpose would revolutionize the music industry and academia. Many existing tools would need to support such a standard. People would use such music files in entirely different ways; for example, the ability to transmit them easily across the Internet would change the way people buy and use sheet music. This could impact music publishers heavily, possibly forcing a change in business models. In 1999, music publishing revenues reached $6.57 billion; reprint and sale of printed music constituted a total of over $711 million, or almost 11% [38, p. 3–8]. Such a large industry could be very sensitive to a technological change such as MEI. In the best case, however, the effects could be positive, leading to reduced costs, greater ease of use, less redundant data, and even uses that are currently not possible because

there is no standard way to encode music.

I anticipate that these changes will be very slow if they happen. There is too much music encoded in too many other formats; converting such music to a universal standard will be a long, painful process. Rather, if a standard emerges, it will do so slowly and in the face of great resistance from publishers, who stand to benefit from keeping proprietary standards in place. Although sheet music sales have, with the advent of recordings, become a secondary source of revenue, they remain important because sheet music for any given song may be distributed in many different arrangements and in different markets [4, p. 47]. In fact, one sheet company distributed an award-winning song in versions for piano and voice, piano solo, easy piano, and many other formats, for a total of 13 different versions [4, p. 203]. This market might even expand once moved online.

1.5 Overview of Contents

This thesis first introduces the reader to the concepts and technologies needed to understand the project. These include a review of relevant literature, music and music notation, XML and related technologies, and Perry Roland's Music Encoding Initiative. The thesis then presents the project's research method and results, and concludes by analyzing the results and presenting suggestions for further work. Appendices present the project's findings in more detail, analyze additional technologies, and elaborate on recommendations.

Chapter 2

Literature Review

This chapter introduces some of the existing musical file format standards and prior research in the field. Since people have used so many file formats for so long, it is important to understand them for several reasons. First, none of them is adequate for every possible need, or even for most needs. All have some flaw or fail to address some need. This has prevented people from adopting any single format, and forced people to create new languages. A universal format must succeed where all these languages have failed. It must also duplicate all the successes of these languages. Finally, much software exists to work with most of these languages, and compatibility is an issue to keep in mind when designing or evaluating a new format.

2.1 Common Musical Notation and Musical Complexity

Because common musical notation is the most common means of notating music graphically, many file formats are designed to encode notation itself rather than some other aspect of music. This presents unique challenges and creates some constraints on the format's usefulness.

Common musical notation is a highly complex system of symbols that evolved over centuries. Selfridge-Field states that it has survived partly because of its flexibility and ability to communicate a composer's intent [51, p. 4]. CMN is complex because music is complex; even the most basic music is difficult to describe. According to Selfridge-Field, music exists in several contexts: the sound context, the notation or graphical context, the rational or analytical context, and the gestural context — performers' gestures and the composer's directions for these gestures, motions, and

actions. Downie extends this into "seven facets, each of which plays a variety of roles in defining the MIR [Music Information Retrieval] domain. These facets are the pitch, temporal, harmonic, timbral, editorial, textual, and bibliographic facets" [10, p. 297]. Music's attributes (pitch, duration, tempo, dynamic level, articulation, parts, timbral definition, and orientation) are often complex themselves. Duration, for example, can be either relative or absolute. Musical notation represents this complexity ingeniously: "graphical features have been used very cleverly to convey aspects of sound that must be accommodated in the realization of a single note" [51, p. 7–11].

CMN's long history and oral tradition have made it imprecise. For example, grace notes are interpreted in various ways.[1] In theory, they have no time value, but in practice they must either borrow their time value from the note following, or steal it from the one preceding — yet a computer cannot steal time that has already passed [51, p. 18–19]. Grace notes and other features of music notation may require either looking into the future or making multiple passes over a piece; many early formats attempted to design a single-pass encoding, which is one reason they failed. In general, it is very difficult to encode CMN with computers.

2.2 Computer Representations of Music and Musical Notation

There are many formats for representing musical data digitally, so this thesis will only examine some of the more important ones. The following paragraphs explore various formats, comparing and contrasting their uses, similarities and differences, and strengths and weaknesses. Since many formats solve similar problems in similar ways, some formats are discussed together.

The first important computer music language was DARMS, the Digital Alternate Representation of Music, which is very powerful and has been extended into several dialects. Stefan Bauer-Mengelberg and others developed it in 1963, and it is now viewed by many as the most mature and complete way to represent musical notation digitally [52, p. 163]. It is limited in that it is designed for creating printed notation. For example, it does not even record pitch explicitly. This makes it very difficult to extract information about the music itself from a DARMS file.

MusiCopy, a project at the Ohio State University, produced important advances in musical

[1] A grace note, or a group of grace notes (called a groupetto) is usually rendered in a smaller size than 'regular' notes, and is sometimes connected to the following 'regular' note by a slur.

notation in the 1980s. The MusiCopy researchers wanted to enable musical typesetting by computer, just as with textual typesetting. Building on the work of Donald Knuth's TEX typesetting program, the MusiCopy researchers investigated problems such as line breaking, which is difficult because inter-note spacings cannot follow a simple linear rule as for text. When slightly modified, Knuth's model can handle all of the choices but the spacing by assigning penalties for badly broken lines and minimizing the total cost of the penalized lines. TEX's algorithm assumes that the last line of a paragraph does not have to be full. However, the last line of music must be full, which requires extra processing [16, p. 2–4]. MusiCopy also investigated how to slant beams between notes while maintaining readability and avoiding interference. Non-interference "is the idea that most characters and signs in music be kept within the staff as much as possible" [56, p. 2]. The results are impressive; in the case of note spacing, "the results of the algorithm have been compared with good manual spacing of music, and the two are almost indistinguishable in the majority of cases" [15, p. 1].

Mup appears to be based on MusiCopy. Mup is both a descriptive language and a program to process the encoded music into finished musical notation. It produces Postscript, the standard file format for high-quality press-ready printing. Mup bears out the MusiCopy research, which concluded that "ultimately, for every character, an absolute Cartesian coordinate with respect to the page, must be provided" [43, p. 2]. PostScript indeed describes each and every mark on the page in terms of its x and y coordinates.

TEX, the de facto standard for producing high quality manuscripts such as textbooks, can also be extended to handle music notation. Several variants exist, including MuTEX, MusicTEX, and MusiXTEX[37]. The need to embed musical notation into TEX documentation, and the fact that TEX itself provides a convenient platform for high-quality typesetting, motivated these extensions.

Brook and Gould developed the Plaine and Easie Code for indexing and cataloging music. Plaine and Easie is text-based, and uses standard typewriter symbols. It is intended for meaningful bibliographical entries and card catalogs. Repertoire Internationale des Sources Musicales (RISM) uses it as the basis for a major bibliographical project, RISM Series A/II [19, p. 363–4]. It is simple and does not pretend to represent musical notation completely.

GUIDO, another text-based code, is based on the belief that "existing representations are

either too weak to encode all the required information (like, for example, MIDI), or they are too complex... its key feature is representational adequacy" [45, p. 1].

Some codes are based on programming languages. Canon is an example of a set of Lisp macros (extensions); it evolved into Arctic [8, p. 47–56]. CMN[2] is another [50, p. 218]. The TEX-based codes are based on Donald Knuth's TEX typesetting language. Other languages are based on common Unix tools; `music` is a `troff` preprocessor for printing scores. Muscript is a Perl script that produces Postscript output from text. Humdrum is a set of tools for analyzing music represented in `kern`, which "permits the representation of the bare bones of traditional Western musical notation — pitch duration, and voicing" [21, p. 377]. Humdrum uses UNIX tools and languages, such as `grep` and `awk`, to analyze music in terms of, for example, melodic accent in Gregorian chant. MuseData uses a database to represent the logical content of musical scores in a software-neutral fashion [17, p. 402].

GNU LilyPond is a free system for typesetting music. The input is plain text, and looks similar to TEX. According to the LilyPond website, "LilyPond prints beautiful sheet music. It produces music notation from a description file" [28]. NoteEdit is another free system for GNU/Linux. Its file format is similar to Mup, and it can in fact read most Mup files. It allows the user to edit notation graphically and export it to a variety of other formats, including GNU LilyPond and MusiXTEX [39].

There are also many sound-related codes. The most important is MIDI, a standard way to represent musical events. MIDI can represent pitch, volume, and other characteristics of music as sound. Many extended MIDI formats exist, but MIDI is most important as an interchange format. More codes can be translated to and from MIDI formats than any other [18, p. 69], making MIDI an important interchange format for any proposed musical encoding. Unfortunately, MIDI is only a series of instructions in the form "turn on a sound of a given pitch at the specified time for a certain duration," and is thus a very poor interchange format because it only stores data about the performance domain. The fact that it is the best existing interchange format reveals the extent to which musical data cannot be shared effectively.

Many of these encodings are procedural, in that the encoding is actually evaluated to generate

[2]CMN stands for the unfortunately named Common Music Notation, not to be confused with actual common music notation.

the final output, as opposed to declarative. Declarative languages simply store information, but procedural ones store instructions to a processor. Procedural codes include the TEX-based codes, Canon, Muscript, and GNU LilyPond.

Codes in the SGML [58] family are the most important to this project, because XML is an SGML derivative and MEI is an XML format. There is a long history of development efforts on SGML formats leading up to XML. HyTime is an application of SGML developed in the late 1980's. It defines a meta-language that was never implemented fully because of its complexity. HyTime is important to this discussion because of the time elements that it used to extend SGML, making possible SMDL, the Standard Music Description Language. SMDL is an application of HyTime [55, p. 469]. Together, HyTime and SMDL can "represent sound, notation in any graphical form, and processes of each that can be described by any mathematical formula" [55, p. 479]. SMDL itself organizes music into several domains, and places the most importance on the logical; the other domains are the visual, the gestural, and the analytical. SMDL can describe nearly anything, including alternate tunings, but "visual and analytic domains are largely undefined by the Standard" [57, p. 38], which leaves them to the implementer. This extremely general approach makes SMDL largely useless for encoding music. It attempts to include anything that could be considered music, defines such confusing terms as "contuse event," and uses unintelligible tag names for encoding music. These factors have contributed to a lack of interest in SMDL.

There are several XML encodings at present. Unfortunately, authors have repeated past mistakes in creating XML encodings. The major formats are MML, MusicML, NIFFXML, MusiXML, MusicXML, and MEI, the subject of this thesis. Some of these were short-lived or experimental, but MusicXML is being actively promoted. MusicXML attempts to be sufficient, not optimal, as an interchange format between musical notation, performance, analysis, and retrieval software [14, p. 114]. MEI attempts to represent common western music notation from the mid-seventeenth century onward and to function as an archival data format [46, p. 126]. EMNML, or Enhanced Musical Notation Markup Language, is the result of a Master's thesis in Computer Science. EMNML addresses the lack of a standard way to transmit music over the Internet, so the format is built to handle transport issues [31]. XML is inherently a good choice for storing data, but as of yet no XML format solves the problem of how to store musical data in a way that is universally

accessible.

It should now be clear that there are many major formats, and almost as many different problems they are built to solve. There are also major divisions in their approaches to encoding musical data, such as procedural vs. declarative encodings. This variety of formats, and their almost mutually incompatible goals and methods, combine with the fact that none of them is a complete solution to create a huge, complicated body of computer-encoded knowledge that is relatively useless for most purposes. The solution to this problem (a universal format) may be a simple idea, but neither the format nor its adoption will be simple.

2.3 Chapter Summary

A great deal of work has gone into developing music representation formats, especially for representing music notation, but no universally useful format exists yet. Most of the groundwork for notational needs already exists, but other domains are still largely unsupported and immature. It is possible that XML's inherent advantages for storing data can provide a basis for a universal, standardized format.

Chapter 3

Music and Music Notation

This chapter defines music both as information and as music notation,[1] and introduces music notation briefly. It is necessary to restrict the definition of "music" in order to understand the problem the MEI format attempts to solve. The reader must also understand the basics of music notation to understand how MEI files are transformed into music notation.

3.1 What is Music?

The word "music" means many different things — for example, "a song," "the idea of a song," "all music, in general," or "pitched events with durations." There are also differing opinions on what constitutes the "authoritative" version or form of a particular work. One could claim that the authoritative version of Beethoven's 9th symphony existed only in his head, while another view is that it "is" what he wrote down. A good definition of "music" is a necessity for using computers to encode music.

This thesis views music as *information* and defines it as *events* with *pitch* and a *start time* and *duration*. Depending on the piece, it may also be necessary to consider other fundamental information, such as the *lyrics*. The MEI project defines musical data as *notation* and defines music as "that which can be notated in common Western musical notation" [48].

[1] For the purposes of this thesis, "music notation" means common Western music notation.

3.2 Music Preservation

There are many reasons to preserve music. Recreating a performance is an obvious one, but one might also want to analyze the music or retrieve information for a cataloging or searching function, for example. Ideally, the composer's thoughts could be preserved, but we must settle for preserving either a performance, or instructions on how to recreate a performance.

Recordings simply preserve the sound vibrations of a particular performance,[2] but because recording is a recent invention, it has been standard practice for centuries to preserve music by leaving instructions for recreating a performance. Because the most complete and useful means of doing this is music notation [51], representing notation adequately is a minimum goal for preserving music.

3.3 Music Notation

Common music notation, or CMN, is essentially a graph of pitch against time. Time progresses from left to right, and pitch varies vertically.

Time values are based on a fractional system. Notes are named *whole note*, 𝅝, *half note*, 𝅗𝅥, and so on down to *128th notes*.[3] Different note shapes and varying numbers of flags on the stems indicate different time values; quarter notes have a stem but no flag ♩, and eighth notes have one flag on the stem ♪. 16th through 128th notes have additional flags on the stem. Placing a dot ♩. after a note increases its time value by 50%. Figure 3.1 demonstrates a variety of notes with different time values.

Figure 3.1: The time value of each group of notes adds up to one whole note.

Music is notated on a *staff*, which is usually a series of five horizontal lines. A note's pitch is indicated by the combination of its vertical position and a *clef* that identifies a particular pitch.

[2] Digital recordings do not record vibrations, but the measured amplitude of the vibrations, typically around 44,100 times every second.

[3] British musicians use a variety of other names, such as semibreve and crotchet.

Pitches are named with the letters A through G. The pitch may be raised or lowered a *half-step* by placing a ♯ or ♭ in front of it. Figure 3.2 shows a staff with a *treble clef* 𝄞, which indicates that the G above middle C is to be placed on the second line up from the bottom.[4] The staff has a *key signature* of 1♯, which means that all F notes on the staff are to be played as F♯ by default. The next item on the staff is the *time signature* $\frac{4}{4}$, which indicates the basic unit of time and how many units of time are in a *measure*. Measures are separated by vertical *bar lines* and usually contain exactly the right number of notes to add up to the time signature. In Figure 3.2, the numerator indicates that there are 4 units in a measure, and the denominator declares a quarter note to be the basic unit of time. This example contains the notes from middle C up to the C above. Each is a quarter note, which according to the time signature has the time value of one beat. Of course, it is possible to notate exactly the same music in a different time signature, as shown in Figure 3.3.

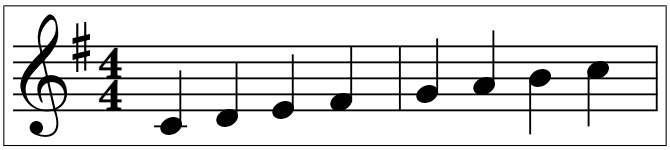


Figure 3.2: A simple example of music notation.

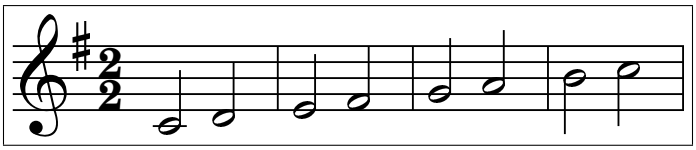


Figure 3.3: The same example in $\frac{2}{2}$ time.

Music notation is very complex; there are many shorthand notations and special cases. For instance, the flags on adjacent notes are often beamed ♫ together for clarity. It is also possible to notate a tremendous variety of information, including lyrics, performance instructions, and dynamics, as well as instrument-specific information such as guitar fingerings. Figure 3.4 on the next page demonstrates some of these features.

This brief introduction to music notation demonstrates that it is possible to record instructions for performing a piece of music, thus preserving in some sense the composer's intentions. Again, the essential representation is a graph of pitch against time; the notation shows a series of pitched events with a start time and a duration.

[4]The treble clef is a stylized G that curls around the second line on the staff.

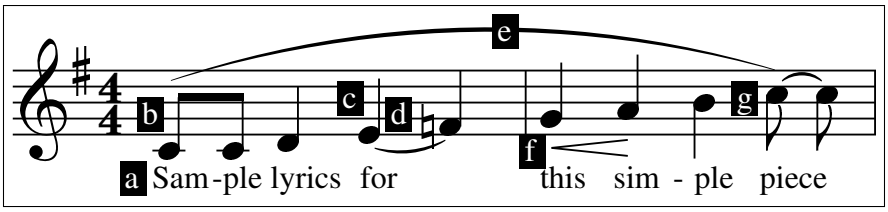


Figure 3.4: This example demonstrates more advanced notation elements. From right to left, they are: a) lyrics beneath the notes b) a beamed pair of eighth notes c) a slurred pair of quarter notes, indicating that the notes should be blended together as much as possible d) an "accidental" ♮ in front of an F, which cancels the automatic sharp implied by the key signature e) a phrase mark, indicating that all the notes are to be performed as one "phrase" (perhaps in one breath) f) a crescendo, instructing the performer to increase the volume g) two tied eighth notes, which are performed as one note with the combined time value of both.

3.4 Chapter Summary

This chapter explained that "music" means many things to many people. It discussed some goals and methods of preserving music, and introduced music notation, the most important means of preserving music briefly. A good definition of "music" is important to this project because the definition of "music" influences heavily what data are encoded in MEI files. Music notation is important because it is the desired output of the MEI feasibility testing.

Chapter 4

XML and XSLT

This chapter introduces XML and XSLT. XML is a meta-language for designing data languages, and XSLT is a programming language for transforming XML documents. XML syntax is extremely powerful for data storage because it imposes structure on the data and requires the file to have a standard syntax. XSLT is useful because it is designed to work directly with XML files. This thesis project used XSLT to transform MEI files from XML format to an intermediate format, Mup, from which printed music notation can be produced.

4.1 Overview

Since its creation in 1997, XML has become a ubiquitous tool for data representation, making it a natural choice for a universal music encoding. An XML file is a plain text file that contains both data and special markup to indicate the data's structure. XML itself is a set of syntax rules that a file must follow; additional rules are usually imposed on the data in the form of a DTD (Document Type Definition) or XML Schema. Whereas XML requires a certain syntax, the DTD or XML Schema requires the data to be of certain types and appear in certain places within the file. The combination of these two restrictions allows a computer to manipulate the file, for example viewing it as a hierarchical "tree" of data. XSLT builds another tree structure in the computer's memory by selecting data from the XML file's tree. This second tree is the XSLT transformation's output.

4.2 What is XML?

XML stands for eXtensible Markup Language:

> Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. [11]

SGML (Standardized General Markup Language) is an extremely general framework for creating markup languages. XML is a limited subset of SGML that simplifies the rules without sacrificing the ability to represent almost any conceivable kind of data.[1] Both SGML and XML are named somewhat confusingly — they are *not* languages, but are meta-languages for defining languages.

4.2.1 Syntax

XML and SGML documents contain structured plain text. Authors indicate the structure by placing special text, called *markup tags*, around text the data. For example, suppose the datum is Shakespeare's Sonnet XVIII (we will later see musical data examples, but for now a simple example is better):

> Shall I compare thee to a summer's day?
> Thou art more lovely and more temperate:
> Rough winds do shake the darling buds of May,
> And summer's lease hath all too short a date;
> ...

This could be written in XML as follows:

```
<sonnet>
   <line>Shall I compare thee to a summer's day?</line>
   <line>Thou art more lovely and more temperate:</line>
   <line>Rough winds do shake the darling buds of May,</line>
   <line>And summer's lease hath all too short a date;</line>
   ...
</sonnet>
```

[1] See [58] for a comparison of SGML and XML.

The structural delimiters are *tags*, which begin and end with angle brackets `<...>`. The text between the angle brackets contains information about the *element*; at a minimum, it names the element. An element consists of an opening tag, the element's contents, and a closing tag. Closing tags have the same name as the opening tag, but start with `</`. Elements can contain text, other elements, or a mixture of the two; elements can also be empty. The example above is a `<sonnet>` element that contains four `<line>` elements, each of which contains text.[2]

SGML permits extremely general syntax rules. For example, tags are case-insensitive, do not have to be closed, and can be *cross-nested*. XML's stricter rules remove complexity from SGML. Tags are strictly case-sensitive, elements must always be closed, and cross-nesting is illegal. Empty elements have either a closing `</element>` tag and no contents, or are written `<element />` to distinguish them from illegal unclosed elements.

For example, an HTML[3] author can indicate that text is to be rendered in bold-face or italics with `<b>` and `<i>` tags, respectively. The following is legal in HTML (see Figure 4.1):

```
Normal text <B>boldface <i>bold and italic </b> italicized only</i>
```

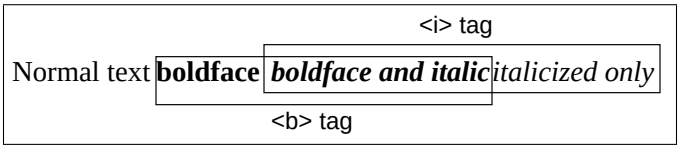


Figure 4.1: The `<b>` and `<i>` elements are cross-nested.

Note that the the elements are both mixed-case and cross-nested; the `<B>` element begins outside the `<i>` element, but ends inside it. This example could be turned into valid XML by rewriting it (see Figure 4.2):

```
Normal text <b>boldface <i>bold and italic </i> </b><i>italicized only</i>
```

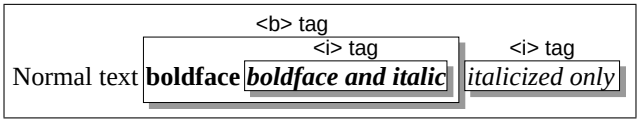


Figure 4.2: The `<i>` element can nest properly if broken in two.

XML documents begin with a *prolog* and a single top-level element that contains the rest of

[2]For clarity, this thesis will indicate an *element* by placing its name in typewriter font inside angle brackets, `<element>` but refer to the *name* of an element without the brackets, thus: `element`.

[3]HyperText Markup Language, an SGML language that is used for most Web pages. See [20] for details.

the document. The prolog identifies the file as an XML document, and optionally declares the set of rules that the document follows in a Document Type Declaration. A typical file might look like the following:

```
1 <?xml version="1.0"?>
2 <!DOCTYPE sonnet SYSTEM "sonnet.dtd" [ ]>
3 <sonnet>
4    <!-- rest of document follows -->
5 </sonnet>
```

The first line identifies this as an XML file, version 1.0. The second is a Document Type Declaration, which states *what kind* of XML document this is (in this case, a `sonnet`), and provides a URL, or Uniform Resource Locator, at which the definition of a sonnet can be found (see the next section for more on Document Type Definitions).

XML comments appear between the characters `<!--` and `-->` and are usually ignored by processing applications. XML is often indented to demonstrate visually that some elements are contained in others. The indentation is not usually meaningful to a program, however.

Tags may also contain additional information called *attributes*. The attributes are placed in the element's opening tag, and are written in the form `name="value"`. For example, a line of the sonnet could have an attribute to indicate its role in the ABABCDCDEFEFGG rhyming scheme. The first line would be written `<line letter="A">Shall I compare ...</line>`.

4.2.2 Document Type Definitions

Section 4.1 mentioned two types of constraints on an XML document. The first is the strict XML syntax, and the second is the Document Type Definition, or DTD, which imposes constraints on the file's contents.[4] Recall the DTD line from the sonnet file:

```
<!DOCTYPE sonnet SYSTEM "sonnet.dtd" [ ]>
```

A DTD is assembled from up to two parts, an internal and an external subset, either of which may be omitted. An external subset is usually contained in a file, and the internal subset is placed

[4] See Appendix F on page 81 for an introduction to XML Schema, an alternative to a DTD.

directly in the XML file between the `[ ]` characters. The internal subset can re-declare or extend the external subset. The keyword `SYSTEM` means that the external DTD subset is in a file named `sonnet.dtd`.

Using element declarations and attribute lists, the DTD specifies which elements can appear in the document, what they can contain, what attributes they can have, and the attributes' data types. An element declaration in the DTD declares the element's name and what the element contains. For example, to declare an element named `sonnet` that contains a single `author` element, a single `title` element, and one or more `line` elements, we would write the following: `<!ELEMENT sonnet (author,title,line+)>`. The `+` indicates "one or more;" there are special characters to indicate the number, order, and optionality of elements. To indicate that an element contains text, we would write `#PCDATA` in place of the element list.

Attribute lists are created similarly. To specify that the `<line>` element contains an attribute named `letter`, which may be one of the letters A through G, we would write `<!ATTLIST line letter (A|B|C|D|E|F|G) #REQUIRED>`. The vertical bars `|` indicate a choice of the listed letters, and the keyword at the end indicates that the attribute must be present in the element.

4.2.3 Well-Formed and Valid Documents

XML files must match both the XML syntax and the rules given in the DTD. There are two different notions of correctness that correspond to these constraints. One is called *well-formedness*, and means that the file is valid XML. The other constraint is validity, which can only be determined by checking that the elements and attributes conform to the DTD. Validity is a stronger constraint than well-formedness, because a valid file must also be well-formed. Not all well-formed files are valid, however; a `sonnet` file without an `<author>` element is invalid even if it is well-formed. Figure 4.3 shows this concept graphically.

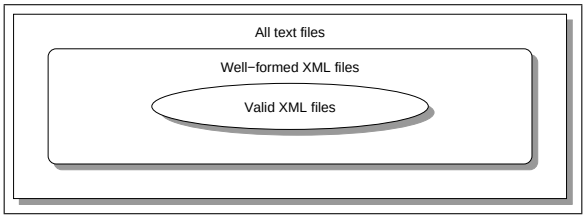


Figure 4.3: A Venn diagram illustrating that a valid XML file must be well-formed.

4.2.4 The Complete Sonnet Example

The following example demonstrates all of the features discussed above. The prolog contains the entire DTD; the optional external part is omitted. Some of the lines are omitted for brevity.

```
<?xml version="1.0"?>
<!DOCTYPE sonnet [
  <!ELEMENT sonnet (author,title,line+)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT title  (#PCDATA)>
  <!ELEMENT line   (#PCDATA)>
  <!ATTLIST line   letter (A|B|C|D|E|F|G) #REQUIRED>
]>
<sonnet>
  <author>William Shakespeare</author>
  <title>Sonnet XVIII</title>
  <line letter="A">Shall I compare thee to a summer's day?</line>
  <line letter="B">Thou art more lovely and more temperate:</line>
  <!-- Some lines omitted for brevity -->
  <line letter="G">So long as men can breathe or eyes can see,</line>
  <line letter="G">So long lives this, and this gives life to thee.</line>
</sonnet>
```

4.2.5 Document Structure

The XML specification does not define how to view an XML document's structure, but a series of related specifications define the Document Object Model, or DOM [9], which treats an XML document as a tree of objects. The specification is too large to cover here fully, but it is important to understand the tree view of a document because XSLT transformations use trees to represent XML documents [24, Chap. 2].

Since all elements in a document must be nested properly, every element is contained inside another element, except for the top-level element, which is contained in the document. Thus each element has a parent, and possibly siblings and children. The DOM views this structure as a *graph*, which is a set of *vertices* and *edges* G={V,E}. Each element becomes a vertex, or *node*, in the graph; edges connect each element to its parent. If an element contains text, it has a special text node as a child, whose value is the text itself. The top-level element in the document becomes the only child of the root node, and all other elements become descendant nodes. Figure 4.4 shows the

sonnet file as a DOM tree; it is simplified for clarity. It should now be clear why the cross-nested tags in Section 4.2.1 are a bad idea; they do not permit a tree view of the document.

All document trees in this thesis will show nodes as rounded boxes with shadows. Nodes are connected with arrows, which point from the parent element to its children. Element nodes contain the element's name in angle brackets, but text nodes contain only the text. Attributes appear as small, sharp-cornered boxes without shadows. For brevity, the root node is omitted and the outermost element is shown as the root.

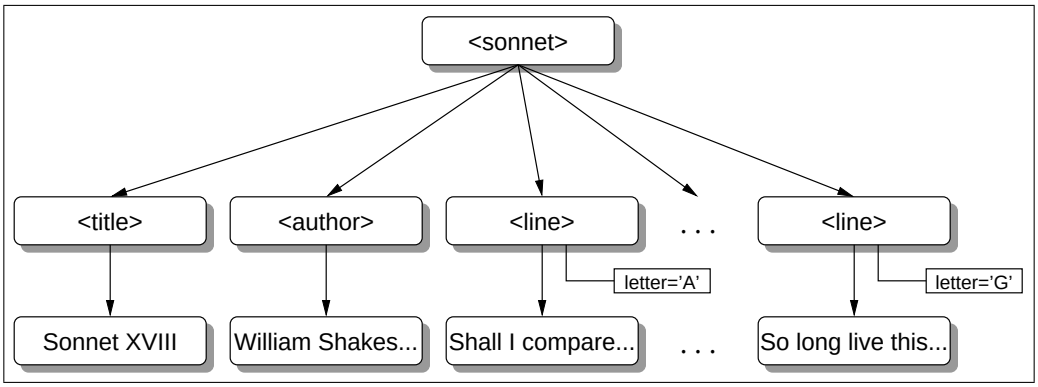


Figure 4.4: The DOM tree for the sonnet file.

4.2.6 Manipulating XML Files

XML is not designed for brevity or for ease of hand-editing [11, §1.1]. Instead, XML files are designed to be easy for computer programs to work with. There are two major ways of manipulating XML files: DOM and SAX, the Simple API for XML.[5]

The DOM [9] provides an interface for manipulating the document structure through document nodes' properties and associated functions. For example, a DOM `Node` object has a property called `previousSibling` that refers to its previous "sibling" object. Related specifications define bindings to other programming languages, document traversal methods, and other ways to relate the DOM to real-world applications.

The DOM's principal advantage is familiarity and conceptual simplicity. Most programmers find it easy to use a tree of objects with functions that can access and modify them. The principal disadvantage is computational complexity. Creating a tree from the document requires reading the

[5]API stands for Application Programmer Interface and essentially means "Interface."

file and building a structure in the computer's memory to store it. This is unacceptable for large files — and XML files can be enormous, sometimes occupying terabytes on disk. It is infeasible to read terabytes of data into memory, call a function on a node of the resulting tree, and write the entire structure back to disk, especially for minor changes to the file.

SAX [49] avoids this problem by making one pass through the file, triggering events along the way as data is encountered. Imagine a person reading a long paper tape that is scrolling under a small window. The person calls out when he sees something: "I see an opening `<author>` tag! Now I see some text, and it says `William Shakespeare`! Now I see a closing `</author>` tag!" This is analogous to how SAX works. The tape is the document, and the person is the SAX parser. SAX allows programmers to define events, such as encountering a start tag, that will call a function to handle the event.

SAX's advantage is efficiency. It only changes the sections of the file that the program modifies. SAX's disadvantages are that it is sometimes not as intuitive or convenient as the DOM, and that it is a once-through process that does not allow "rewinding."

4.2.7 Advantages and Uses of XML

XML is flexible by design. It can encode nearly any type of information; some real examples are mathematical markup language (MathML) (see Figure 4.5), scalable vector graphics (SVG) (see Figure 4.6 on the next page), financial transaction data [41], file formats for word processing and other office applications [40], and many text markup languages, such as XHTML [69], a replacement for HTML.

```
<mrow>
  <mi>x</mi>
  <mo>=</mo>
  <mfrac>
    <mrow>
      <mrow>
        <mo>-</mo>
        <mi>b</mi>
      </mrow>
      <mo>&PlusMinus;</mo>
      <msqrt>
        <mrow>
          <msup>
            <mi>b</mi>
            <mn>2</mn>
          </msup>
          <mo>-</mo>
          <mrow>
            <mn>4</mn>
            <mo>&InvisibleTimes;</mo>
            <mi>a</mi>
            <mo>&InvisibleTimes;</mo>
            <mi>c</mi>
          </mrow>
        </mrow>
      </msqrt>
    </mrow>
    <mrow>
      <mn>2</mn>
      <mo>&InvisibleTimes;</mo>
      <mi>a</mi>
    </mrow>
  </mfrac>
</mrow>
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Figure 4.5: A simple MathML example.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg width="12cm" height="4cm" viewBox="0 0 1200 400"
     xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink">
  <polygon fill="gray" stroke="black" stroke-width="10"
           points="350,75  379,161 469,161 397,215
                   423,301 350,250 277,301 303,215
                   231,161 321,161" />
  <polygon fill="gray" stroke="black" stroke-width="10"
           points="850,75  958,137.5 958,262.5
                   850,325 742,262.6 742,137.5" />
</svg>
```

Figure 4.6: A simple SVG example.

Because of its simplified syntax, it is easy to write programs that use XML. Because of its basis in computer science math, XML can in turn be used to write descriptions of programs, which are often translated into a compilable representation such as C++ or Java code. In fact, it is possible to write a programming language in XML syntax, which is the subject of the next section.

4.3 XSLT

XSLT [71] stands for eXtensible Stylesheet Language: Transformations. It is a functional programming language, written in XML syntax. An XSLT processor accepts an XML document[6] as one input, an XSLT stylesheet as another input, and transforms the document tree as specified in the XSLT. The result is another tree, which the processor usually writes back to disk.

To avoid confusion between XSLT stylesheets and the other type of stylesheet this thesis will discuss, XSLT stylesheets will be called **scripts** from this point on.

4.3.1 Mathematical Properties

XSLT is a functional language [24, p. 609]. This means that instead of a sequence of instructions and assignment statements, the programmer simply declares functions that map input data to output data. A typical script specifies functions for a variety of elements; when a processor applies it to an XML document, it transforms the data and outputs the result. These functions are called *templates*.

It is important to understand that a template is a *mapping* from one set to another, which is defined over a set of inputs (the domain). The template maps each element in the domain to an

[6]XML files are often referred to as documents, even though they may contain any type of data; the XSLT specification views files as documents.

element in the co-domain, the set of outputs. A function in this sense is a relationship between the domain and the co-domain. Given an element in the XML document (the domain), a template simply outputs the corresponding element in the co-domain. Thus an XSLT script describes a relationship between the XML document and the transformation's output.

XSLT uses some special definitions. For example, sets are mathematically unordered, but they are ordered in XSLT. This allows an XSLT script to know, for example, whether it is examining the first `<line>` element in the document. Numbering begins at 1 in an XSLT set, unlike many programming languages where the first number is 0.

4.3.2 Syntax and Language Constructs

An XSLT script contains a single top-level `<xsl:stylesheet>` element, which contains one or more templates. A script usually contains a template declared to *match* the document's root element. When the XSLT processor encounters the root element — `<sonnet>` in the running example — it applies this template to it. The top-level template usually applies other templates to the rest of the document, either explicitly or by asking the processor to search for a matching template for each element in the document.

XSLT provides a variety of constructs for controlling the behavior of the script, such as the iterative `<xsl:for-each>`, which declares a mapping to apply to each element in a set. It also provides functions (in the traditional C-programming sense of the word) to do useful things, such as return the current element's position.

XSLT is not compiled into an executable file like traditional programming languages. Instead, like many scripting languages, another program (an XSLT processor) interprets it at runtime. This means that one does not "run" an XSLT script; one executes the XSLT processor and provides the XSLT file as one of the inputs.

Because XSLT is written in XML, XSLT scripts can be used to write XSLT scripts, another simple but enormously powerful concept: "Programs that write programs are the happiest programs of all."[7] This is possible because of the property of *closure*, which means that a transformation's output is also a valid input to a transformation.

[7] Quoted by Wall[65, p. 207]; widely attributed to Andrew Hume.

4.3.3 XPath

XSLT views an XML document as a tree, just like the DOM. In fact, it relies on an XML parser to read the file and build a tree in memory before it starts to work on the transformation. It does not matter whether the parser that builds the tree is a DOM or SAX parser, but the resulting tree is manipulated in a DOM-like fashion.

XSLT relies on a related specification, XPath [70], to traverse the tree. The XPath view of a document assigns a *path* from the root to each element. UNIX users will recognize the path model: the root node is assigned a path of `/`. The path to a node is a list of the element names between the root node and the node of interest, separated by slashes. Thus, the XPath expression to specify a `<line>` node is `/sonnet/line`.

The *context* [24, p. 85] contains the set of elements within which the processor is working at a given time, in this case the `<sonnet>` node. The context is analogous to the current working directory. Many operations are defined in terms of the context. For instance, if the script specifies that all elements of a given type should be selected, it means all elements of the specified type in the current context. This is analogous to a command at the command line; for example `ls *.xml` means "list all xml files in the current directory." Similarly, when using the `position()` function mentioned above, the script returns the position in the current context. If the script is working on the 5th `<line>` element, the `position()` function will return the value 5.

4.3.4 Transformations

XSLT scripts always transform the tree representation of the document into a new tree by selecting nodes, applying templates to them, and building an output tree, which is usually written to a file. Figure 4.7 on the following page demonstrates the whole process graphically.

The following simple XSLT script transforms the sonnet example and formats it into lines:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3    <xsl:output method='text' encoding='UTF-8' />
4
5    <!-- A template to match the document's root element. -->
6    <xsl:template match="/">
```

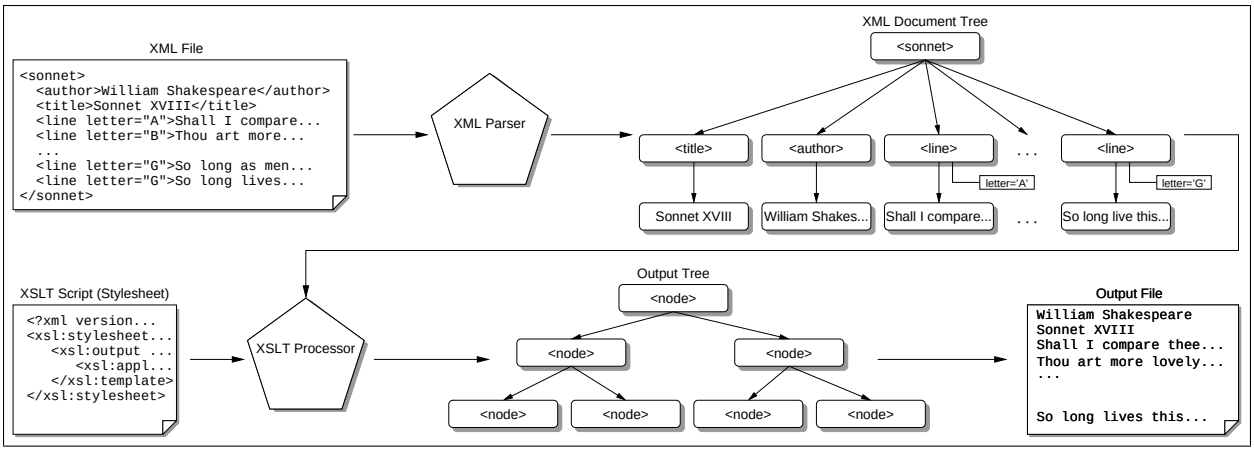


Figure 4.7: A graphical view of an XSLT transformation.

```
7        <xsl:apply-templates />
8     </xsl:template>
9
10    <xsl:template match="line">
11       <xsl:apply-templates />
12    </xsl:template>
13
14 </xsl:stylesheet>
```

Lines 1 and 2 are the standard XSLT prolog and the XSLT script's root element.[8] Line 3 is the XSLT `<xsl:output>` element, which defines the transformation's output method; common methods are text, HTML, and XML. This controls how the output tree is written to a file. Lines 6, 7, and 8 are the template that matches the root element, as specified by the `match` attribute. The `<xsl:apply-templates>` element instructs the processor to process each node in the tree, using either a matching template or the automatically provided *default template*, which simply writes the node's text value to the output tree. Lines 10, 11, and 12 define a template that the processor will use whenever it finds a `<line>` element. Again, it simply tells the processor to find and apply templates to its children.

When the XSLT processor applies the script to the sonnet file, it will first match the root element, `<sonnet>`. It will then select all nodes in the current context and apply templates to them in order. The first two, the `<author>` and `<title>` elements, have no matching templates, so the

[8]Notice that all XSLT element names are prefixed with `xsl:`. This is a *namespace* that distinguishes an XSLT element from other elements in the script; for instance, if an author wanted to create an `<output>` element in the output tree, the name would conflict with the XSLT `<output>` element. The namespace avoids these collisions.

default template will write the text value of these nodes to the output. The processor will then apply the `<line>` template to each `<line>` element. This template will apply the default template again to write the value to the output. The result is the following output:

```
William Shakespeare Sonnet XVIII
Shall I compare thee to a summer's day?
Thou art more lovely and more temperate:
...
```

The second template does nothing but output the node's value. The following modification is more interesting:

```
  <xsl:template match="line">
     <xsl:value-of select="@letter" /><xsl:text> </xsl:text>
     <xsl:value-of select="." />
  </xsl:template>
```

This introduces two new XSLT elements. The `<xsl:value-of>` element outputs the selected node's value (`@` specifies an attribute node), and the `<xsl:text>` element outputs its own value, which in this case is a single space. The modified template outputs the `letter` attribute, a space, and the current node's value (`<line>`).[9] Here is the result:

```
 William Shakespeare
 Sonnet XVIII
 A Shall I compare thee to a summer's day?
 B Thou art more lovely and more temperate:
 ...
 G So long as men can breathe or eyes can see,
 G So long lives this, and this gives life to thee.
```

XSLT provides many more language constructs, such as the ability to write conditional statements. For example, to output a `<line>` element if its `letter` attribute contains `G`, the template could be written as follows:

[9]The XPath expression "." means "the current element."

```
<xsl:template match="line">
   <xsl:if test="@letter='G'">
     <!-- perform processing here -->
   </xsl:if>
</xsl:template>
```

This can be written much more simply by matching only `<line>` elements whose `letter` attribute is `G`:

```
<xsl:template match="line[@letter='G']">
  <!-- perform processing here -->
</xsl:template>
```

4.4 Chapter Summary

This chapter introduced XML and examined some of the rules for defining XML files, using a simple file to encode a sonnet as an example. It explained the structure of an XML file and methods for manipulating this structure from a computer program. It explained some of the theory and practice of XSLT transformations, and showed a simple XSLT script (stylesheet) to transform the sonnet, with a twist to demonstrate some features of XSLT. Since musical files are relatively complex, this chapter used a sonnet as an example, but XML and XSLT are important to this thesis because the MEI format is written in XML, and XSLT is used to transform these files into other formats — in this case, into Mup, which can be transformed into printed music notation.

Chapter 5

The Music Encoding Initiative

Perry Roland, the MEI format's author, is attempting to create a Music Encoding Initiative. The goal is to replace current music encodings, which are not universally useful (see Chapter 2). The MEI attempts to do this by emulating the Text Encoding Initiative (see below). The MEI project's stated goals [48] are to

- Create a framework for the encoding of music data
- Enable content-based searching, analysis, etc.

The MEI format attempts to meet some core requirements that Huron has identified for musical encodings [22]. In addition, desirable characteristics such as extensibility are built into the DTD, which is separated into modules and defined in terms of external entities that can be changed easily. To accommodate authors, the DTD defines an "inheritance" and "propagation" model.

5.1 What is the MEI?

The MEI is both an effort to standardize computerized musical data, and a format to allow the standardized encoding itself. The MEI approach to music encoding is partially based on the TEI approach to textual encoding. From the TEI website [60],

> The TEI is an international and interdisciplinary standard that helps libraries, museums, publishers, and individual scholars represent all kinds of literary and linguistic texts for

online research and teaching, using an encoding scheme that is maximally expressive and minimally obsolescent.

The TEI provides "detailed recommendations for the encoding of all kinds of textual material of all kinds in all languages from all times." [60] Perry Roland believes that the TEI is successful because it limits its scope to those expressions that can be written [48, 47]. Similarly, to limit the project's scope and prevent an overly-ambitious or too-general encoding, Mr. Roland limits MEI to those aspects of music that can be notated in CMN [47].

MEI is partially inspired by the Mup [34] and Humdrum [23] file formats, uses international standards from the Acoustical Society of America and others (Roland, [47]), and supports the notational conventions established by well-known authorities such as Read [44].

5.2 Requirements

According to Roland [47], the MEI encoding should have the following characteristics:

Comprehensive Any non-comprehensive encoding is of limited use.

Declarative Knowledge that is declared can be analyzed; knowledge that is procedural is inaccessible.

Explicit All knowledge should be represented explicitly in the encoding.

Interpreted All encodings are interpretations of the thing encoded. This should be explicitly built into the encoding.

Hierarchical A hierarchical data structure is object-oriented, but not unnecessarily complex or restrictive.

Formal The encoding should be provably correct without reference to anything external. Incorrect practices should not be accommodated.

Flexible Minor variations must be accommodated, but not at the risk of de-standardization. Possibly irrelevant details should be made optional.

Extensible The encoding must allow for unforeseen needs.

The MEI DTD is explicit because it uses elements to mark up structure, and attributes to specify information about the structure. Attribute values are constrained, reducing the chance of sloppy markup that is valid and well-formed, yet meaningless. The encoding is hierarchical, because elements naturally result in a hierarchical document. The combination of the formal XML syntax and an explicit encoding means that the MEI format is formal. Finally, the DTD is highly flexible and extensible by design.

5.3 The MEI DTD

Because the DTD is built from primitive pieces, it contains very little redundancy and is highly extensible and easy to maintain. These are important qualities, because a successful format must be easily adaptable to changing needs and technologies. This section explains how the DTD is built and organized.

5.3.1 DTD Structure

Parameter entities are named strings of characters that can be referred to elsewhere in the DTD. For example, the string "`hello`" might be named `greeting`; the XML parser inserts "`hello`" wherever the reference to `greeting` appears. A DTD author declares an entity with the `ENTITY` keyword and refers to the name with the `%entityname;` syntax. One of the ways the MEI DTD uses entities is to load external entities from other files and use them to assemble the DTD. The DTD currently loads the following files:

Name Declarations This file uses entities to define names used in the DTD, such as element names. For example, it defines the `<note>` element's name as `<!ENTITY % n.note "note">`.

Attribute Value Lists Since many elements have common attributes, they are defined separately to reduce redundancy and make them easier to change. All attribute values are kept in one file and defined as entities; for example, possible values for a pitch name are defined as follows: `<!ENTITY % PITCHNAME '(a|b|c|d|e|f|g)'>`.

Shared Attribute and Content Models Just as attributes are shared across elements, combinations of attributes and elements may be common to many elements. The following code

declares the attributes for the logical domain of a note. Notice that it defines the `pname` attribute in terms of the `%PITCHNAME` entity defined above.

```
<!ENTITY % a.log.note
                      "acci              %ACCIDENTAL;          #IMPLIED
                      artic              %ARTICULATION;        #IMPLIED
                      dots               %AUGMENTDOT;          #IMPLIED
                      oct                %OCTAVES;             %PROPAGATED;
                      pname              %PITCHNAME;           %PROPAGATED;
                      ptab               CDATA                 #IMPLIED
                      pnum               %PITCHNUMBER;         #IMPLIED
                      tie                %TIES;                #IMPLIED">
```

Core Element Declarations This file assembles all other files into the DTD itself. Each element is defined in terms of entities. For example, the `note` element and its attribute list are defined as follows:

```
<!ENTITY % note 'INCLUDE'>
<![%note;[
<!ELEMENT %n.note;  (%n.accid;|%n.artic;)*>
<!ATTLIST %n.note;
          meiform            CDATA                 #FIXED 'note'
          %a.common;
          %a.log.event;
          %a.log.note;
          %a.vis.note;
          %a.ges.note;
          %a.anl.note;>
]]>
```

Notice that both the element's name and attributes are entities, including `%n.note` and `%a.log.note`. Only one attribute, `meiform`, is defined directly; this is typical of most MEI elements.

Miscellaneous This includes character entities and notations.

The naming convention for the entities makes it easier to understand their purpose. For example, `%n.note` means "name of note," and `%a.common` means "attributes that are common to all

elements." The resulting DTD is not easily human-readable, because it requires a great deal of cross-referencing, but is easy to maintain, because there is very little redundancy.

5.3.2 Inheritance and Propagation

The DTD defines some unconventional[1] extensions to a MEI document's semantic interpretation. For instance, the code above defines the `<note>` element's `oct` attribute to be `%PROPAGATED`, which is defined as follows:

```
1  <!--These parameter entities are used as keywords to express rules or constraints
2  which cannot be fully expressed in attribute declarations; their expansions show
3  the nearest available equivalent.-->
4  <!ENTITY % INHERITED ’#IMPLIED’> <!-- if not supplied, value inherited from an
5                                       ancestor node -->
6  <!ENTITY % ISODATE   ’CDATA’>    <!-- ISO date format -->
7  <!ENTITY % NUMBER    ’CDATA’>    <!-- one or more digits -->
8  <!ENTITY % PERCENT   ’CDATA’>
9  <!ENTITY % PROPAGATED ’#IMPLIED’><!-- if not supplied, value obtained from a
10                                      previous event in the same bar -->
```

Line 9 indicates that any attribute missing from an element, but defined as `%PROPAGATED`, should be propagated from a previous element. Similarly, line 4 defines an inheritance model, in which a missing attribute may be inherited from an ancestor. This code also defines data types that do not exist in XML, such as NUMBER. To see how a document could use these features, suppose a file contains the following fragment:

```
1  <bar n="4">
2     <staff def="s1">
3        <note pname="g" dur="4" />
4        <note pname="b" />
5        <note dur="2" />
6     </staff>
7  </bar>
```

After an XML parser reads this, the second and third `<note>` nodes have only one attribute each,[2] as shown in Figure 5.1A. Because the DTD defines the `pname` and `dur` attributes to be

[1] These extensions are unconventional for XML, but sometimes used in SGML.

[2] The "missing" attributes actually exist, but they have no value.

`%PROPAGATED`, they are "brought forward" from the preceding note as shown in Figure 5.1B. These attributes are then present on the nodes, even though they were omitted from the XML file. The `<note>` element "inherits" the `oct` attribute, because it is defined on an ancestor node.

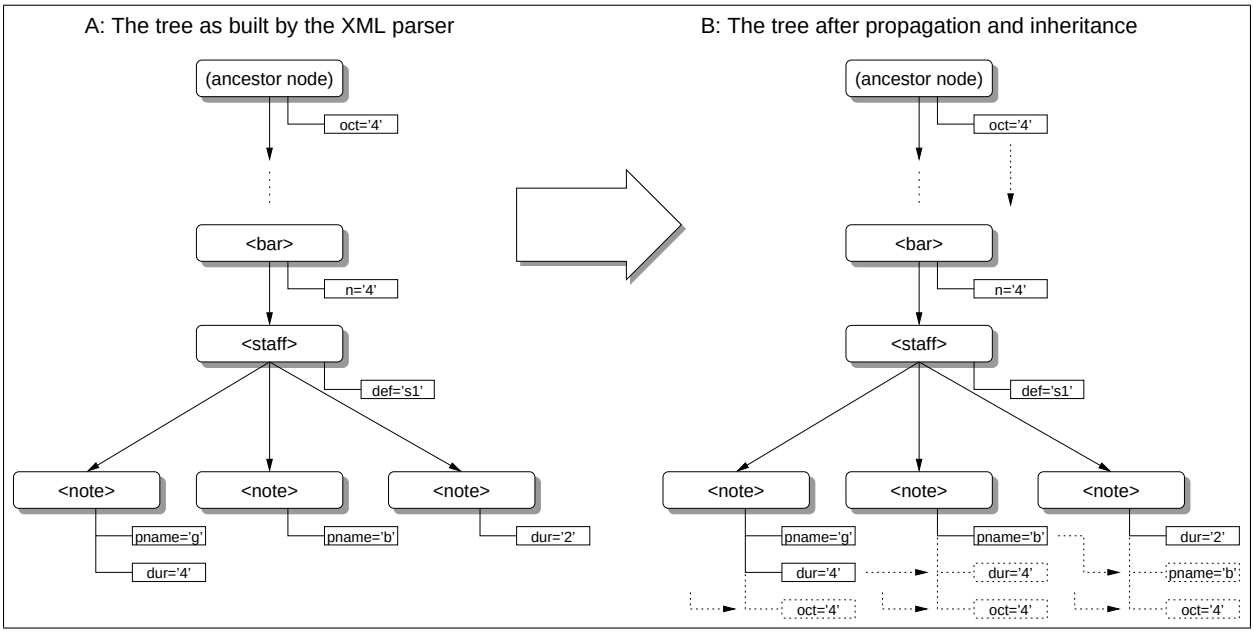


Figure 5.1: Attribute propagation and inheritance. Attributes that are not defined explicitly may be inherited from an ancestor or propagated from a previous element. The tree on the right shows inherited and propagated attributes with dotted borders. The diagram is simplified for clarity.

The inheritance and propagation features are intended to ease hand-editing of files, but a file may always be written in "canonical form,"[3] or pre-processed to bring it into canonical form if desired [48].

5.3.3 Familiar Terminology

The MEI DTD uses familiar names for elements and attributes. For example, a note is named `note`. Using common music notation terminology has the benefit of making MEI files clear. This is a significant advantage over other encodings, such as SMDL, that define an entirely new terminology. It also makes it clear that there is a correspondence between MEI-encoded data and music notation.

[3] "Canonical form" means that the file is written out fully, with all information expressed explicitly, and all allowed variations transformed into their equivalent "official" format.

5.3.4 Critical Apparatus

Critical editions of music and collections of works often contain extensive text as well as music notation, and because the MEI's goal is to be comprehensive, it must support encoding this information. For example, there might be images, a table of contents, an introduction, commentary on the music, the composer's biography, and an index. MEI defines elements to include all of this information in a single file.

5.4 A Sample MEI File

The children's song "Mary Had a Little Lamb" is the simplest example of encoded music this project used. Portions of the file are shown below; the full file can be found in Appendix I on page 112.

The file opens with a `<mei>` element, followed by a `<meihead>`. This element contains information such as the author, revision history, and other meta-data about the music.

```
<meihead>
   <meiid>MaryHadaLittleLamb</meiid>
   <filedesc>
      <pubstmt>
         <agent>Perry Roland</agent>
         <rights>Electronic edition copyright &copy; 2002 Perry Roland.
                 All rights reserved.</rights>
      </pubstmt>
   </filedesc>
</meihead>
```

The `<body>` element contains the music itself. There may be one or more `<mdiv>` (musical division) elements, each of which contains a sequential division of the work; this example has only one `<mdiv>`. Next is the `<score>`, which contains a full view of the work. The element begins with a `<scoredef>` (score definition), which records the key signature, timing, and other defaults for the score. MEI uses the `<staffdef>` element to define the staffs for the piece.

```
<body>
   <mdiv type="children's song">
```

```
<score>
   <scoredef meter.count="4" meter.unit="4" key.tonic="Eb"
   key.mode="major" key.sig="3f" beam.group="4,4,4,4">
      <staffdef id="_s1" octave.default="4" />
   </scoredef>
```

The rest of the file is a series of `<bar>` elements, each of which has a single `<staff>` element that contains notes. Each note may have a variety of attributes. The notes in this example specify the note's `pname` (pitch name) and `dur` (duration). The `dur` attribute's value is the reciprocal of the note's time value.[4] For example, a quarter ($\frac{1}{4}$) note's `dur` attribute has the value 4. After the notes are encoded, the `staff` and `bar` elements elements are closed.

```
   <bar n="1">
      <staff def="_s1">
         <note pname="g" dur="4" dots="1" />
         <note pname="f" dur="8" />
         <note pname="e" dur="4" />
         <note pname="f" dur="4" />
      </staff>
   </bar>
   <!-- The rest of the file is <bar> elements similar to the above -->
```

If the piece had more than one staff, the music for each staff would be encoded in the same bar, but enclosed in separate `<staff>` elements. This demonstrates the design decision to first break the music into bars, then encode all information in one bar together.[5] Each `<staff>` element refers to its corresponding staff definition with the `def` attribute, which matches the `<staffdef>`'s `id`. Here is a sample bar for a piece with two staffs:

```
   <bar n="2">
      <staff def="_s1">
         <note pname="g" dur="4" />
         <note pname="g" dur="4" />
         <note pname="g" dur="2" />
      </staff>
      <staff def="_s2">
```

[4] This definition of note duration emulates Mup's way of defining durations.

[5] It is also possible to encode music staff-by-staff if it is encoded with `<parts>` instead of `<score>` elements.

```
8               <note pname="g" dur="4" />
9               <note pname="g" dur="4" />
10              <note pname="g" dur="2" />
11           </staff>
12        </bar>
```

5.5 Chapter Summary

This chapter presented the MEI project and encoding format. The MEI project seeks to create a standard framework for encoding musical data. The MEI format is defined in the MEI DTD. This chapter explained the characteristics and design principles of Perry Roland's MEI music encoding, the DTD's structure and organization, the inheritance and propagation features that the encoding defines, and analyzed a simple MEI file. The following chapters will explain how these files may be transformed from MEI format into Mup and then into printed music notation, establishing the MEI format's usefulness for encoding musical information.

Chapter 6

Project Research

This project's central research activity was determining if it is possible to transform musical data encoded in MEI format into printed musical notation. I also attempted to define a good separation between musical content and presentation, but realized that separating content and presentation is not a good approach for a musical markup language like MEI.

6.1 Transformations

The project's ultimate goal was to create an XSLT stylesheet that can transform MEI-encoded files into Mup-encoded files. Once the data is in Mup format, it is easy to run the Mup program on the file and create printable PostScript. Though the MEI project's goal is to create an encoding that can be used for far more than music notation, focusing on only the notational aspect narrows the thesis project's scope. It is also easier to know when the work is done, because a human can tell easily whether the resulting notation is "good enough" by comparing it with a published version.

Because the project's goal was to show that MEI files represent the music itself, the transformation is a test of equivalency, which can be defined at several levels. To illustrate this, consider two extremes: directly representing the shapes and colors of printed music notation, and representing a label associated with the music. At the one extreme, a page description language can describe the shapes the printer should draw on the page. At the other, a program can accept the label as input and then display or print the music. Figure 6.1 on the following page shows these two extremes. In each case, the system contains a certain amount of "knowledge;" the example demonstrates that

this knowledge can be contained in the *input* or in the *system that processes the input* and produces the output. In reality, the inputs and processing applications usually share the knowledge. For example, a word processor "knows" what characters a document contains, but the font "knows" how to draw those characters.

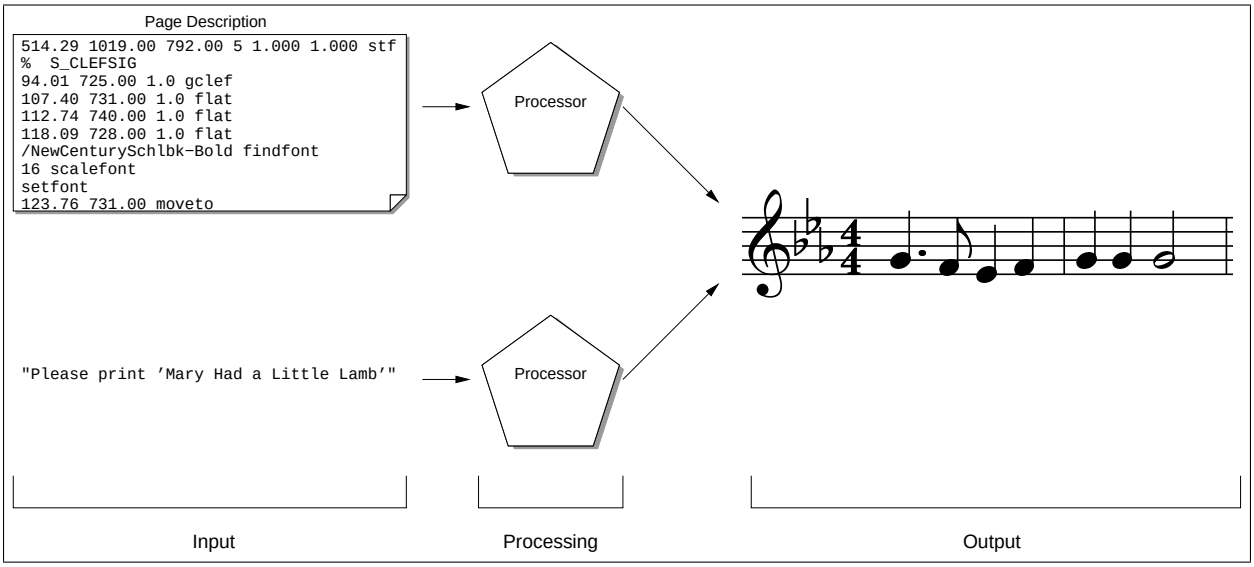


Figure 6.1: Two extremes of equivalency. In the upper example, the input file describes completely the notation (PostScript). In the lower, the input contains the instruction to produce the piece. In both cases, the processing makes the input equivalent to the output.

The XSLT script that performs the MEI-to-Mup transformation only "knows" how to transform MEI to Mup. It does not know anything about the final page layout, paper width, or the shape of a quarter note, for example. Thus it makes no visual decisions at all; it defers these to the Mup program, which "knows" these things and many more, such as what font to use for the $\frac{4}{4}$ time signature, for example. After processing with Mup, the resulting PostScript file contains almost all information about the music notation's appearance, *explicitly* encoded in the file. The printer need only draw the shapes described in the file.[1]

It should be clear that this process is a "division of labor," with some parts of the chain handling things at a very low level, and others at a high level. The chain contains all the knowledge necessary to transform between equivalent forms of the information.

Recall that XSLT is a functional programming language, and an XSLT script is a set of map-

[1]There is an exception: the fonts used in the file. Fonts themselves describe a set of shapes, so the printer has to refer to the font to know how to draw any text included in the file.

pings from the domain to the co-domain (see Section 4.3.1 on page 25). Thus an XSLT script represents a relationship between its input and its output. In this case, the XSLT is a formally defined relationship between an MEI file and a Mup file. The Mup source code and the PostScript grammar define the relationship between Mup files and printed music notation. Therefore, there is a relationship between MEI files and printed music notation. To understand why this is true, consider the transitive property on a relation, which states that

> If a is equivalent to b and b is equivalent to c, then a is equivalent to c.

In this context, if MEI notation is equivalent to Mup, and Mup is equivalent to printed music notation, then MEI is equivalent to printed music notation (see Figure 6.2). It is not possible to prove formally that the transitive relation holds on this set of transformations, because the Mup syntax is not defined [35] formally, but it is useful to build some intuition about why the transformations mean something about the MEI encoding.

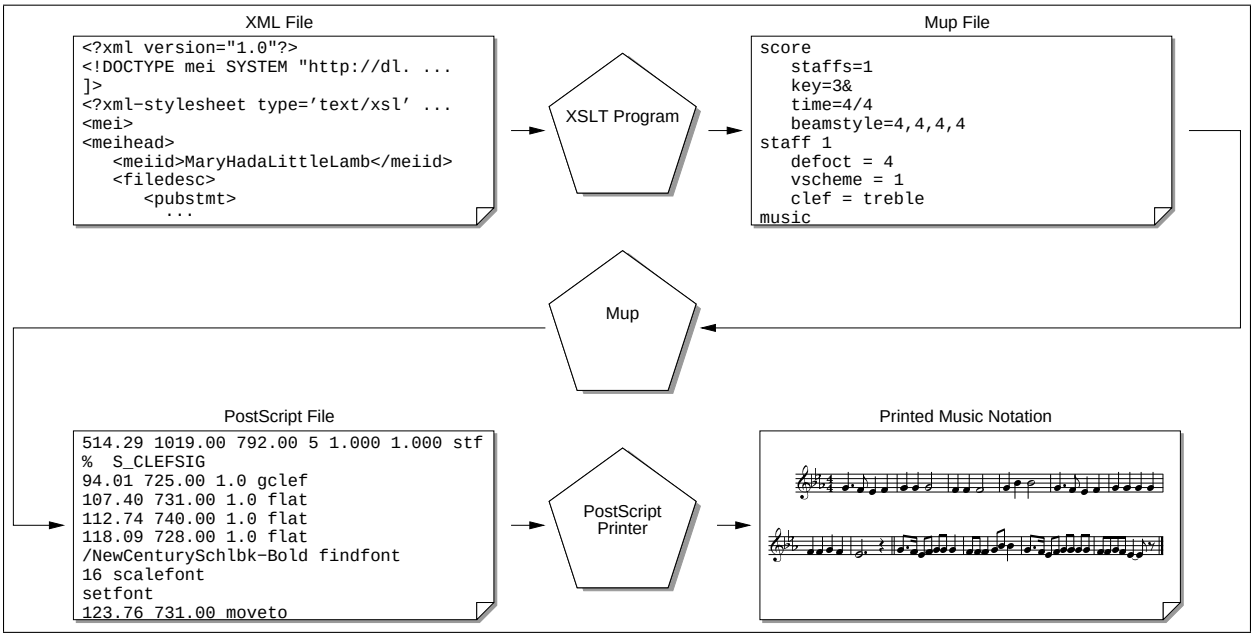


Figure 6.2: Because of transitivity, MEI-encoded files are equivalent to printed music notation.

6.1.1 A Sample Transformation

Let us examine how the XSLT script transforms "Mary Had a Little Lamb" into Mup notation. Recall that the MEI file is divided into music and its accompanying meta-data, such as the author

and copyright information. The XSLT does not transform this information, because Mup provides only basic facilities for including text. This thesis is concerned only with the file's musical content.

Mup files begin with a score definition similar to MEI files, so this part of the transformation is fairly straightforward. Here is the MEI `<scoredef>` element and the corresponding Mup notation:

```
        <scoredef meter.count="4" meter.unit="4" key.tonic="Eb"
        key.mode="major" key.sig="3f" beam.group="4,4,4,4">
           <staffdef id="_s1" octave.default="4" />
        </scoredef>
```

```
score
   staffs=1
   key=3&
   time=4/4
   beamstyle=4,4,4,4
staff 1
   defoct = 4
   vscheme = 1
   clef = treble
music
```

Briefly, this notation means that there is one staff in the key of 3♭, the time signature is $\frac{4}{4}$, and notes are to be beamed together in four time units of quarter notes by default.[2] The staff is defined next; it has a default octave of 4, one voice, and a treble clef. The keyword `music` marks the end of the preamble and the beginning of the musical content.

The XSLT script then transforms each event in the MEI notation into the corresponding Mup events. The first bar appears as follows in XML and Mup notation, respectively:

```
        <bar n="1">
           <staff def="_s1">
              <!-- accidentals required by the key signature not encoded. -->
              <note pname="g" dur="4" dots="1" />
              <note pname="f" dur="8" />
              <note pname="e" dur="4" />
              <note pname="f" dur="4" />
           </staff>
        </bar>
```

[2]See Section A.3 on page 60 for more details.

```
1    // begin bar 1
2  1 1: 4.g; 8f; 4e; 4f;
3  bar
```

The XML notation has already been explained in Section 5.4 on page 37; the Mup notation begins with a comment on the bar number. The next line is the first bar, beginning with the staff and voice number, followed by a colon. Each note is in the form `[time][pitchname][octave];`, but the octave is omitted in this case because Mup handles the default octave itself. Once processed with Mup, the file is ready for printing (see Figure 6.3).

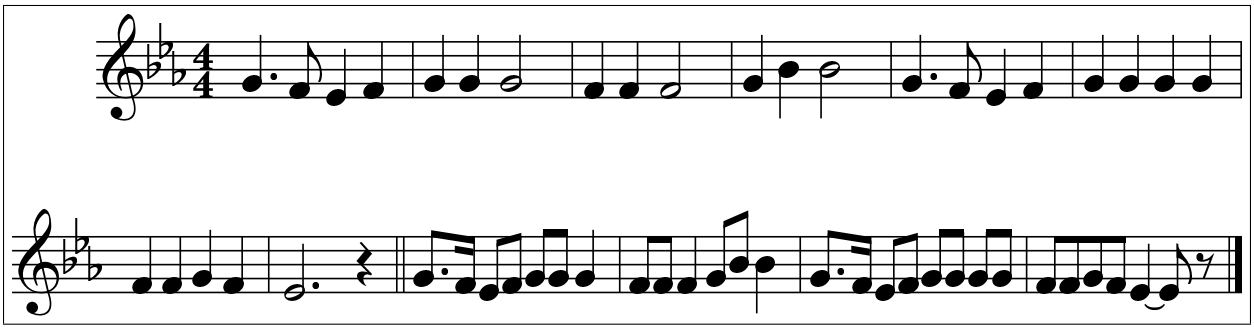


Figure 6.3: The fully transformed "Mary Had a Little Lamb."

6.1.2 Results

The thesis proposal did not define success criteria clearly, so it was difficult to say when the transformations were complete. When I realized this, I spoke to Mr. Roland and Prof. Martin. We agreed that it would be sufficient to transform the examples used in Selfridge-Field [51], because they are fairly complex, well-known, contain many special cases and exceptions to rules, and are used in Selfridge-Field to test music typesetting programs. Some of these examples cannot be notated with Mup easily, but others can be transformed very satisfactorily. The examples can be found in Appendix G on page 83.

After some experience with Mup, I realized that it can only represent a subset of what MEI can encode. Figure 6.4 shows that Mup does not support some clefs, and Mup only supports three voices, which is not sufficient for all music. In spite of this, the transformation is good enough to demonstrate a reasonable equivalence between MEI and music notation.

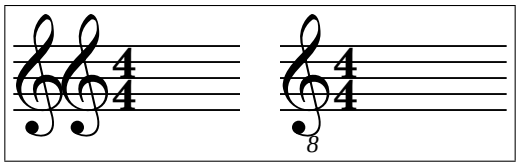


Figure 6.4: Mup does not support the archaic double-G clef shown at left. Instead, it is necessary to display a standard G clef with a joined 8 beneath it, which according to Read is equivalent [44, p. 55].

6.2 Content and Presentation

The project proposal stated that I would try to define a separation between musical content and presentation. Musical content can be thought of as the abstract information — the essence of what is encoded. Presentation, on the other hand, is the act of making that information accessible and meaningful to a human in some way, such as generating notation or playing the information back through speakers. Information about presentation inevitably needs to be encoded along with the information about content; the issue is how best to do this. Many encoding languages that contain some abstract information make special provisions for encoding presentation data as well, but some separate the presentation data out into a different language.

Unfortunately, it is difficult to apply the concept of "separation of content and presentation" to musical data, because it assumes that a clear separation between the two types of data exists. In fact, I discovered that "one man's content is another man's presentation" [30]. Instead of defining such a separation, I learned more about how and why the W3C [66] created CSS, or Cascading Style Sheets, to augment HTML. I learned that separating content and presentation is a specific application of a general technique in data representation.

6.2.1 HTML and CSS

Because "content and presentation" is not a very useful way to approach music, I began to re-assess the problem. Why seek such a separation? My previous experience is with HTML [20] and CSS [6]. HTML was originally a simple language, containing basic facilities for defining headings, paragraphs, tables, and so forth. As the Internet became popular, HTML authors desired greater control over page appearance, and browser vendors began extending HTML. As a result, web pages became non-standardized, would not work in all browsers, and the markup itself filled up with

messy, confusing formatting codes. CSS was the eventual answer, allowing authors a great deal of control over appearance in an auxiliary language while keeping HTML simple. This is usually called "separation of content and presentation," so I followed suit in my thesis proposal.

One advantage of using CSS with HTML is that it enables an author to indicate the document's logical structure. For example, it is possible to format text in a large bold font, but it is better to indicate that the text represents a heading, for example with an `<H1>` (heading level 1) element. A CSS stylesheet can then be applied document-wide to control the appearance of all `<H1>` elements uniformly. This can lead to very sparse markup, smaller file sizes, and the ability to extract meaning programmatically, based on document structure.

CSS brings many benefits to HTML and related markup languages. Here are a few:

- it allows the markup to be maintained easily
- it reduces the document complexity
- it reduces file sizes
- it gives authors control over document appearance
- it gives end users a means to view the document differently from the author's intention, if desired
- it can be used to define defaults globally, which can be changed in one place
- it helps impaired surfers use the Internet

However, the WC3 did not create CSS to separate content and presentation, but to ensure that authors could control the visual layout of their documents without changing the HTML standard. By doing so, the WC3 eliminated the need to extend HTML's capabilities. This helps ensure that the HTML language is a stable standard that will be forward-compatible with new generations of browsers and backward-compatible with older browsers. It was necessary to define the separation between essential content and the visual presentation of Web pages during this process, but the primary goal was extensibility. It has worked very well in practice. For example, CSS now allows control over how "voice browsers" read web pages to blind surfers. Extending HTML itself to allow this would be prohibitively difficult, and would require re-writing software that uses HTML.

CSS can also have disadvantages, depending on its usage. For example, if the CSS is kept in a separate file, it can become separated from the document or lost. On the other hand, by keeping the stylesheet in a separate file it becomes possible to change many documents in a single place, which could be beneficial (making it possible, for example, to change an entire website by changing a global stylesheet). It is also possible to override the CSS on a file-by-file or element-by element basis by redefining an externally defined CSS file, much as an externally defined DTD is overridden. For example, suppose the external CSS file defines a margin of 1 centimeter for a `<p>` element:

```
p   {margin:1cm}
```

An author can override this in a particular document, and on an individual element, as follows:

```
<!-- This tag goes in the document header -->
<style type='text/css'>
p   {margin:5cm}
</style>
<!-- This markup goes in the document body -->
<!-- this paragraph has a margin of 5cm -->
<p>Some text here</p>
<!-- this paragraph has a margin of 10cm -->
<p style="margin:10cm">Some text here</p>
```

One of the reasons CSS works well with HTML is that most HTML elements have similar properties, such as margins and padding. This may not be the case with music, even with similar types of data. For example, notes have stems, which have a position and direction relative to the note, a length, and a thickness. Rests, which are also musical events, have no stems. This means that setting defaults in the stylesheet and applying them to the entire document may not make much sense for music notation. On the other hand, there are some properties that would fit into this model, such as color. HTML elements are not all uniform either; it makes no sense, for example, to choose a font family for an `<IMG>` element, but it is possible to do so. Stylesheets are not perfect, but they are often a good solution.

To summarize, CSS gives document authors a great deal of control over the document's visual formatting. Authors can keep the formatting information in the file, externally, or just apply it to every element as desired. The benefits outweigh the disadvantages of separating information into

another language.

6.2.2 Results

The most important insight I gained from my research into the history of CSS and HTML is that rather than "content and presentation," it is more useful to think in terms of a markup language's different uses or *domains*. Since HTML's primary domain is visual, it makes sense to define separately how to render documents visually. This is a specific application of what could be termed *separating domain knowledge*. In practice, this means defining a language to encode data, considering its uses, and then providing for these uses separately from the data itself.

The primary benefit of this approach is that an auxiliary language can be changed without changing the markup language. In fact, the W3C is now working on CSS version 3, but HTML has not changed significantly since CSS was created.[3] This technique can also help to reduce the DTD's size; a DTD that contains information specific to all of the language's possible uses will probably be very unwieldy. A smaller DTD may be more attractive to users and designers, because of its shorter learning curve.

Since I did not focus on tangible results for this part of the project, but sought only to learn, I have little to present. I present my opinions and recommendations in the Appendices to this thesis.

6.3 Chapter Summary

This project's research showed that MEI-encoded musical data can be transformed into music notation, and investigated the motivation for separating structural data and domain-specific data in a markup language. This chapter explained why transforming MEI to printed notation is an indication of the MEI format's feasibility, and presented a sample transformation step-by-step. The second section of this chapter explained why separating "content and presentation" is not the most important reason to define auxiliary languages; instead, extensibility, stability, and compatibility are important reasons to define usage-specific information separately, and separating content and presentation may be a step in that process.

[3]The HTML 4.01 standard was released about the same time CSS2 was finalized and remains the most current version.

Chapter 7

Methods

This chapter describes how I solved the problems of transforming MEI files to printed music notation and learned about separating domain knowledge. Chapter 6 describes both activities, but whereas that chapter is about the project research's *what* and *why*, this chapter is about *how*.

7.1 Activities

I studied XML and XSLT over the summer of 2002 while working on the thesis proposal. I used both languages for a documentation project at work, for which I designed a custom markup language with XML Schema. This experience prepared me to understand the MEI format. After school began in fall 2002, I began writing the XSLT script to transform MEI into Mup. My advisor approved my research as a 3-credit class, and I worked approximately 2 to 10 hours a week on the project. By the end of the semester I had written a script that could handle the MEI format's most commonly used features. I spent most of this time learning about the languages and technologies I was using.

I met with Worthy Martin and Perry Roland at irregular intervals to give progress reports, ask questions, and receive guidance with the direction of the project. I wanted to ensure that the thesis project contributed to the MEI project and met Mr. Roland's and Prof. Martin's expectations.

There were no formal requirements for the XSLT script. Instead, the finished product was the guide: the script was correct if it produced correct Mup markup. The resulting XSLT is not well-designed from a software engineering point of view, but it does not need to be. Since the MEI

format is speculative and likely to change, there is little to gain from perfecting the XSLT. The script is essentially "hacked" together to prove that transforming MEI to music notation is possible.

The development process typically involved reading through a MEI file to check for features not yet implemented in the XSLT, finding the corresponding feature in the Mup manual, determining the desired output, changing the XSLT to implement that functionality, testing, and moving to the next feature. Perry Roland agreed to encode test pieces, so there were several files to experiment with.

7.2 Materials, Equipment, and Software

The project is almost entirely software-based, so I did not use any materials except for printing something occasionally. However, I used a variety of equipment and software. My primary equipment is my own computer; I use GNU/Linux [29] as my operating system. I sometimes used computer labs at the University of Virginia when I was away from my computer.

I used the Computer Science department servers to store the project's files, and accessed it through the CVS version control system [7]. Version control is essential for a project like this; it enables retrieval of every past version of every file, prevents overwritten files or other lost work, and keeps multiple copies of the work synchronized.

I wrote the XSLT in a text editor called Kate, which is part of the K Desktop Environment [25]. I ran XSLT transformations from the command line, so Kate's built-in terminal window was very helpful. After a while, I began using Vim [64] instead, because it enabled greater productivity.

I used Xalan-Java [1] as an XSLT processor initially. Xalan is a free Java implementation of the XSLT standard. However, it took so long to start up that the faster C++ implementation of the same software was more usable. I later discovered the `libxml` collection of XML tools (see [27] for details), and began using `xsltproc`, the XSLT processor that is included with the collection. It has some nice features, such as the ability to output the result tree, that made the work easier.

I use standard command-line tools to do my work. The GNU [13] text utilities are at the heart of all necessary text manipulation; they enable searching, sorting, counting words, concatenating, comparing, and many other tasks with files. Ghostscript [12] and the `psutils` packages enable viewing and manipulating PostScript files, including transforming PostScript into PDF and EPS

files for the proposal and thesis.

Mup [34] was the only software I needed to buy. Mup sends PostScript to the standard output, which can be saved to a file with the > operator and manipulated as desired.

Finally, I use LaTeX [26] to typeset the documents, and XFig [68] to create drawings and figures. I use MrProject [33]to create schedules and plan my work.

7.3 New Languages and Technologies

I needed to learn a variety of languages and technologies for this project. The most important languages were MEI, Mup, and PostScript, along with XML and XSLT. Since my only experience defining an XML format was with XML Schema, I had no experience with a DTD, and needed to study how a DTD works as well, especially since Perry Roland uses entities extensively to modularize the MEI DTD.

7.4 Stumbling Blocks

At several points I found that I did not know enough about markup languages to proceed, but I was always able to learn quickly and continue the work. Because the MEI format's semantics are not defined formally, some of the sample MEI files were also invalid or unclear. Emailing with Mr. Roland usually resolved these ambiguities.

The biggest stumbling block was a hard drive failure during the summer of 2002. I did not lose any data, because the CVS repository is on the Computer Science department's servers, but I lost my work environment of choice. It was necessary to log into a Unix server and work remotely, because the UVa lab computers run Windows, and do not have version-control software and other utilities installed.

The Mup language also presented some problems. The language has many shortcuts to save time, and learning the language by using the shortcuts kept me from understanding the full format of many features. It was necessary to rewrite the XSLT several times to handle the unabbreviated form of the commands, which the Mup manual does not always define clearly.

7.5 Valuable Resources

I found that my most valuable resources were people, books, and online references. Perry Roland and Worthy Martin were invaluable; Perry was especially helpful in answering questions about XML, DTDs, and XSLT. We met sometimes for entire afternoons to discuss MEI and related technologies.

Several books were very valuable. Michael Kay's book on XSLT [24] is the best such book I have found; the XSLT book from O'Reilly [62] was less helpful. Priscilla Walmsley's book *Definitive XML Schema* [67] was also very helpful. I referred to both of these books frequently. I often find good advice on books by browsing Amazon.com and reading the reviews.

The Internet contains many resources, of course. The specifications for XML, XSLT, and other languages are online, and I referred to them constantly. There are also many tutorials and introductions to various topics online. Table 7.1 lists some of the sites I have bookmarked for this project. I used Google's directory listings to find most of them.

Website	**Description**
http://www.mdwconsulting.com/postscript/postscript-operators/index.php	PostScript language reference
http://www.cs.indiana.edu/docproject/programming/postscript/postscript.html	A first guide to PostScript
http://directory.google.com/	Links to tools, technologies, and more
http://www.w3.org/TR/xslt	XSLT homepage
http://www.w3.org/XML/	XML homepage
http://www.w3.org/DOM/	DOM homepage
http://www.w3.org/XML/Schema	XML Schema homepage
http://www.w3schools.com	A variety of tutorials and references
http://www.saxproject.org/	SAX homepage
http://www.w3.org/Style/CSS/	Cascading Style Sheets homepage
http://www.w3.org/TR/CSS2/	CSS Level 2 specification

Table 7.1: Websites I referred to during this project.

Chapter 8

Conclusion

Perry Roland's MEI project seeks to create a comprehensive means of encoding musical data in XML. The MEI format is intended to be comprehensive, declarative, explicit, interpreted, hierarchical, formal, flexible, and extensible. The MEI project's goal is to create a framework for the storage and retrieval of musical data.

The MEI format is very complicated, defining not only a very complete XML encoding of music notation, but facilities for advanced typesetting such as a book's critical apparatus. It also extends XML encoding to allow the propagation and inheritance features discussed in Section 5.3.2. The DTD itself is defined as a set of character entities, which are used as building blocks to assemble the DTD. The result is a complicated, yet manageable, DTD.

Because XML is a generic way to represent data, it is a good choice for MEI, enabling simple XSLT transformations into other formats. This thesis project demonstrated transformations into printed music notation and showed that the MEI format can represent music adequately for generating printed music notation.

The second question of "separation of content and presentation" misses the point with data markup languages. The best reason to define an auxiliary language is to ensure that the primary language can be extended without changing it. Thus I do not define a division of musical content and presentation, but recognize that the idea does not apply to MEI as formulated.

8.1 Interpretation

Since MEI files are written in XML, it is easy to transform. However, because the format is based on the decision to treat music notation as the essential content, the format may be of limited use. This is contrary to the goal of comprehensiveness. The DTD also contains some design decisions, such as inheritance and propagation of attributes, that may violate other design principles, such as formality and explicitness, and may make it difficult to implement. Defining the format entirely in the DTD, instead of separating out domain-specific parts of the language, may also limit the format's extensibility and flexibility.

Transforming MEI into music notation, especially with the fidelity we were able to acheive, is a strong indication that the MEI format encodes sufficient information about music to be useful for many purposes. Most uses of music, such as analysis, only use a small subset of the information required to encode notation, so the MEI format may indeed develop into a universal language to encode musical information.

8.2 Recommendations

As a student of Computer Science, I have studied the value of good requirements gathering. Gathering requirements acknowledges explicitly what Covey calls "the principle that all things are created twice [5, p. 99]" by taking control of the first (intellectual) creation to help ensure that the second (actual) creation is a success. I suggest creating high-level and detailed requirements documents to help ensure the MEI project's success.

Regarding music notation as the essential content may limit the format's utility. I recommend regarding musical information as the essential content of music, and approaching music notation as one view upon that information. In particular, I suggest basing MEI on a very simplistic view of music. Musical information should be regarded as a series of events with a pitch and duration. Structural and other essential content, such as words associated with the notes a singer sings, should also be included, but the visual and other domains should be separated as much as possible from the music. Representing notation makes MEI two steps removed from reality and may limit the ways the encoded information can be used (see Figure 8.1 on the following page). Because MEI

files are XML, it is more flexible than other formats, but it would still be better to encode musical data directly rather than indirectly.

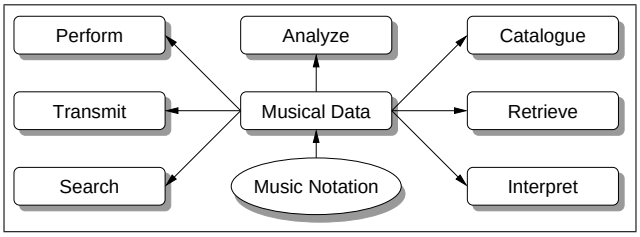


Figure 8.1: Compare this figure with Figure 1.1 on page 3. Representing music notation, rather than musical information in the abstract, removes MEI files another step from the thing represented. The arrows represent the fact that MEI data can be used for other purposes, but abstract musical data must first be extracted from the MEI file.

Because notation is perhaps the most important use for musical data, I recommend developing a comprehensive auxiliary language to give authors control over the appearance of a MEI document when viewing it as notation. However, I first suggest defining and documenting a set of units, a visual formatting model, and a standard means of extending MEI without rewriting the DTD. This may avoid haphazard extension by authors who need something that is not directly supported. Appendix C suggests ways to think about a structured, defined extension mechanism. The goal is not only backward compatibility with previous versions, but with other extensions (sideways compatibility) and future extensions (forward compatibility) as well. This goal cannot be realized without careful planning and a defined standard by which auxiliary languages can be designed. To help set a good precedent for future extension, domains could be defined by considering what the data will be used for, though the auxiliary language need not be implemented immediately.

While I recommend making the MEI format more general in these ways, in another way I recommend narrowing its scope significantly. I recommend restricting MEI to musical data and meta-data. For the visual domain, I suggest defining MEI with the following uses in mind: embedding a small fragment of notation into a Web page, inserting notation into a TEI text, mixing MEI notation with arbitrary XML markup in another file, and including a MEI file in another file as is currently done with images in HTML. I suggest removing textual markup facilities. If MEI files can be included in another document, authors can use an existing format to encode the text. To define this in MEI directly is to reinvent the wheel, and while it may be more convenient for some purposes, this comes at a cost. It is important not to prioritize ease of use for document authors

over ease of use for programmers and the programs they might write.

I recommend abandoning MEI's terse element and attribute names in favor of explicit names. Since XML files are supposed to be used by computers first and foremost, it makes sense not to worry about long element names. More importantly, though, humans will have to work with the format closely to write tools that can use it. The DTD should make this work as easy as possible.

I suggest discarding language features such as inheritance and propagation. These features cannot be supported by XML, because after the DTD is parsed the client application cannot know of their existence. These features also violate the requirement of explicitness. This is important, because it means that the MEI file is passing some decision or knowledge about the file's contents to another part of the chain, as discussed in Section 6.1 on page 40. This is a good idea for visual layout, where information should be defined by an auxiliary language and a visual formatting model, but not for the musical data itself.

Similarly, I suggest replacing information currently encoded in an application-specific manner, such as the use of `#CDATA`-typed attributes used to pass information to Mup in the example files, with explicit encodings that are application-independent. Again, relinquishing decisions or information to a processing application in an uncontrolled manner is likely to make MEI files less generally useful.

If advanced features not supported by DTDs are necessary, then other technologies may be appropriate. The XML Schema specification has matured significantly, and is now a stable, endorsed standard. It offers some features, such as the idea of a data type and the ability to extend it, that may serve as a replacement for the use of character entities in the MEI DTD.

I develop these and other recommendations more fully in the Appendices to this thesis.

Part II

Appendices

Appendix A

Suggestions for the MEI Project

The MEI DTD is well-built overall, but there may be room for improvement in future versions. The following suggestions address key areas that I identified during my research.

A.1 Avoid Non-Standard Extensions

The DTD defines several extensions to the *default declaration* of some attributes. An attribute's default declaration specifies whether an attribute is required, whether it exists even if omitted from the document, what its default value is, and whether the author can specify the value or not.

Because the DTD is built from externally defined entities, the XML parser sees the element specification as a simple element specification, and has no information about these extensions. For example, the following is the attribute list definition for the `<note>` element (portions of the following definitions are omitted for clarity):

```
<!ATTLIST %n.note;
          meiform            CDATA                #FIXED 'note'
          %a.log.note;
```

Recall the definitions of the entities referred to above:

```
<!ENTITY % a.log.note
                       oct              %OCTAVES;             %PROPAGATED;>
```

```
<!ENTITY % n.note     'note'>
<!ENTITY % PROPAGATED '#IMPLIED'>
<!ENTITY % OCTAVES    '(0|1|2|3|4|5|6|7|8|9)'>
```

Once all the entity references are replaced with the entity values, the XML parser's view of the definition is as follows:

```
<!ATTLIST note
      meiform               CDATA                  #FIXED 'note'
      oct                   (0|1|2|3|4|5|6|7|8|9)  #IMPLIED>
```

The XML parser has no way of distinguishing an attribute whose default declaration is the entity `%PROPAGATED` (the value of which is `#IMPLIED`) from one defined as `#IMPLIED` directly. Thus neither the parser nor any other application has information about these extensions. They can only serve as internal DTD documentation.

For two reasons, the MEI document format should not attempt to extend the XML syntax in this way. First, expecting an attribute to be defined on one element because it is defined on another violates the principle that a MEI document should be explicit, i.e. all information should be stated explicitly. These extensions must be handled by client software, which means that some information about the file is implicit in the client software.

Second, the document's formality is compromised. It should be possible to verify a file as correct formally without reference to anything external, but there is no way to verify that an attribute defined as a `DATE` or `NUMBER` conforms to this expectation. If it is truly necessary to constrain data further than is possible with a DTD, XML Schema may be a better technology to use.

A.2 Avoid Terse Attribute and Element Names

Since XML is designed for machines to manipulate, terseness is not important. Section 1.1 of the XML specification, "Origin and Goals," states that "terseness in XML markup is of minimal importance" [11]. Using longer element and attribute names does increase file sizes, and requires more keystrokes when hand-editing files, but these are minor issues because of compression and client software. Files are usually compressed, and long names compress more than short ones, so

a compressed XML file will probably be the same size no matter how long the names, because redundant data compresses well. Additionally, just as a human does not hand-edit a Microsoft Word file, a client application should usually use the MEI format as its native file format. Longer element names also have the significant advantage of being meaningful to a human reader who is working with the DTD directly.

A.3 Avoid Using `#PCDATA` for Attribute Values

There are many occasions where the MEI format encodes information as character data. Perry Roland advised that this data was to be passed directly through to Mup; the intention is to let authors use this data in a way that a particular program can understand. This has the undesirable effect of tying any such file to a particular program, which defeats the purpose of creating a universal encoding language.

An example of this type of character data is the `beam.group` attribute, which specifies how notes are to be beamed together. "Mary Had a Little Lamb" uses `4,4,4,4` for this value, which means that when possible, notes should be beamed together into four groups of notes, each having a combined time value of a quarter-note (see Figure A.1 on the following page). The transformation sends this data directly to Mup. Unfortunately, this file is now useful only with Mup because another program may have an entirely different method of encoding beam stylings. It would be better to define a structured XML representation of beam groups and their time values. An example fragment to encode this might look like the following:

```
<staff-definition ... >
  <beam-grouping>
    <group time-value="1bt" />
    <group time-value="1bt" />
    <group time-value="1bt" />
    <group time-value="1bt" />
  </beam-grouping>
</staff-definition>
```

This code defines four groups of notes, each of which contains enough notes to fill one beat of time. This example is essentially equivalent to the intuitive `beamstyle="4,4,4,4"` notation, but is

more formal — it is easier to write a computer program that works with such an encoding, because it takes advantage of the XML syntax. Note that the code does not use the unitless value "4." For more on a suggested system of units, see Appendix C on page 68.

I believe it would be even better to define beaming information in a stylesheet language than to use the XML construction shown above (see Appendix C). However, the example demonstrates how such information can be encoded in XML if desired.

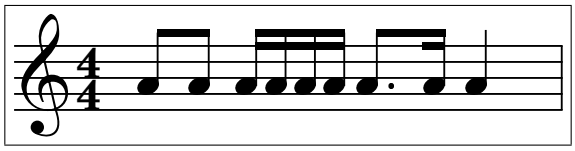


Figure A.1: The results of the `beamstyle=4,4,4,4` parameter to Mup: notes are beamed together in four groups, each with a total time value of one quarter note.

Another example is the placement of phrase marks, as in this example, which is the second staff from the last measure of the excerpt from the Mozart piano sonata in A Major, K. 331 (see Figure A.2 on the following page):

```
1   <staff def="s2">
2      <note id="_g5" pname="e" oct="2" dur.vis="16" grace="unacc" />
3      <note pname="g" oct="2" dur.vis="16" grace="unacc" />
4      <note pname="b" dur.vis="16" grace="unacc" />
5      <beam>
6         <note id="_n5" pname="e" dur="8" />
7         <note pname="e" dur="8" />
8         <note pname="e" dur="8" />
9         <note pname="e" dur="8" />
10     </beam>
11  </staff>
12  <phrase staff="s2" end1="_g5.x,_g5.y+6" end2="_n5.x,_n5.y+2" bulge="2" />
```

Note the format of the `<phrase>` element's attributes. Information about which notes the phrase mark begins and ends on, as well as horizontal and vertical offsets, is packed into dense text strings. The code means that the mark should be drawn on staff 2 beginning at the note named `_g1` and ending at the note named `_n1`. The (x, y) coordinates on the page for the respective ends of the phrasing mark are to be (`_g1`'s x position, `_g1`'s y position plus an offset of 6) and (`_n1`'s x position, `_n2`'s y position plus an offset of 2). The phrase mark should have a bulge value of 2.

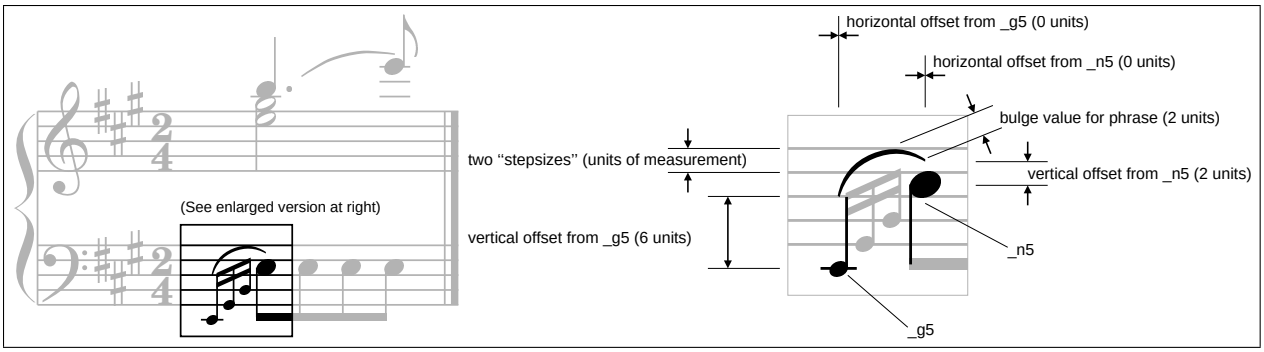


Figure A.2: The ends of the phrase mark are positioned relative to the two notes identified by _g5 and _n5. The bulge value for the phrase is the distance the phrase mark bulges from a straight line. The unit is a “stepsize” (one-half the distance between two staff lines).

Instead of placing the information together in this way, the phrase mark's beginning, end, bulge value, and horizontal and vertical offsets should be defined separately. Any processing application can then extract the information it needs and transform it into a suitable form.

This section may seem to conflict with other parts of this thesis, which suggest placing all domain-specific information into a single `#PCDATA` attribute and passing it on to a processing application. The difference is that the text is currently passed to a specific piece of software, whose behavior may differ from others. The domain-specific information, however, should be valid markup in an auxiliary language (see Appendix C), and the target application's behavior should be defined by a standard (see Appendix B), ensuring that any conforming processor could process the information as expected.

A.4 Avoid Including Formatting Information in the DTD

The last section gave an example of visual formatting attributes that are defined by the DTD. In fact, the essential musical information is very simple: the phrase starts on note `_g1` and extends to `_n1`. The visual placement of the curve that is drawn to represent the phrase should be left to a style language, or a visual user agent could decide where to place the mark to keep it from interfering with the rest of the notation. Again, the DTD should only include information about musical content and structure.

A.5 Discourage Authors From Modifying the DTD

The MEI DTD is designed explicitly to be modified by document authors. Perry Roland has said [48] that it is important for authors to be able to redefine the DTD from within the document (by first including the DTD and then overriding or adding to it). This allows flexibility, but it means that the MEI format itself is being changed, perhaps on a document-by-document basis. If this happens, there will be no single MEI format anymore. An XML processor will be able to verify that the file validates against the modified DTD, but programs that are expecting "canonical" MEI files will not know what to do with the modified files. For example, authors can change an element's name easily. Perry noted that this might allow authors to use a locale-specific version of an element, for instance to use the German word for "bar." If an author encoded a file with German names for all the elements, other applications would be unable to use that file without first translating the names into "canonical form." In short, I believe that the DTD should ensure that authors never need to modify it, but can instead extend it in a well-defined way, possibly with auxiliary languages as explained in Appendix C. To do otherwise would be to encourage multiple definitions of the MEI format, endangering its value as an interchange format.

A.6 Avoid Creating a Monolithic DTD

At present the MEI DTD itself appears relatively compact, but after a processor assembles it from entities, it is very complex indeed, making MEI files complex as well. For example, the DTD defines many attributes that appear in the document tree even if omitted from the original file. Here is an XML file's version of the `<scoredef>` element:

```
<scoredef meter.count="3" meter.unit="4" key.tonic="A" key.mode="major"
    key.sig="3s">
```

Here is the full form after the processor adds all the elements that the DTD specifies:

```
<scoredef meiform="scoredef" tune.temper="equal" tune.Hz="440" tune.pname="a"
    midi.track="1" midi.tempo="120" midi.port="1" midi.i nstr="1" midi.duty="80"
```

```
midi.div="96" midi.channel="1" tie.rend="medium" text.size="12"
text.font="rom" text.fontfam="times" spacing.system ="12,20"
spacing.staff="2" spacing.packfact="1" spacing.packexp="0.8"
slur.rend="medium" pedal.rend="term" page.scale="1" page.panels="1"
page.rightmar=".5" page.leftmar=".5" page.botmar="1" page.topmar="1"
page.units="in" page.width="8.5" page.height="11" optimize="no"
multi.number="yes" meter.rend="norm" lyric.size="12" lyric.font="rom"
lyric.fontfam="times" lyric.align="0.25" key.sig.showchange="yes"
key.sig.show="yes" grid.show="no" ending.rend="top" enclose.reh="box"
dist.text="2" dist.harm="3" dist.dynam="2" clef.visible="yes"
beam.slope="1.0,2 5.0" barplace="norm" barnum.visible="no" trans.semi="0"
trans.diat="0" clef.line="2" clef.shape="G" bar.number="no" meter.count="3"
meter.unit="4" key.tonic="A" key.mode="major" key.sig="3s">
```

Most of this information is domain-specific. Again, if information to support all possible uses of the musical data is encoded in the DTD directly, it will become very large. This could be a significant disincentive for its adoption as a standard.

A.7 Use Configuration Management Tools

As a student of software engineering, I would like to see the revision process managed more carefully. Perhaps using an open-source development model such as anonymous read-only CVS access would be a good idea: it would allow tracking changes, it would help ensure that people always have the most up-to-date versions of files, it would let people find out what changed between versions and why, and most importantly, it would let multiple developers collaborate effectively.

Appendix B

Visual Rendering of Music Notation

Most languages that define visual data layout also define something often called a "display model" or a "rendering model," which describes how a hypothetical piece of software might decide how to place the content, and gives context to the layout specification's meaning. Display models such as these appear in the Cascading Style Sheet, MathML, and SVG specifications, to name a few. They usually describe layout in terms of analogies, to clarify the language's semantics. For example, CSS defines a "box" model, and SVG speaks of "painting" objects onto the "rendering canvas."

Music notation's layout is extremely complicated. Element placement cannot be determined by simple rules; even an algorithm for determining where system breaks (a line of staff(s) across the page is called a "system") should occur is complicated. There are also many exceptions depending on what other elements are nearby; for example, phrase marks should be placed out of the way of the rest of the notation as much as possible.

I believe that a logical way of thinking about notational layout is necessary for a successful standard. A documented layout model serves as clarification for the standard itself. It specifies nothing about the way processing software must actually work, but specifies the expected results. This helps keep implementations consistent; for example, web browsers that implement the CSS standard render pages very uniformly, demonstrating that the CSS approach worked well in this case. A layout model can also help clarify the layout language's requirements during the design stage.

As mentioned in Section 6.1 on page 40, this type of "knowledge" can be embedded anywhere in

the chain of data and processing applications between the source document and the final output. It is best to have this knowledge *explicitly* encoded in the system, rather than implicitly leaving it to some step of the process, but that does not mean that it must be written into a source document. For example, the way a browser lays out an HTML document is specified even if no CSS stylesheet is attached — the CSS specification contains a "default" stylesheet, and browsers are supposed to render documents as though this stylesheet were attached if no other style information is given. A defined standard that a processor is expected to follow can be thought of as explicitly encoded in the document, though it is in fact encoded in the processor. This is a much better way to control document appearance than by placing formatting information into the DTD.

B.1 Nested Boxes

It is logical to define music notation as nested boxes. The outermost enclosing element could be a single box, with nested boxes to enclose each system and perhaps each bar. These boxes could form a framework within which to lay out the visual elements of the notation. This approach makes it easy to imagine embedding a fragment of music notation inside another file, just as an Encapsulated PostScript file can be embedded in this thesis document.

Each element itself might be thought of in terms of a containing box, with points defined as references. For instance, these points could be used as "handles" to which other elements could be attached, or as points from which to measure offsets. Figure B.1 demonstrates this graphically.

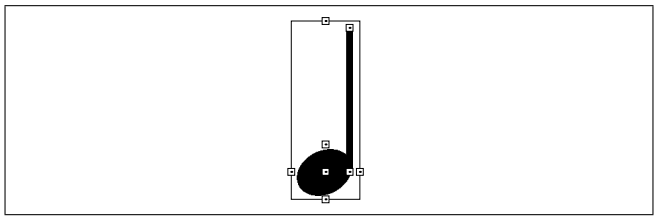


Figure B.1: An element box and its handles. The ▫ symbol represents a handle.

This model leaves it up to the author to ensure that no elements overlap or conflict; for example, it would be possible to specify the absolute position of an element, and this could interfere with another element. This approach has the benefit of letting authors have complete control over where things are placed — if authors want elements to overlap, then they are free to make them do so.

B.2 Self-Aware Elements

It might also be possible to make notational elements "aware" of their own surroundings. OpenType fonts use a similar idea [42]. Thus a graphic element could actually reshape or move itself to avoid conflicts. Programmers could create a standardized set of these symbols, perhaps as a font or in a similar manner. This relieves authors of the burden to place everything correctly. It also makes sense from an implementation standpoint; an entire class of decisions are handled at a low level, eliminating the need to implement them in every piece of software built to work with the format (again, just like fonts) and ensuring uniformity across implementations. On the other hand, it might introduce issues with elements stubbornly refusing to display as the author wishes because the elements themselves "know better."

In either case, having a defined formatting model as an imaginary reference implementation could help in creating and interpreting an auxiliary language.

Appendix C

Stylesheet Design Concerns

C.1 Why Have a Style Language?

The need to use a style language for visual control may not be obvious. One of the most often-cited examples in this thesis and elsewhere is the tremendous benefit CSS offers to HTML authors. As an HTML author, the two things I appreciate the most are the ability to specify style information for an entire website globally in a single file, and the clean HTML markup that results from the absence of style information in the markup itself. As a side benefit, the stylesheet, which is in a separate file, can be cached by a browser, eliminating the need to download it more than once and reducing file sizes.

These benefits might not apply to music notation in the same way, because it is not clear that style information could be applied globally to music notation in a way that would benefit authors. For instance, imagine defining the default vertical and horizontal offsets for a slur's ends. Slurs usually look best when they start and end near the centers of the notes they tie, and are offset vertically by about one inter-staff distance from the notes' centers (see Figure C.1 on the next page). If the note stem or another element is in the way, the end must be moved to compensate.

Defining this type of information in an attached stylesheet would be almost useless. These defaults serve very well for most cases, and in the instances when authors would need to override the defaults, they would want to attach that information directly to the element itself. Having a stylesheet with defaults either encoded in each document or as an external file would provide no

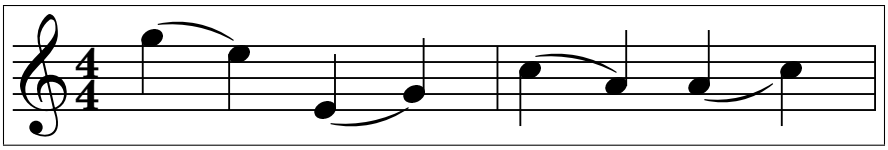


Figure C.1: Slurs are placed to avoid the note stems. Each end is offset horizontally a small amount from the center of the note it is attached to, and vertically offset about one inter-staff distance from the center of the note. The last pair of notes is a special case, because the slur cannot extend through the stem of the C.

benefit. Instead, this information should be in the default stylesheet specified by the standard.

Thus style languages would not necessarily bring the benefits of smaller file sizes and cleaner markup to a document (though having all style information in a single `style` attribute, as with HTML and CSS, would significantly reduce the number of attributes on elements). Instead, style languages would enable extensions to the format without changing it. I believe that this benefit alone outweighs any inconveniences, such as needing to create a separate language for the style information.

C.2 Categories of Style Information

It would sometimes be necessary to specify the appearance of some notational elements. For example, when compiling several files together into a book of notated pieces, the pieces should all look similar. Therefore, some user-level control over the notation is necessary, and should be very easy to use: an axiom in Computer Science is to "make the common case fast" (or easy, in this case). This suggests a divided approach:

- A rendering model as in Appendix B
- A set of rules on how elements are to be laid out
- A way to specify families of shapes to use for the notation
- Styling concerns

The distinction between the second and third items is subtle, but note that one has to do with element placement, while the other deals with exactly what shapes are placed in the space reserved by the layout software. In other words, part of the layout process is to allocate space on the page

for each element's "box," and then choose a shape to insert into the allocated space. The second type of default above deals with where these reserved spaces are placed; the third is similar to selecting a font (and in the case of any text associated with the music, actually *does* select a font). The final category of defaults deals with styling concerns, for example the decision to make bar lines join staffs, whether to number the measures of a piece, and so on.

C.3 A Default Stylesheet

This re-introduces the concept of a default stylesheet. In the absence of explicit instructions on how to lay music out, all processors conforming to the standard would be expected to render the music notation as if there were a default stylesheet attached. This stylesheet would address the second category of information; examples of defaults would be inter-staff spacing, line thickness, length of a note stem, how much a ledger line projects on either side of a note, and so forth. For the vast majority of music, authors would not need to use or even know about these defaults. However, in the event someone needed to make a ledger line project out more than usual on one side of a note, there would be a way to do so. Some common sets of these defaults, if applicable, could be grouped together under the name of a "family." An author could then set many defaults at once simply by declaring the family. For instance, an author might wish to declare that this music should be laid out in "tiny" mode, and the staff spacing, inter-note spacing, and other parameters would be adjusted accordingly.

The default stylesheet would specify standard fonts and other shape families, but authors would probably find these controls useful more frequently, so they could be grouped together and treated with special shortcuts, not unlike the CSS specification's shortcuts for common uses of styling information. Setting a font family and size is an obvious application.

The default stylesheet would also include the last category of styling information. For example, bar numbering would be turned off for most music. With these aspects of music notation specified in the default stylesheet, the need to actually encode with the document all the information that never changes, and is never needed by the author, is obviated. It is still explicitly there because it is required by the standard, but it is usually redundant and can be eliminated.

C.4 Stylesheets as an Extension Mechanism

Stylesheet languages could be defined for many different domains. As a starting point, at least the visual, gestural, analytical, and performance domains could be considered. However, it is likely that at some point a universal musical data format will need to handle some unforeseen domain or use, and style languages may be able to meet this requirement without requiring changes to the DTD. If a general style language specification is created, it should be possible to define some formal rules that any new style language or extension to an existing one must follow. This could be expressed as a grammar that any new language must comply with, just as XML is a grammar for defining a markup language. The style languages might reap the same benefits as XML data enjoys just by being encoded in XML: many applications could be written to work with it, standard libraries might exist for parsing it, and so on.

A means of defining the semantics, or formal meaning, of such a language would also be necessary. This seems like a formidable task, but perhaps it is possible. For example, text editors often read a "syntax definition" file to know how to highlight the syntax of a particular computer language, easing the programmer's job. Most such editors provide a facility for installing new syntax definitions, so the editor can support a language definition that may not have been part of the standard installation. Perhaps it would be possible to describe the semantics of a domain-specific style language in a similar way. These definitions could be kept as part of the standard MEI documents, and users could simply download and install a file to let their processing applications know how to deal with the features of a new style language.

C.5 Stylesheet Namespaces

Note that all information in the default stylesheet is still essentially visual layout information, and is only separated into categories to draw a distinction between different types of style information. In fact, there is no reason not to define any document-level styles in the same place or stylesheet file, regardless of the category it belongs to. Other categories of visual styling information might be defined for guitar chords or other application-specific domains.

To separate out these categories, a concept similar to that of XML namespaces could be

used. A namespace prefix might be affixed to style information. To deal with highly specialized means of control over styling, namespaces could even have sub-namespaces. These namespaces would differ from XML namespaces in that they would be standardized, and a formal definition of the meaning for each namespace would be available as mentioned above. As an example of such a namespace, consider the "visual domain" of music notation styling. Perhaps an author would want to specify that a note should be displayed one inch above its normal position for some reason. The author could write the markup as `<note ...(attributes)... style="visual:x-offset='1in'">`. The namespace, in this case, is the part in front of the colon, `visual`. Sub-namespaces could be defined by additional colon-separated words; for example, styling information for a guitar chord chart's appearance is clearly in the visual domain, but it only applies to guitar chords, so perhaps it would appear as `visual:chord:guitar:font-family='Helvetica'`. Namespace aliases could be defined in stylesheets to reduce the burden of typing the full namespace each time it is needed; for example, a namespace alias `g` could be mapped to the full name, `visual:chord:guitar`. A default namespace could also be declared, eliminating the need to use a namespace on each element; this might be useful, for example, when notating organ music.

C.6 A System of Units

One of CSS's most important features is its flexible system of units. CSS defines units for lengths and colors. Authors can specify units of length both absolutely and relatively.[1] Absolute units include many different systems, such as English, Metric, and device-dependent (pixels). Relative units include the em and the ex. Music adds the need for at least one more relative unit of length (what Mup refers to as the "stepsize," or half the distance between staff lines), units of time, and units of pitch.

Both absolute and relative units should be defined for time, even though relative units are used more commonly. In addition to the obvious unit, seconds, it may be necessary to consider samples as an absolute unit (this assumes that the stylesheet defines a sampling rate as well). Relative units could be expressed as beats, or again, because it is so convenient, as the denominator of the fraction of a whole note, as with Mup. However, whereas Mup uses the denominator alone (a `4`

[1] Position can also be absolute and relative — do not confuse position and units.

indicates a quarter note, an `8` an eighth note), a unit suffix should be appended, for example the letters `th`.

Pitch also needs to be specified in both ways. Absolute pitch is best expressed in cycles per second. Relative pitch is usually expressed in terms of *semitones*, which are the interval between adjacent keys on the keyboard. Since most musical scales repeat at one-octave intervals, and a note at one octave vibrates at twice the frequency of the one below it, a semitone is defined in the Western well-tempered scale to be relative: each note's frequency is $\sqrt[12]{2}$ higher than the last. Semitones are further divided into 100 *cents*, or $\sqrt[1200]{2}$ steps of pitch.

Most music conforms to the Western well-tempered scale, or can at least be notated on the staff even if the tuning is different,[2] but to be universal there must be a way to specify a different interval between notes, different interval between octaves, and different numbers of notes per octave. Perhaps one way to do this would be to create optional parameters for inter-octave spacing, an indication of how many notes are in an octave, and whether the notes are spaced evenly. These three pieces of information could be used to calculate the inter-note interval: for example, if each octave's frequency is 2.5 times the frequency of the previous, and there are 11 equally spaced notes in an octave, each note's frequency is $\sqrt[11]{2.5}$ times higher than the previous. It should also be possible to specify a pattern of notes and pitches that form a "scale" that repeats every octave, regardless of the intervals.

Other common ways of thinking about intervals are in seconds, thirds, fourths, and so on, which represent the tonal distance between the root of a scale and the second note, root and third note, etc. Some intervals can be diminished or augmented by a semitone.

This is very complicated, but necessary for a universal encoding language. On the other hand, it would be easy to make things too complicated, as SMDL did. The key is to remember that there is a point of diminishing returns (make the common case fast), and that only the most necessary features should be included in the basic definition. An easy way to extend the language should be sufficient to support the less common cases.

[2] Just intonation is one common alternate tuning; notes are not separated by a uniform interval.

C.7 Attaching Style to a Document

The HTML/CSS model of allowing authors to attach styling information in several different ways seems to work very well. Stylesheets can be linked from external files in several ways, written directly into the document via the `<style>` element, and attached directly to elements with a `style` attribute. Authors can also specify families, or "classes," of styles, and then declare that an element "belongs" to that class with the `class` attribute, as in `<p class="large">`. Stylesheets can even single out a particular element by its `ID` attribute. These ways of attaching styling to a document are mature and well-accepted, so they may be good candidates for the MEI format.

Style information could also be defined to propagate or inherit. Since the stylesheet processing application must conform to the implementation model, there is no reason not to add these features into a stylesheet language, even though it appears to be a poor design decision for the XML schema itself. CSS and other style languages commonly define such models — in fact, this is what it means for a CSS attribute to "cascade."

Appendix D

Mup: The Music Publisher

Mup is an easy-to-use, but quite capable, music publishing program. I chose it for this thesis research at Perry Roland's suggestion, partly because MEI is based upon some of its features and partly because of its ease of use and standard Unix behavior: input goes into the standard input and PostScript comes out of the standard output. While studying for this thesis research I have learned to use it enough to create fairly complex music notation. This Appendix contains some observations on the Mup file format and the way the program works.

Mup files are plain text files containing instructions in the Mup language. The language itself is not defined [35] formally. Mup's creators provide a manual that explains the language. The language is defined by the manual and the program's behavior.

Mup is designed to be very easy for humans to use. Since there is no graphical editing interface, a person must edit the text, and a user-friendly language is important. The Mup language is therefore very flexible, almost casual. Notes can be entered one at a time or in entire chords. The pitch and time value of a note "propagate" from one to the next to ease editing. Every parameter has a default value, so if it is left off the program inserts it automatically. Most of the time these defaults are exactly what the average user needs. For example, paper size defaults to 8.5x11 inches. Many of these features are included in MEI.

Mup does have some limitations. It can only handle three voices, and it does not handle some archaic features such as uncommon clefs. Cross-bar beaming is difficult. Curves, such as phrase marks, are sometimes difficult to get right, especially when they continue past a system break.

Mup also requires each measure to have exactly the right amount of notes in it to match the time signature, which requires awkward workarounds for common needs such as a "pick-up measure," which is usually incomplete — this requires placing "spaces" into the notation, which take up time but are invisible. There are also "noncompressible spaces," which seem to be inspired by TEX. Displaying such notations as time signature changes as desired may require awkward uses of invisible bars or other odd methods. It is also difficult to get Mup to typeset two pieces on the same page. It is possible to add text to the page, but it is difficult to control this feature.

Because Mup is so informal, there are many ways to write the same thing, but the long, explicit way can always be used. After my experiences transforming *to* the Mup format, I have gained enough insight to realize that explicitness is desirable. If I were ever to transform *from* Mup, I would start by writing a script to ensure that the Mup file is in the long, explicit format.

In short, Mup is centered around the notational aspect of music, with only basic facilities for other tasks, but it is robust enough to be very useful for an average person's notation needs, and even do a good job on demanding markup, with some effort. It served well for this thesis project, but my experiences so far with music notation markup languages have convinced me that formality is a good thing!

Appendix E

Comparison of MEI and MusicXML

Of the XML musical data languages being developed currently, MusicXML [36] is the most widely used. Several commercial tools can work with MusicXML via plug-ins, and programs exist to transform MusicXML to and from other formats. Given this, why create MEI?

MusicXML and MEI have very different goals. For example, MEI seeks to provide a framework for encoding, whereas MusicXML is focused on the immediate need for an interchange format. Beyond the differences in purpose, however, there is an important difference in the formats. The MusicXML format encodes nearly everything in elements, whereas MEI mixes elements and attributes. Here is a sample measure encoded in MusicXML and MEI, respectively:

```
<measure number="1">
    <attributes>
        <divisions>24</divisions>
        <key>
            <fifths>-3</fifths>
            <mode>major</mode>
        </key>
        <time>
            <beats>3</beats>
            <beat-type>4</beat-type>
        </time>
        <clef>
            <sign>G</sign>
            <line>2</line>
        </clef>
    </attributes>
    <!-- some markup omitted -->
```

```
    <note>
        <pitch>
            <step>B</step>
            <alter>-1</alter>
            <octave>4</octave>
        </pitch>
        <duration>24</duration>
        <voice>1</voice>
        <type>quarter</type>
        <stem>down</stem>
        <lyric number="1">
            <syllabic>single</syllabic>
            <text>Auf</text>
        </lyric>
    </note>
    <!-- some markup omitted -->
</measure>
```

```
<bar n="1">
    <staff id="s1">
        <voice id="s1v1">
            <note pname="b" acci="f" oct="4" dur="4" />
            <!-- some markup omitted -->
        </voice>
    </staff>
</bar>
```

The most obvious difference is the MEI markup's terseness. The MusicXML markup uses much longer element names. This is good, but it also uses an all-element approach to encode the music. The MusicXML FAQ includes a question about this [36, FAQ]:

> **Why do you use all these elements instead of attributes?**
> This is mainly a stylistic decision. Several XML books advise representing semantics in elements rather than attributes where possible. One advantage of doing this is that elements have structure, but attributes do not. If you find that what you are representing really has more than one part, you can create a hierarchical structure with an element. With attributes, you are limited to an unordered list. For information retrieval applications, it can also be easier to search directly for elements rather than for attribute/element combinations.

My experience with XML has been the reverse — it is a design decision, not a stylistic decision. Attributes can be much less expensive to process, especially in a language like XSLT that relies on XPath to locate elements. It may be necessary to use recursive processing and function calls, which

are very expensive, to retrieve deeply nested information. It may also be more difficult to retrieve some information if the elements and attributes are not distinguished correctly; an all-element approach may not preserve functional dependencies between data, and lossless joins of data sets may be impossible because some information has been discarded (more on this). In other words, a mixture of elements and attributes can encode semantic meaning that is discarded when using all elements.

I believe that a firm grasp of the mathematics involved should inform decisions about attributes and elements, rather than breaking anything up if it is composed of several parts. A database designed in this manner would be difficult to use, and there is a strong relationship between XML data and databases. If the XML schema is designed correctly, it is possible to store the data in a relational database, but "in particular, nested elements and elements that recur (corresponding to set valued attributes) complicate storage of XML data in relational format" [54, p. 382]. However, a mapping to relations can be designed, in which XML elements whose schema is known are mapped to relations and attributes. Another mapping involves creating a tree view of the data and replicating this in relations and attributes in the database, but this may require many joins to reassemble the data [54, p. 383]. The mapping from relational databases to XML may be accomplished by several methods, but one way is to simply map rows to elements and columns to attributes. Microsoft SQL server implements this method [54, p. 1004]. For more on XML and databases, see [3].

The fundamental problems in database schema design and markup language design are related, and should be approached with an eye toward eliminating redundancy, enabling flexible information storage, and preserving functional dependencies between data. While I have not tested MusicXML, I suspect that it might perform poorly with large data sets because of its all-element structure. The common sense approach also suggests that information *about* something, such as a note's pitch, should be an attribute. The element named `<attributes>` indicates design confusion.

Using all elements may also limit the format's flexibility. It may be easier to add an attribute if necessary than to add an element. Adding an attribute is not likely to break compatibility with existing files, and existing client software will probably have no problems with the new attribute. Well-designed software looks explicitly for the attributes it needs, like well-designed SQL queries (it

is a bad idea to write `INSERT INTO EMPLOYEE VALUES("BEN", ...);` — this software will break as soon as the database schema is changed), and the presence of an additional attribute is usually no problem. Adding an element to a language changes file structures and may cause incompatibilities with client applications. In short, MusicXML's design might make evolution expensive and difficult, and may represent a disincentive for its adoption.

Appendix F

DTDs and XML Schema

The MEI format is defined in a DTD, but there are several ways to design data markup languages. DTD has the advantage of being supported broadly and simple. DTDs can also be extremely easy to maintain and extend. However, the DTD method's simplicity is also a disadvantage in some applications. DTDs do not provide a means to specify many data types, and cannot constrain an element's data at all. DTDs are also difficult to use for more complicated markup. For example, it is very difficult to specify that an element must occur between 4 and 99 times inside another element. Choosing two out of three items is even difficult:[1]

```
<!ELEMENT happiness ((good,(fast|cheap))|(fast,cheap))>
```

Schemas are a much more powerful way to specify a markup language. They make it easy to constrain data in any way desired, even supporting regular expressions for advanced pattern-matching constraints. They make it easy to define structure; it is simple to solve the 4-to-99 problem in a schema language. Unfortunately, they are not yet as widely supported, and some are still in development.

The major disadvantage in using a schema language for a project like MEI is the lack of support for external entities. Recall that entities are used like Legos in the MEI DTD; they are defined once and used many times, making the DTD maintainable and extensible. This is not possible with a schema language, but it may be possible to use the concept of data typing to achieve the same result. Data types can be defined as desired in XML Schema, the most important schema language.

[1] With thanks to Perry Roland.

Simple and complex types can be defined, and then types can be derived from other types, much as data types are designed in object-oriented software modeling and design. This technique may allow a similar level of flexibility and extensibility, while giving much finer control over the contents of a MEI document.

Appendix G

Transformation Test Cases

This chapter contains pieces used in *Beyond Midi* [51, p. 22–27] to test the capabilities of various encoding formats. Most of the examples are rendered very similarly to the original. In most cases, the XML files from which these pieces were produced are not specially "tweaked" to get the notation to look the same as the examples in *Beyond Midi*, other than scaling the notation to fit correctly onto the page. Exceptions are noted in the text.

For ease of comparison, a second figure follows each figure transformed from the Mup notation that resulted from the XSLT transformations. These figures are used with the kind permission of Eleanor Selfridge-Field and are taken from the *Beyond Midi* website [2]. These specific images are from the MuseData section of the website, and were created from MuseData files.

I timed the transformations on a Toshiba Portege 2010 laptop with an 866MHz processor and 256MB of RAM. The laptop is running GNU/Linux RedHat 8.0, and uses `xsltproc` as the XLST processor. The times, as reported by the `time` command, are as follows:

Example	**Time (real, user, sys)**		
The Mozart Trio	.113	.096	.018
The Mozart Clarinet Quintet	.162	.150	.012
The Saltarello	.132	.113	.020
The Telemann Aria	.216	.205	.010
The Unmeasured Chant	.109	.096	.014
The Binchois *Magnificat*	.134	.121	.012

G.1 The Mozart Trio

This example (see Figure G.1) is taken from the second trio section of Mozart's Clarinet Quintet.[1] The challenge in this piece is transposing the first staff, which is notated in C Major but, since it is played on an A clarinet, is actually in A Major. It is very well rendered on the whole. There is a phrase or tie that begins on the last note of the first staff, but since there is no note for it to extend to, Mup ignores this. Mup also places some phrase marks oddly, such as the phrase mark on the tuplet, and has trouble with phrases that cross system breaks.

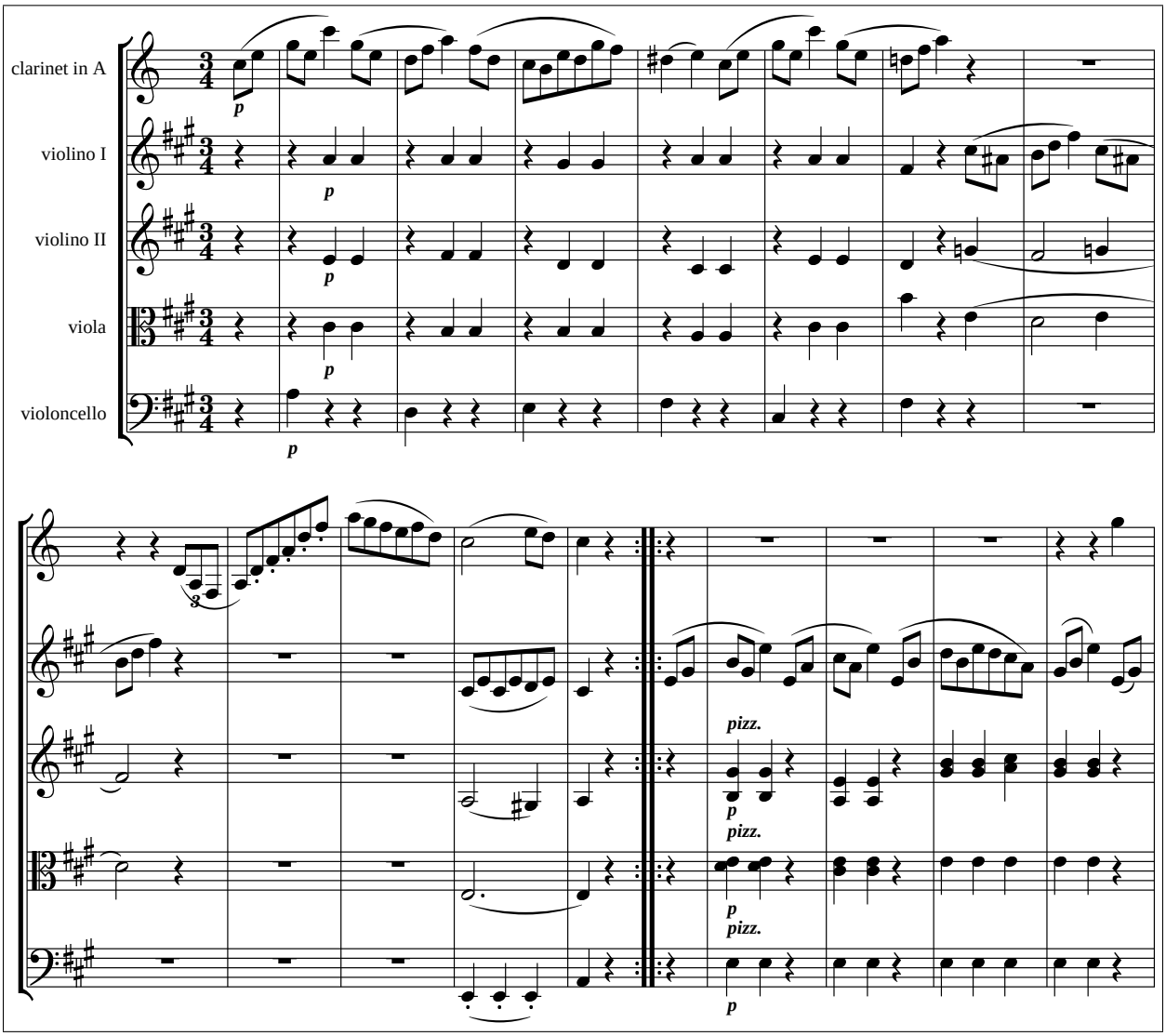


Figure G.1: Second trio from the Mozart Clarinet Quintet, K. 581 ("Mozart Trio").

[1]The instruments appear to be incorrectly labeled, but this example attempts to duplicate the original notation exactly.

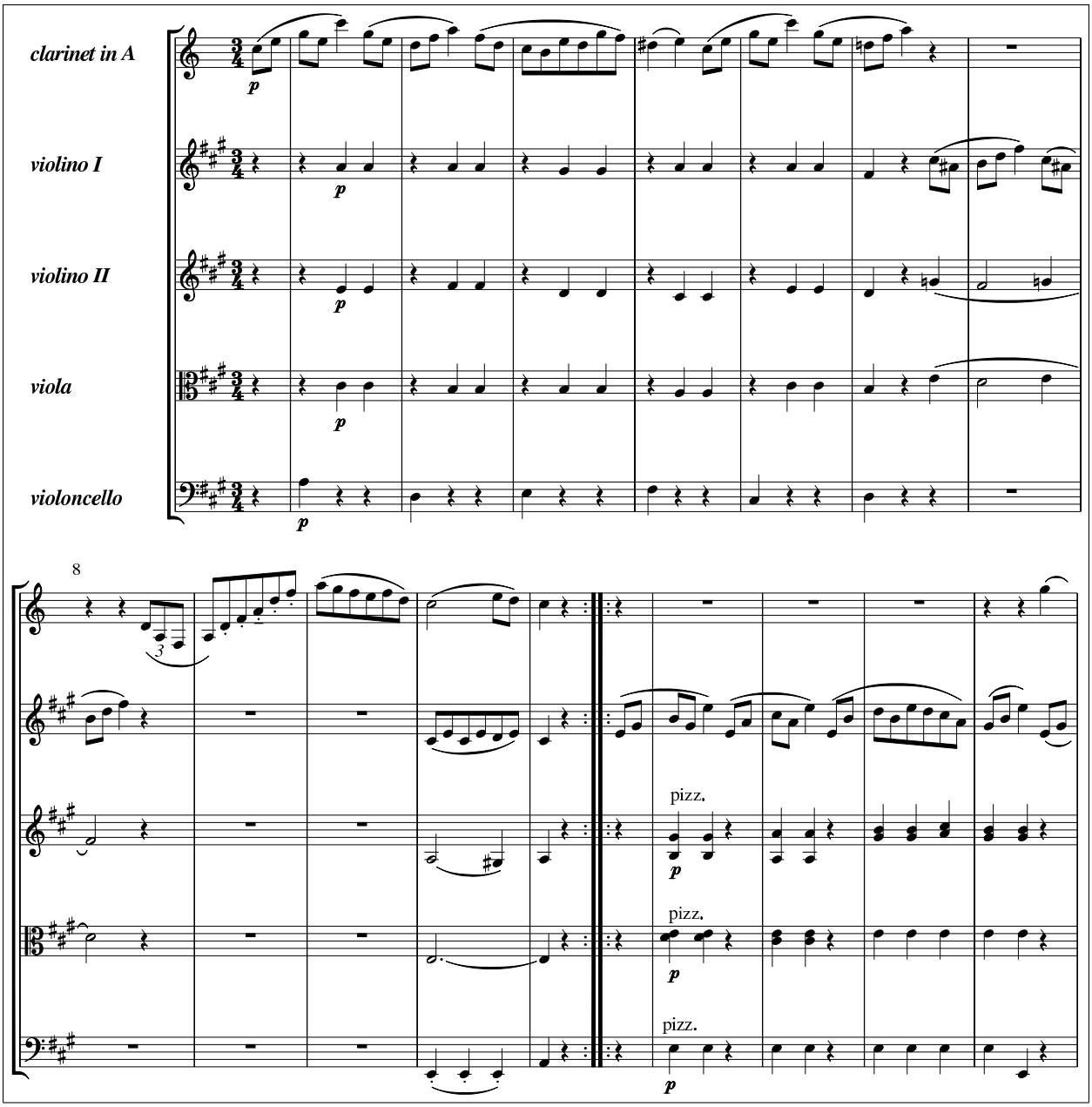


Figure G.2: The original figure: Second trio from the Mozart Clarinet Quintet, K. 581 ("Mozart Trio").

G.2 The Mozart Clarinet Quintet

This example (see Figure G.3 on the next page) has mixed durations within chords, grace notes preceding chords, and arpeggiated grace notes. The roll on staff 1 in the first measure also crosses voices. In this case, Mup's placement of the slurs on the left-hand grace notes is so bad that the

slurs are encoded with explicit endpoints and curve values. Mup cannot slur to an entire chord, as in the right-hand part in measure 4, so those are also placed explicitly. The disadvantage of this is that the logical structure of the XML can no longer indicate that there is something special about those notes — instead, the XML just indicates that there is a curve on the page.

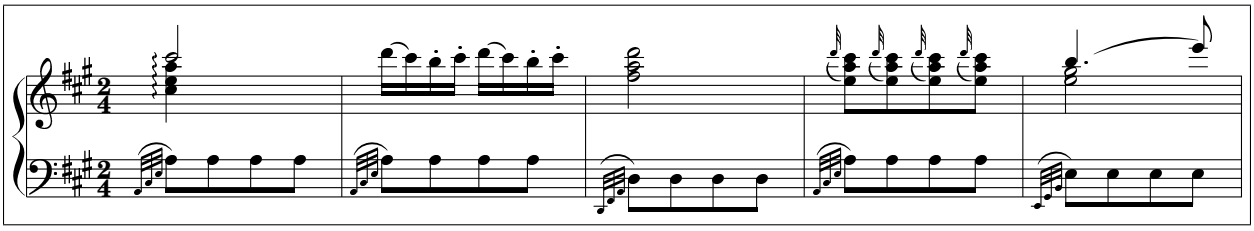


Figure G.3: Excerpt from the Mozart piano sonata in A Major, K. 331.

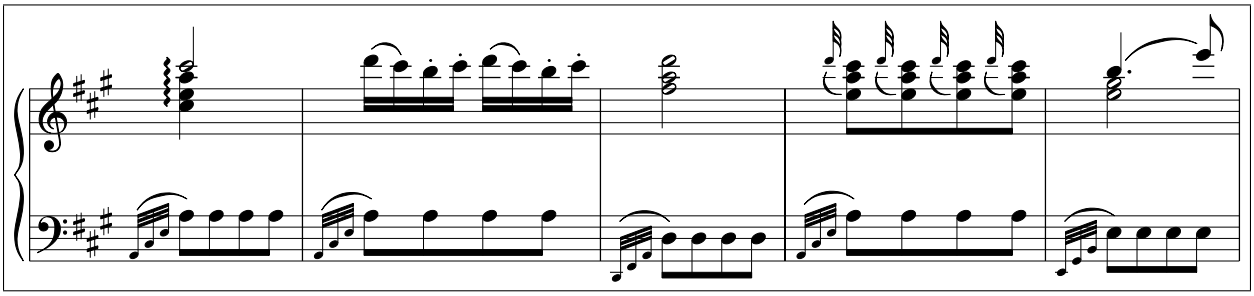


Figure G.4: The original figure

G.3 The Saltarello

This piece demonstrates multiple endings. The parts were originally numbered with a small number above the staff in the first measure of each part, but this information is not encoded in the XML.

G.4 The Telemann Aria

This example (see Figure G.7 on page 88) demonstrates lyrics and multiple voices. Many of the notes have small heads ("cue" size), and the voices are extremely complex. Mup's default note placement does not match the original in some places, but little can be done about this without hand-editing the Mup code after it is transformed. It is necessary to remove the fourth voice from the piece before Mup will process it (Mup is limited to three voices). The lyrics also use German characters, and complex timing.

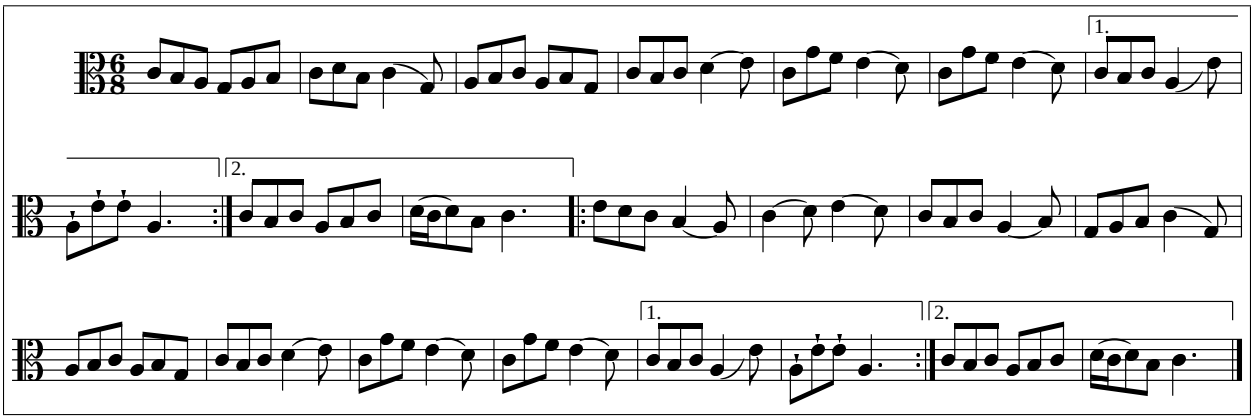


Figure G.5: Anonymous saltarello.

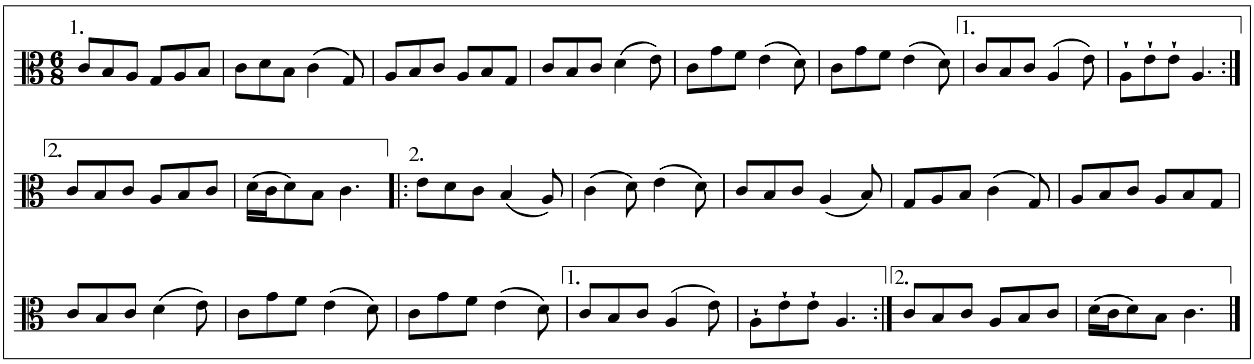


Figure G.6: The original figure

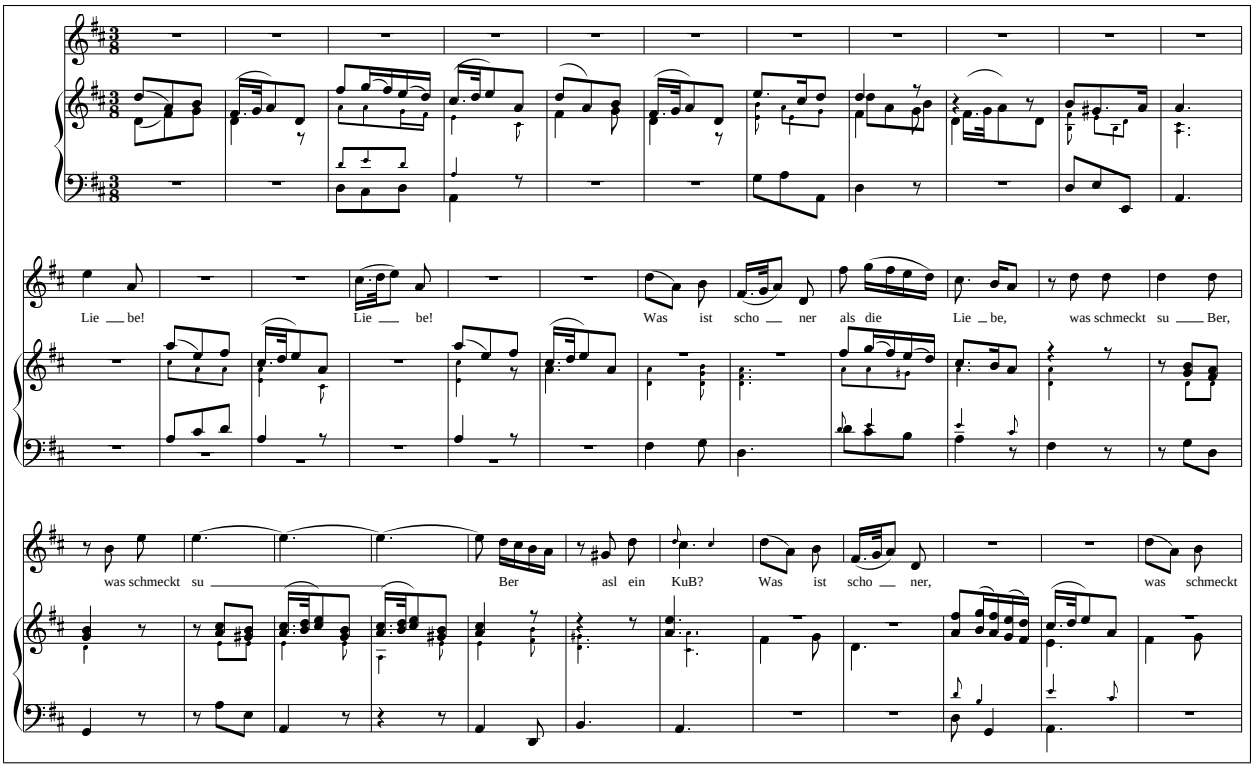


Figure G.7: The Telemann aria “Liebe! Liebe! Was ist schöner als die Liebe?”.

G.5 Unmeasured Chant

This example (see Figure G.9 on the next page) is difficult because there is no meter. Mup requires each bar to have exactly the right number of notes, so it is necessary to change the time signature, instruct Mup not to print the time signature, and print invisible bars. The original notation does not use modern notes or a modern staff, but this is impossible to duplicate with Mup.

G.6 The Binchois *Magnificat*

This example (see Figure G.11 on page 90) demonstrates several tricky layout problems Mup does not handle gracefully. In particular, there is no way in Mup to get a stem to point down on the right-hand side of a note. It was necessary to use a small macro to get Mup to draw a line in the appropriate place. This macro is embedded in the XSLT. Some of the other interesting features of this piece are the absence of note stems in the first measure, the absence of a second staff in the first measure (Mup displays the staff, but there is none in the original), and a number of editorial

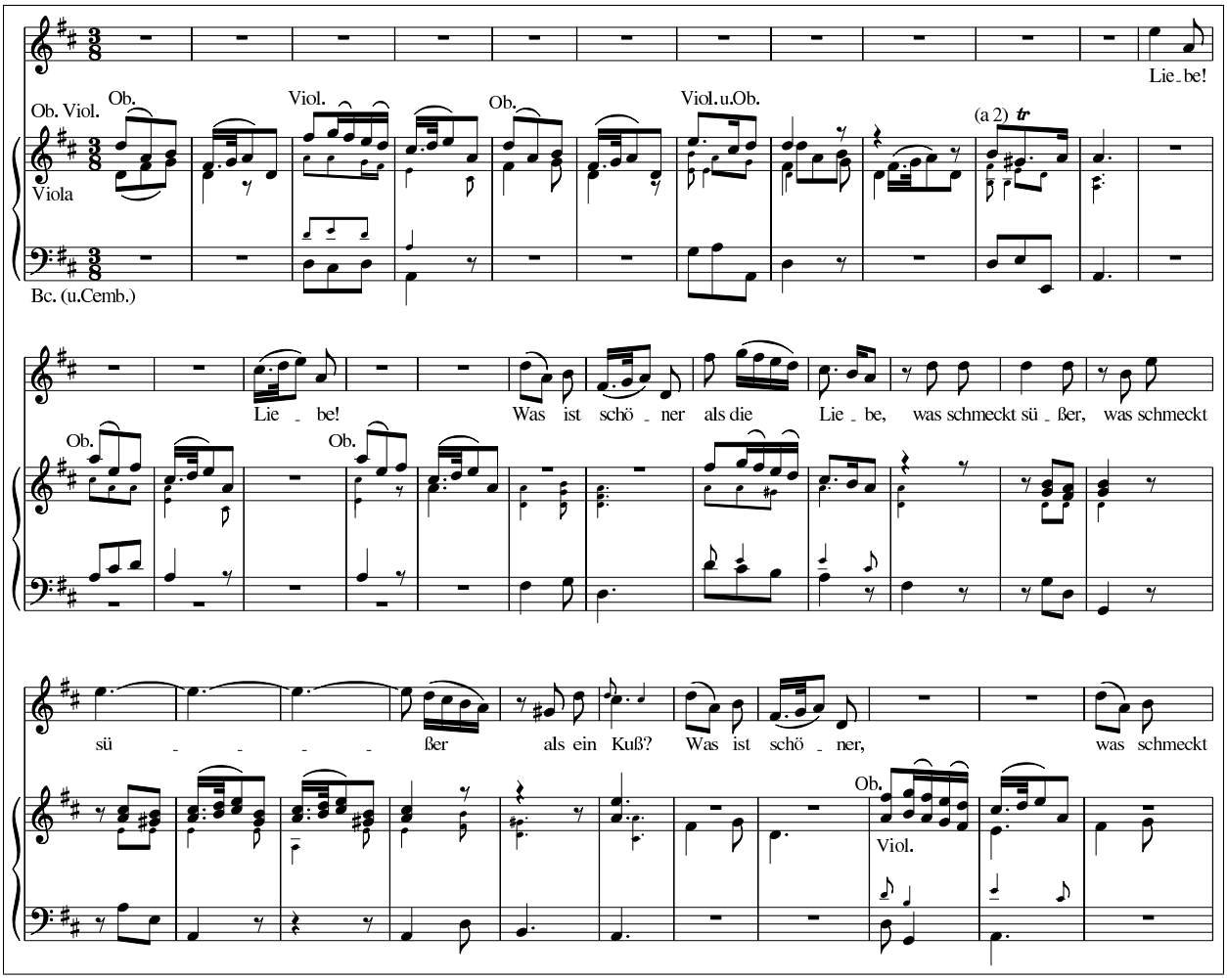


Figure G.8: The original figure

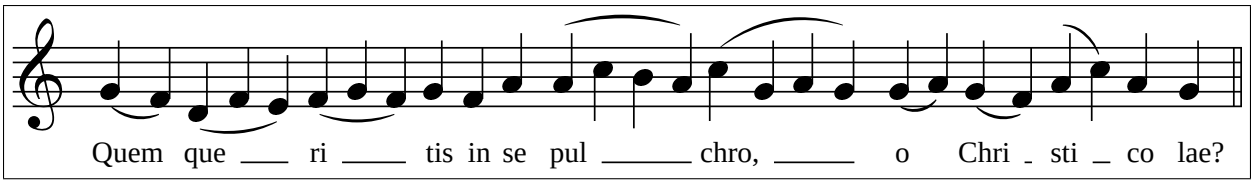


Figure G.9: The unmeasured chant “Quem queritis” written in modern notation.

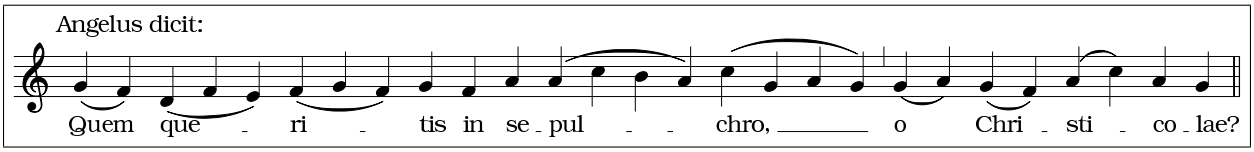


Figure G.10: The original figure

elements, such as editorial accidentals.

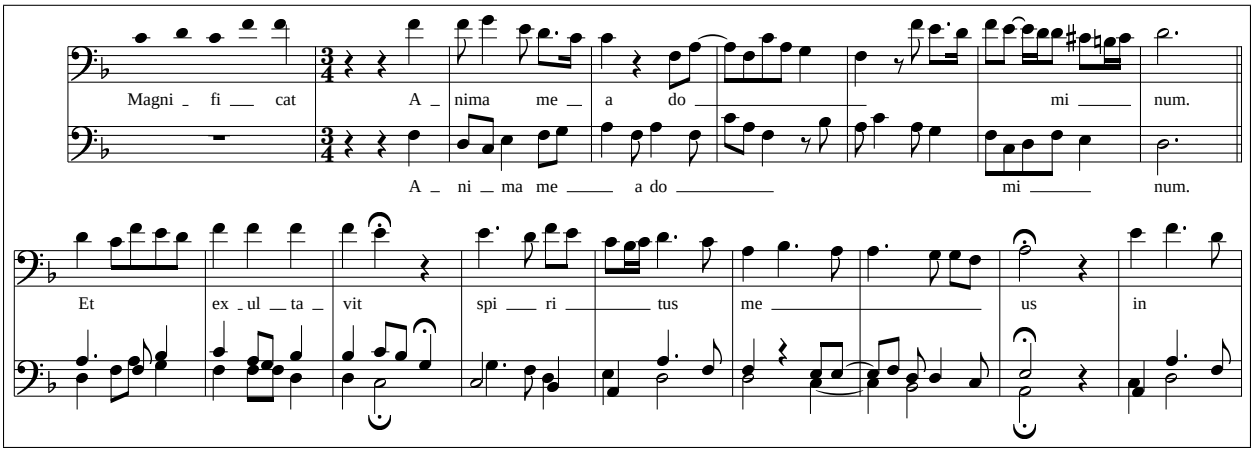


Figure G.11: The Binchois *Magnificat*.

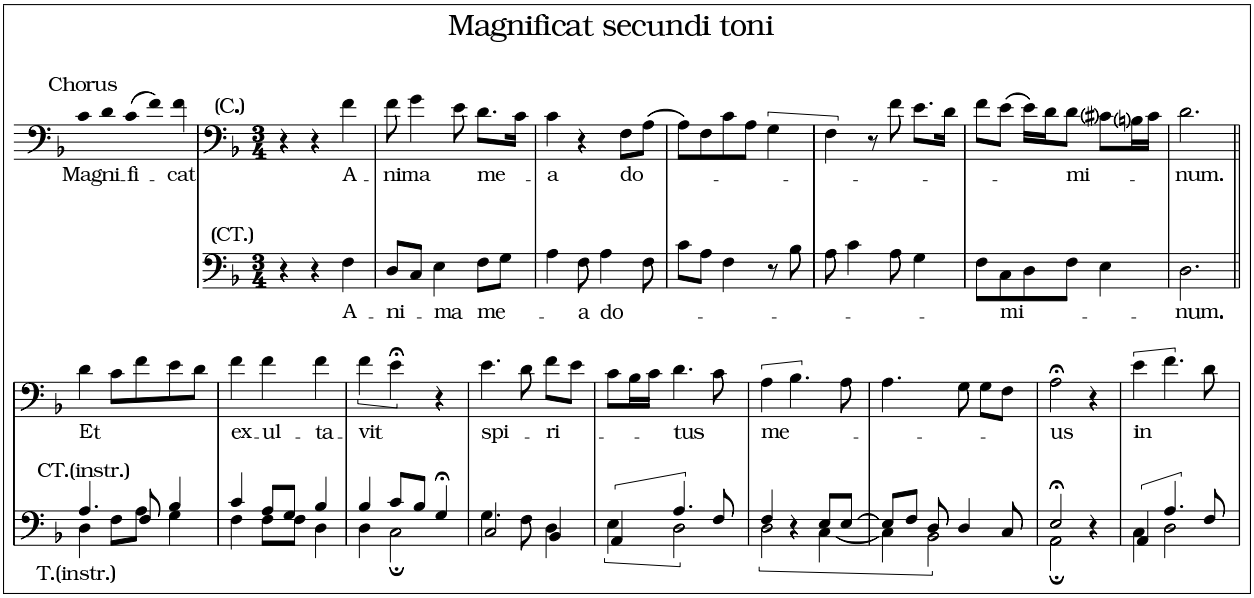


Figure G.12: The original figure

G.7 Summary

Most of the examples match the original closely. Special processing was sometimes necessary, and once it was necessary to remove a 4th voice from the Mup file by hand, but overall these examples demonstrate convincingly that the MEI format can represent music notation acceptably.

Appendix H

Code Listing: XSLT Transformation Program

This appendix contains a code listing of the XSLT stylesheet for transforming from MEI to Mup formats. The XSLT program is designed to be minimally capable — in other words, just enough to transform the examples used for this thesis. However, much of the framework is there for transformations in general. The remaining musical data features of MEI are simply a matter of handling special cases.

Designing this stylesheet in this manner means that it fails to exploit some of the tricks it could use. For example, much of the Mup language is very logical, with many similar constructions. If I had to design the XSLT again, I would take advantage of this and cut down on redundancy. I would also separate the file into several files and use `<xsl:include>` elements to assemble the stylesheet, a technique I have used in the past.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3     <xsl:output method='text' encoding='UTF-8' />
4     <xsl:strip-space elements="*" />
5
6     <!--
7     XSLT stylesheet for transforming from Perry Roland's MEI format into Mup
8     format for creating printed music output. The MEI DTD is available online at
9     http://dl.lib.virginia.edu/bin/dtd/mei/mei.dtd.  Author: Baron Schwartz
10    $Revision: 1.5 $
```

```
   -->

<!-- Global variables -->
   <!-- A newline (carriage return) -->
   <xsl:variable name="nl">
<xsl:text>
</xsl:text>
   </xsl:variable>

   <!-- Mup macros for making special stem types -->
   <xsl:variable name="macros">
   <xsl:text>
   // macros for defining custom stems
   define STEMDNRIGHT(NOTEID, LEN)
   line (NOTEID.x+1.3, NOTEID.y) to (NOTEID.x+1.3, NOTEID.y-LEN)
   @
   define STEMDNLEFT(NOTEID, LEN)
   line (NOTEID.x-1.3, NOTEID.y) to (NOTEID.x-1.3, NOTEID.y-LEN)
   @
   </xsl:text>
   </xsl:variable>

   <!-- Begin matching the document's root element. -->
   <xsl:template match="/">
      <xsl:value-of select="$macros" />
      <xsl:apply-templates />
      <xsl:value-of select="$nl" />
   </xsl:template>

   <!-- Match the meihead, which is the first part of the document. -->
   <xsl:template match="meihead">
      <xsl:value-of select="$nl" />
      <xsl:text>//meihead not yet implemented</xsl:text>
   </xsl:template>

   <!-- Match the front matter. This is not yet implemented. -->
   <xsl:template match="front" >
      <xsl:value-of select="$nl" />
      <xsl:text>//front matter not yet implemented</xsl:text>
   </xsl:template>

   <!-- The body is the main content of the document. -->

   <xsl:template match="body">
      <xsl:apply-templates />
   </xsl:template>

```

```
58    <!-- Each mdiv element is a movement, act, song, etc. Simple documents will
59    have just one mdiv, which contains the whole work. -->
60
61    <xsl:template match="mdiv">
62       <xsl:apply-templates />
63    </xsl:template>
64
65    <!-- The full view of the musical work. An alternate view(s) may be given in
66    the separate parts for each performer. -->
67
68    <xsl:template match="score">
69       <xsl:value-of select="$nl" />
70       <xsl:apply-templates />
71    </xsl:template>
72
73    <!-- The first element in the score, containing meta-information. -->
74    <xsl:template match="scoredef">
75       <xsl:value-of select="$nl" />
76       <xsl:text>score</xsl:text>
77
78       <!-- Extract the number of staves by recursively adding all staves -->
79       <xsl:value-of select="$nl" />
80       <xsl:text>   staffs=</xsl:text>
81       <xsl:value-of select="count(//staffdef)" />
82
83       <xsl:call-template name="score-stuff" select="." />
84       <xsl:call-template name="beamstyle" select="." />
85
86       <!-- Extract the page scale -->
87       <xsl:if test="@page.scale">
88          <xsl:value-of select="$nl" />
89          <xsl:text>   scale=</xsl:text>
90          <xsl:value-of select="@page.scale" />
91       </xsl:if>
92
93       <!-- Extract a staff definition for each staff -->
94       <xsl:apply-templates />
95
96       <xsl:value-of select="$nl" />
97       <xsl:value-of select="$nl" />
98       <xsl:text>music</xsl:text>
99    </xsl:template>
100
101   <xsl:template name="score-stuff">
102      <!-- Extract the key signature -->
103      <xsl:if test="@key.sig != '0' and (@key.sig)">
104         <xsl:value-of select="$nl" />
```

```
105         <xsl:text>   key=</xsl:text>
106         <xsl:value-of
107            select="substring(@key.sig, 1, string-length(@key.sig) - 1)" />
108         <xsl:choose>
109            <xsl:when test="contains(@key.sig, 'f')">
110               <xsl:text>&amp;</xsl:text>
111            </xsl:when>
112            <xsl:otherwise>
113               <xsl:text>#</xsl:text>
114            </xsl:otherwise>
115         </xsl:choose>
116      </xsl:if>
117      <!-- Extract the time signature -->
118      <xsl:value-of select="$nl" />
119      <xsl:text>   time=</xsl:text>
120      <xsl:value-of select="@meter.count" />
121      <xsl:text>/</xsl:text>
122      <xsl:value-of select="@meter.unit" />
123      <xsl:if test="@meter.rend='invis'">
124         <xsl:text>n</xsl:text>
125      </xsl:if>
126   </xsl:template>
127
128   <xsl:template name="beamstyle">
129      <!-- Extract the beamstyle -->
130      <xsl:if test="@beam.group">
131         <xsl:value-of select="$nl" />
132         <xsl:text>   beamstyle=</xsl:text>
133         <xsl:value-of select="@beam.group" />
134      </xsl:if>
135   </xsl:template>
136
137   <xsl:template match="sectiondef">
138      <!-- Extract the new score def -->
139      <xsl:value-of select="$nl" />
140      <xsl:value-of select="$nl" />
141      <xsl:text>score</xsl:text>
142      <xsl:call-template name="score-stuff" select="." />
143      <xsl:value-of select="$nl" />
144      <xsl:value-of select="$nl" />
145      <xsl:text>music</xsl:text>
146   </xsl:template>
147
148   <xsl:template match="staffgrp">
149      <xsl:if test="name(ancestor::node()[1]) = 'scoredef'">
150         <xsl:call-template name="extract-staff-symbols" select="." />
151      </xsl:if>
```

```
152      <xsl:apply-templates />
153   </xsl:template>
154
155   <xsl:template name="extract-staff-symbols">
156      <xsl:if test="@symbol and @symbol != 'line'">
157         <xsl:value-of select="$nl" />
158         <xsl:text>   </xsl:text>
159         <xsl:value-of select="@symbol" />
160         <xsl:text>=</xsl:text>
161         <xsl:for-each select="staffdef[1]">
162            <xsl:value-of select="substring-after(@id, 's')" />
163         </xsl:for-each>
164         <xsl:text>-</xsl:text>
165         <xsl:for-each select="staffdef[position() = last()]">
166            <xsl:value-of select="substring-after(@id, 's')" />
167         </xsl:for-each>
168      </xsl:if>
169      <xsl:if test="@barthru='yes'">
170         <xsl:value-of select="$nl" />
171         <xsl:text>   barstyle=</xsl:text>
172         <xsl:for-each select="staffdef[1]">
173            <xsl:value-of select="substring-after(@id, 's')" />
174         </xsl:for-each>
175         <xsl:text>-</xsl:text>
176         <xsl:for-each select="staffdef[position() = last()]">
177            <xsl:value-of select="substring-after(@id, 's')" />
178         </xsl:for-each>
179      </xsl:if>
180      <xsl:for-each select="staffgrp">
181         <xsl:call-template name="extract-staff-symbols" select="." />
182      </xsl:for-each>
183   </xsl:template>
184
185   <!-- Template to specify stuff for each staff that is defined with a
186   <staffdef> -->
187
188   <xsl:template match="staffdef">
189      <xsl:value-of select="$nl" />
190      <xsl:value-of select="$nl" />
191      <xsl:text>staff </xsl:text>
192      <!-- As per email, the @id is expected to be 's' + the staff # -->
193      <xsl:value-of select="substring-after(@id, 's')" />
194      <xsl:if test="@octave.default" >
195         <xsl:value-of select="$nl" />
196         <xsl:text>   defoct=</xsl:text>
197         <xsl:value-of select="@octave.default" />
198      </xsl:if>
```

```
199       <xsl:if test="@label.full">
200          <xsl:value-of select="$nl" />
201          <xsl:text>   label = &amp;quot;</xsl:text>
202          <xsl:value-of select="@label.full" />
203          <xsl:text>&amp;quot;</xsl:text>
204       </xsl:if>
205       <xsl:call-template name="beamstyle" />
206       <xsl:if test="@trans.diat">
207          <xsl:call-template name="transpose">
208             <xsl:with-param name="steps" select="@trans.semi" />
209             <xsl:with-param name="interval" select="@trans.diat" />
210          </xsl:call-template>
211       </xsl:if>
212       <!-- The voice scheme; 'vscheme = 1' by default -->
213       <xsl:value-of select="$nl" />
214       <xsl:text>   vscheme = </xsl:text>
215       <xsl:choose>
216          <xsl:when test="voicedef">
217             <xsl:value-of select="count(voicedef)" />
218             <xsl:text>f</xsl:text>
219          </xsl:when>
220          <xsl:otherwise>
221             <xsl:text>1</xsl:text>
222          </xsl:otherwise>
223       </xsl:choose>
224       <xsl:value-of select="$nl" />
225       <xsl:call-template name="clef" />
226       <!-- Extract any voice definitions -->
227       <xsl:value-of select="$nl" />
228       <xsl:apply-templates select="voicedef" />
229    </xsl:template>
230
231    <!-- Extract out voice definitions. -->
232    <xsl:template match="voicedef">
233       <xsl:value-of select="$nl" />
234       <!-- The @id of the voicedef is expected to be 's#v#' -->
235       <xsl:text>voice </xsl:text>
236       <xsl:value-of select="substring(@id, 2, 1)" />
237       <xsl:text> </xsl:text>
238       <xsl:value-of select="substring(@id, 4, 1)" />
239       <xsl:if test="@octave.default" >
240          <xsl:value-of select="$nl" />
241          <xsl:text>   defoct=</xsl:text>
242          <xsl:value-of select="@octave.default" />
243       </xsl:if>
244       <xsl:call-template name="beamstyle" />
245    </xsl:template>
```

```
246
247    <!-- Set a clef. MEI defines G|GG|F|C|perc and line numbers; Mup
248    names them all -->
249    <xsl:template name="clef">
250       <xsl:text>   clef = </xsl:text>
251       <xsl:choose>
252          <xsl:when test="@clef.shape = 'G'">
253             <xsl:choose>
254                <xsl:when test="@clef.line = 1">
255                   <xsl:text>frenchviolin</xsl:text>
256                </xsl:when>
257                <xsl:when test="@clef.line = 2">
258                   <xsl:choose>
259                      <xsl:when test="@clef.trans = '8va'
260                                      and clef.octave = 'basso'">
261                         <xsl:text>treble8</xsl:text>
262                      </xsl:when>
263                      <xsl:otherwise>
264                         <xsl:text>treble</xsl:text>
265                      </xsl:otherwise>
266                   </xsl:choose>
267                </xsl:when>
268                <xsl:otherwise>
269                   <xsl:text>treble</xsl:text>
270                </xsl:otherwise>
271             </xsl:choose>
272          </xsl:when>
273          <xsl:when test="@clef.shape = 'GG'">
274             <xsl:text>treble8</xsl:text>
275          </xsl:when>
276          <xsl:when test="@clef.shape = 'F'">
277             <xsl:text>bass</xsl:text>
278          </xsl:when>
279          <xsl:when test="@clef.shape = 'C'">
280             <xsl:choose>
281                <xsl:when test="@clef.line = 1">
282                   <xsl:text>soprano</xsl:text>
283                </xsl:when>
284                <xsl:when test="@clef.line = 2">
285                   <xsl:text>mezzosoprano</xsl:text>
286                </xsl:when>
287                <xsl:when test="@clef.line = 3">
288                   <xsl:text>alto</xsl:text>
289                </xsl:when>
290                <xsl:when test="@clef.line = 4">
291                   <xsl:text>tenor</xsl:text>
292                </xsl:when>
```

```
293               <xsl:when test="@clef.line = 5">
294                  <xsl:text>baritone</xsl:text>
295               </xsl:when>
296               <xsl:otherwise>
297                  <xsl:text>alto</xsl:text>
298               </xsl:otherwise>
299            </xsl:choose>
300         </xsl:when>
301         <xsl:when test="@clef.shape = 'perc'">
302            <xsl:text>drum</xsl:text>
303         </xsl:when>
304         <xsl:otherwise>
305            <xsl:text>treble</xsl:text>
306         </xsl:otherwise>
307      </xsl:choose>
308   </xsl:template>
309
310
311   <!-- Music is specified on a bar-by-bar level; there may be multiple staves
312   in a single bar. These should correspond to the <staffdef>'s id attribute.
313   -->
314
315   <!-- Template to extract out a bar -->
316   <xsl:template match="bar">
317      <xsl:if test="@n">
318         <xsl:value-of select="$nl" />
319         <xsl:value-of select="$nl" />
320         <xsl:text>   // begin bar </xsl:text>
321         <xsl:value-of select="@n" />
322      </xsl:if>
323      <xsl:apply-templates />
324      <xsl:value-of select="$nl" />
325      <!-- Possible kinds of barlines: -->
326      <!-- dashed|dotted|dbl|dbldashed|dbldotted|end| -->
327      <!-- invis|rptstart|rptboth|rptend|single (default) -->
328      <xsl:choose>
329         <xsl:when test="@rrend = 'dashed'">
330            <xsl:text>dashedbar</xsl:text>
331         </xsl:when>
332         <xsl:when test="@rrend = 'dotted'">
333            <xsl:text>dottedbar</xsl:text>
334         </xsl:when>
335         <xsl:when test="@rrend = 'dbl'">
336            <xsl:text>dblbar</xsl:text>
337         </xsl:when>
338         <xsl:when test="@rrend = 'dbldashed'">
339            <xsl:text>dasheddblbar</xsl:text>
```

```
340            </xsl:when>
341            <xsl:when test="@rrend = 'dbldotted'">
342               <xsl:text>dotteddblbar</xsl:text>
343            </xsl:when>
344            <xsl:when test="@rrend = 'end'">
345               <xsl:text>endbar</xsl:text>
346            </xsl:when>
347            <xsl:when test="@rrend = 'invis'">
348               <xsl:text>invisbar</xsl:text>
349            </xsl:when>
350            <xsl:when test="@rrend = 'rptstart'">
351               <xsl:text>repeatstart</xsl:text>
352            </xsl:when>
353            <xsl:when test="@rrend = 'rptboth'">
354               <xsl:text>repeatboth</xsl:text>
355            </xsl:when>
356            <xsl:when test="@rrend = 'rptend'">
357               <xsl:text>repeatend</xsl:text>
358            </xsl:when>
359            <xsl:otherwise>
360               <xsl:text>bar</xsl:text>
361            </xsl:otherwise>
362         </xsl:choose>
363      </xsl:template>
364
365      <!-- Template to extract out a stave -->
366      <xsl:template match="staff">
367         <xsl:if test="not(voice)">
368            <xsl:value-of select="$nl" />
369            <!-- As per email, the @def is expected to be 's' + the staff # -->
370            <xsl:value-of select="substring-after(@def, 's')" />
371            <xsl:text> 1: </xsl:text>
372         </xsl:if>
373         <xsl:apply-templates />
374      </xsl:template>
375
376      <xsl:template match="lyrics">
377         <xsl:value-of select="$nl" />
378         <xsl:text>lyrics </xsl:text>
379         <xsl:if test="@place">
380            <xsl:value-of select="@place" />
381            <xsl:text> </xsl:text>
382         </xsl:if>
383         <xsl:value-of select="substring-after(@staff, 's')" />
384         <xsl:text>: </xsl:text>
385         <xsl:if test="@rhy">
386            <xsl:value-of select="@rhy" />
```

```
387       </xsl:if>
388       <xsl:text> "</xsl:text>
389       <xsl:apply-templates />
390       <xsl:text>";</xsl:text>
391    </xsl:template>
392
393    <xsl:template match="syl">
394       <xsl:value-of select="." />
395       <xsl:text> </xsl:text>
396    </xsl:template>
397
398    <!-- Template to extract out a voice -->
399    <xsl:template match="voice">
400       <xsl:value-of select="$nl" />
401       <!-- Get the def of the parent staff element -->
402       <xsl:value-of select="substring-after(ancestor::staff[1]/@def, 's')" />
403       <xsl:text> </xsl:text>
404       <!-- As per email, the @def is expected to be 's#v' + the voice # -->
405       <xsl:value-of select="substring-after(@def, 'v')" />
406       <xsl:text>: </xsl:text>
407       <xsl:apply-templates />
408    </xsl:template>
409
410    <!-- Endings -->
411    <xsl:template match="ending">
412       <xsl:text> ending "</xsl:text>
413       <xsl:value-of select="@label" />
414       <xsl:text>"</xsl:text>
415       <xsl:apply-templates />
416       <!-- If there is no ending immediately after this one, outupt
417       'endending'.  If there is no next node, don't output anything -->
418       <xsl:variable name="next-sibling" select="following-sibling::node()[1]" />
419       <xsl:if test="name($next-sibling) != 'ending'
420                     and following-sibling::node()">
421          <xsl:text> endending</xsl:text>
422       </xsl:if>
423    </xsl:template>
424
425    <!-- Beams can contain note, chord, rest, pad, and tup elements.  The first
426    and last elements are followed, in Mup, by bm and ebm as markers -->
427    <xsl:template match="beam">
428       <xsl:apply-templates>
429
430          <!-- Call the template with a parameter to indicate that it is
431          beamed.  I do this rather than use a template with a mode because the
432          code would be very similar. -->
433          <xsl:with-param name="is-beamed" select="1" />
```

```
      </xsl:apply-templates>
   </xsl:template>

   <xsl:template match="chord">
      <xsl:param name="is-beamed" select="0" />
      <xsl:call-template name="chord-attributes">
         <xsl:with-param name="is-chord" select="1" />
      </xsl:call-template>

      <!-- Call the template with a parameter to indicate that the notes are
      chorded. I do this rather than use a template with a mode because the
      code would be very similar. -->

      <xsl:apply-templates>
         <xsl:with-param name="is-chorded" select="1" />
      </xsl:apply-templates>
      <xsl:if test="$is-beamed = 1">
         <xsl:call-template name="beam-groups" />
      </xsl:if>
      <xsl:text>; </xsl:text>
   </xsl:template>

   <xsl:template match="note">
      <xsl:param name="is-chorded" select="0" />
      <xsl:param name="is-beamed" select="0" />
      <!-- Print out the things that Mup wants before anything else. -->
      <xsl:if test="$is-chorded = '0'">
         <xsl:call-template name="chord-attributes" />
      </xsl:if>

      <!-- The time value only goes on the first note of a chord. If there is
      none, let Mup handle propagation or the default. -->

      <xsl:choose>
         <xsl:when test="$is-chorded = 0">
            <xsl:choose>
               <xsl:when test="@grace">
                  <xsl:value-of select="@dur.vis" />
               </xsl:when>
               <xsl:otherwise>
                  <xsl:value-of select="@dur" />
               </xsl:otherwise>
            </xsl:choose>
            <xsl:if test="@dots">
               <xsl:call-template name="augment-dots">
                  <xsl:with-param name="num" select="@dots" />
               </xsl:call-template>
```

```
481            </xsl:if>
482         </xsl:when>
483         <xsl:otherwise>
484            <xsl:if test="position() = 1">
485               <xsl:value-of select="../@dur" />
486               <xsl:if test="../@dots">
487                  <xsl:call-template name="augment-dots">
488                     <xsl:with-param name="num" select="../@dots" />
489                  </xsl:call-template>
490               </xsl:if>
491            </xsl:if>
492         </xsl:otherwise>
493      </xsl:choose>
494      <!-- The actual name of the note -->
495      <xsl:choose>
496         <xsl:when test="@pname">
497            <xsl:value-of select="@pname" />
498         </xsl:when>
499         <!-- If no note name, look for one to propagate -->
500         <xsl:otherwise>
501            <xsl:if test="preceding-sibling::note[@pname][1]/@pname">
502               <xsl:value-of
503                   select="preceding-sibling::note[@pname][1]/@pname" />
504            </xsl:if>
505         </xsl:otherwise>
506      </xsl:choose>
507      <!-- Optional accidentals go next
508      MEI: s = sharp, f = flat, ss = dblsharp, ff = dblflat, n = natural
509      Mup: # = sharp, & = flat,  x = dblsharp, && = dblflat, n = natural -->
510      <xsl:if test="@acci">
511         <xsl:choose>
512            <xsl:when test="@acci = 's'">
513               <xsl:text>#</xsl:text>
514            </xsl:when>
515            <xsl:when test="@acci = 'f'">
516               <xsl:text>&amp;</xsl:text>
517            </xsl:when>
518            <xsl:when test="@acci = 'ss'">
519               <xsl:text>x</xsl:text>
520            </xsl:when>
521            <xsl:when test="@acci = 'ff'">
522               <xsl:text>&amp;&amp;</xsl:text>
523            </xsl:when>
524            <xsl:when test="@acci = 'n'">
525               <xsl:text>n</xsl:text>
526            </xsl:when>
527         </xsl:choose>
```

```
528         </xsl:if>
529         <!-- Put the octave after the accidental -->
530         <xsl:choose>
531            <xsl:when test="@oct">
532               <xsl:value-of select="@oct" />
533            </xsl:when>
534            <!-- If no octave, look for one to propagate from preceding notes -->
535            <xsl:otherwise>
536               <xsl:if test="preceding-sibling::note[@oct][1]/@oct">
537                  <xsl:value-of select="preceding-sibling::note[@oct][1]/@oct" />
538               </xsl:if>
539            </xsl:otherwise>
540         </xsl:choose>
541
542         <!-- Mup does not care what order the following are in -->
543
544         <!-- Mup recognizes a ~ as a tie; MEI uses the tie attribute with the
545         value being either i=initial, m=medial, and t=terminal. Only i|m need the
546         ~. -->
547
548         <xsl:if test="@tie = 'i' or @tie = 'm'">
549            <xsl:text>~</xsl:text>
550         </xsl:if>
551
552         <!-- Mup's way of specifying which notes to slur to is not easy to
553         implement Todo: this is not done yet... need to find a way to indicate
554         ending notes-->
555         <xsl:if test="@slur = 'i'">
556            <xsl:text>&lt;&gt;</xsl:text>
557            <!-- find the ending note -->
558         </xsl:if>
559         <!-- Add a Mup 'tag' to a note if it has an id -->
560         <xsl:if test="@id and ($is-chorded=0 or position() = last())">
561            <xsl:text> =</xsl:text>
562            <xsl:call-template name="fix-id">
563               <xsl:with-param name="id" select="@id" />
564            </xsl:call-template>
565         </xsl:if>
566
567         <xsl:if test="$is-beamed = 1">
568            <xsl:call-template name="beam-groups" />
569         </xsl:if>
570
571         <!-- If this note is part of a chord, the chord template called this
572         template and will place the closing semi-colon. -->
573
574         <xsl:if test="$is-chorded = 0">
```

```
575            <xsl:text>; </xsl:text>
576         </xsl:if>
577
578         <!-- check for notes that have a special stem position -->
579         <xsl:if test="@stem.pos">
580            <xsl:choose>
581               <xsl:when test="@stem.pos='dnright'">
582                  <xsl:value-of select="$nl" />
583                  <xsl:text>STEMDNRIGHT(</xsl:text>
584                  <xsl:value-of select="@id" />
585                  <xsl:text>,</xsl:text>
586                  <xsl:value-of select="@stem.len" />
587                  <xsl:text>)</xsl:text>
588               </xsl:when>
589            </xsl:choose>
590         </xsl:if>
591      </xsl:template>
592
593      <xsl:template name="fix-id">
594         <xsl:param name="id" />
595         <xsl:if test="not(starts-with($id, '_'))">
596            <xsl:text>_</xsl:text>
597         </xsl:if>
598         <xsl:value-of select="$id" />
599      </xsl:template>
600
601      <xsl:template name="transpose">
602         <xsl:param name="steps" />
603         <xsl:param name="interval" />
604         <xsl:value-of select="$nl" />
605         <xsl:text>   transpose =</xsl:text>
606         <xsl:choose>
607            <xsl:when test="starts-with($steps, '-')">
608               <xsl:text> down </xsl:text>
609            </xsl:when>
610            <xsl:otherwise>
611               <xsl:text> up </xsl:text>
612            </xsl:otherwise>
613         </xsl:choose>
614         <xsl:choose>
615            <xsl:when test="$steps=1">
616               <xsl:if test="$interval=0">
617                  <xsl:text>perfect 1</xsl:text>
618               </xsl:if>
619               <xsl:if test="$interval=1">
620                  <xsl:text>dim 2</xsl:text>
621               </xsl:if>
```

```
622            </xsl:when>
623            <xsl:when test="$steps=2">
624               <xsl:if test="$interval=0">
625                  <xsl:text>aug 1</xsl:text>
626               </xsl:if>
627               <xsl:if test="$interval=1">
628                  <xsl:text>min 2</xsl:text>
629               </xsl:if>
630            </xsl:when>
631            <xsl:when test="$steps=3">
632               <xsl:if test="$interval=1">
633                  <xsl:text>aug 2</xsl:text>
634               </xsl:if>
635               <xsl:if test="$interval=2">
636                  <xsl:text>min 3</xsl:text>
637               </xsl:if>
638            </xsl:when>
639            <xsl:when test="$steps=4">
640               <xsl:if test="$interval=1">
641                  <xsl:text></xsl:text>
642               </xsl:if>
643               <xsl:if test="$interval=2">
644                  <xsl:text></xsl:text>
645               </xsl:if>
646            </xsl:when>
647            <xsl:when test="$steps=5">
648               <xsl:if test="$interval=2">
649                  <xsl:text></xsl:text>
650               </xsl:if>
651               <xsl:if test="$interval=3">
652                  <xsl:text></xsl:text>
653               </xsl:if>
654            </xsl:when>
655            <xsl:when test="$steps=6">
656               <xsl:if test="$interval=2">
657                  <xsl:text></xsl:text>
658               </xsl:if>
659               <xsl:if test="$interval=3">
660                  <xsl:text></xsl:text>
661               </xsl:if>
662            </xsl:when>
663            <xsl:when test="$steps=7">
664               <xsl:if test="$interval=3">
665                  <xsl:text></xsl:text>
666               </xsl:if>
667               <xsl:if test="$interval=4">
668                  <xsl:text></xsl:text>
```

```
669            </xsl:if>
670         </xsl:when>
671         <xsl:when test="$steps=8">
672            <xsl:if test="$interval=3">
673               <xsl:text></xsl:text>
674            </xsl:if>
675            <xsl:if test="$interval=4">
676               <xsl:text></xsl:text>
677            </xsl:if>
678         </xsl:when>
679         <xsl:when test="$steps=9">
680            <xsl:if test="$interval=4">
681               <xsl:text></xsl:text>
682            </xsl:if>
683            <xsl:if test="$interval=5">
684               <xsl:text></xsl:text>
685            </xsl:if>
686         </xsl:when>
687         <xsl:when test="$steps=10">
688            <xsl:if test="$interval=4">
689               <xsl:text></xsl:text>
690            </xsl:if>
691            <xsl:if test="$interval=5">
692               <xsl:text></xsl:text>
693            </xsl:if>
694         </xsl:when>
695         <xsl:when test="$steps=11">
696            <xsl:if test="$interval=5">
697               <xsl:text></xsl:text>
698            </xsl:if>
699            <xsl:if test="$interval=6">
700               <xsl:text></xsl:text>
701            </xsl:if>
702         </xsl:when>
703      </xsl:choose>
704   </xsl:template>
705
706   <!-- This is a skeleton. Todo: make it do chord attributes correctly -->
707   <xsl:template name="chord-attributes">
708      <xsl:param name="is-chord" select="0" />
709      <!-- make a string to hold values I need to put into the [] list -->
710      <xsl:variable name="attributes">
711         <xsl:if test="@id and $is-chord=1">
712            <xsl:text>=</xsl:text>
713            <xsl:value-of select="@id" />
714            <xsl:text>;</xsl:text>
715         </xsl:if>
```

```
716         <xsl:if test="@artic = 'wedge'">
717            <xsl:text>with "\(wedge)";</xsl:text>
718         </xsl:if>
719         <xsl:if test="@artic = '.'">
720            <xsl:text>with .;</xsl:text>
721         </xsl:if>
722         <xsl:if test="@grace">
723            <xsl:text>grace;</xsl:text>
724         </xsl:if>
725         <xsl:if test="@artic='ferm'">
726            <xsl:text>with "\(ferm)";</xsl:text>
727         </xsl:if>
728         <xsl:if test="@stem.len">
729               <xsl:text>len </xsl:text>
730               <xsl:choose>
731                  <xsl:when test="@stem.pos='dnright'">
732                     <xsl:text>0</xsl:text>
733                  </xsl:when>
734                  <xsl:otherwise>
735                     <xsl:value-of select="@stem.len" />
736                  </xsl:otherwise>
737               </xsl:choose>
738               <xsl:text>;</xsl:text>
739         </xsl:if>
740         <xsl:if test="@stem.dir">
741               <xsl:value-of select="@stem.dir" />
742               <xsl:text>;</xsl:text>
743         </xsl:if>
744         <xsl:if test="@size = 'cue'">
745            <xsl:text>cue;</xsl:text>
746         </xsl:if>
747      </xsl:variable>
748      <xsl:if test="$attributes != ''">
749         <xsl:text>[</xsl:text>
750         <xsl:value-of
751         select="substring($attributes, 1, string-length($attributes) - 1)" />
752         <xsl:text>]</xsl:text>
753      </xsl:if>
754   </xsl:template>
755
756   <xsl:template match="rest">
757      <xsl:value-of select="@dur" />
758      <xsl:if test="@dots">
759         <xsl:call-template name="augment-dots">
760            <xsl:with-param name="num" select="@dots" />
761         </xsl:call-template>
762      </xsl:if>
```

```
763         <xsl:text>r; </xsl:text>
764      </xsl:template>
765
766      <xsl:template match="msrest">
767         <xsl:text>mr; </xsl:text>
768      </xsl:template>
769
770      <xsl:template match="space">
771         <xsl:value-of select="@dur" />
772         <xsl:text>s; </xsl:text>
773      </xsl:template>
774
775      <xsl:template match="tup">
776         <xsl:text>{</xsl:text>
777         <xsl:apply-templates />
778         <xsl:text>} </xsl:text>
779         <xsl:value-of select="@num.place" />
780         <xsl:text> </xsl:text>
781         <xsl:value-of select="count(note|chord|rest)" />
782         <xsl:text>;</xsl:text>
783      </xsl:template>
784
785      <!-- These similar constructs should be handled by the same code according
786      to p.80 of the Mup manual ("Tempo, Dynamic Marks, Ornaments, etc") -->
787
788      <xsl:template match="phrase">
789         <xsl:value-of select="$nl" />
790         <xsl:choose>
791            <xsl:when test="@tstamp">
792               <xsl:text>phrase </xsl:text>
793               <xsl:if test="@place">
794                  <xsl:value-of select="@place" />
795                  <xsl:text> </xsl:text>
796               </xsl:if>
797               <xsl:value-of select="substring-after(@staff, 's')" />
798               <xsl:text>:</xsl:text>
799               <xsl:value-of select="@tstamp" />
800               <xsl:text> til </xsl:text>
801               <xsl:value-of select="@dur" />
802            </xsl:when>
803            <xsl:when test="@end1">
804               <xsl:text>medium curve (</xsl:text>
805               <xsl:call-template name="fix-id">
806                  <xsl:with-param name="id" select="@end1" />
807               </xsl:call-template>
808               <xsl:text>) to (</xsl:text>
809               <xsl:call-template name="fix-id">
```

```
               <xsl:with-param name="id" select="@end2" />
            </xsl:call-template>
            <xsl:text>) bulge </xsl:text>
            <xsl:value-of select="@bulge" />
         </xsl:when>
      </xsl:choose>
      <xsl:text>;</xsl:text>
   </xsl:template>

   <xsl:template match="dyn">
      <xsl:value-of select="$nl" />
      <xsl:text>boldital </xsl:text>
      <xsl:if test="@place">
         <xsl:value-of select="@place" />
         <xsl:text> </xsl:text>
      </xsl:if>
      <xsl:value-of
         select="translate(substring-after(@staff, 's'), 's', ',')" />
      <xsl:text>:</xsl:text>
      <xsl:value-of select="@tstamp" />
      <xsl:text> &quot;</xsl:text>
      <xsl:value-of select="." />
      <xsl:text>&quot;;</xsl:text>
   </xsl:template>

   <xsl:template match="arpeg">
      <xsl:value-of select="$nl" />
      <xsl:text>roll </xsl:text>
      <xsl:choose>
      <!-- select which staff and voice the roll covers -->
         <xsl:when test="contains(@staff, ' ') or contains(@voice, ' ')">
            <!-- It covers multiple staffs or voices -->
            <!-- select the staff and voice it starts on -->
            <xsl:value-of
               select="substring-before(substring-after(@staff, 's'), ' ')" />
            <xsl:text> </xsl:text>
            <xsl:value-of
               select="substring-before(substring-after(@voice, 'v'), ' ')" />
            <xsl:text> to </xsl:text>
            <!-- select the staff and voice it ends on -->
            <xsl:value-of select="substring-after(@staff, ' s')" />
            <xsl:text> </xsl:text>
            <xsl:value-of
               select="substring-after(substring-after(@voice, ' s'), 'v')" />
            <!-- select the staff and voice it ends on -->
         </xsl:when>
         <xsl:otherwise>
```

```
            <xsl:value-of
               select="substring-before(substring-after(@voice, 's'), 'v')" />
            <xsl:text> </xsl:text>
            <xsl:value-of select="substring-after(@staff, 'v')" />
         </xsl:otherwise>
      </xsl:choose>
      <xsl:text>: </xsl:text>
      <xsl:value-of select="@tstamp" />
      <xsl:text>;</xsl:text>
   </xsl:template>

   <xsl:template match="tempo">
      <xsl:value-of select="$nl" />
      <xsl:text>boldital </xsl:text>
      <xsl:if test="@place">
         <xsl:value-of select="@place" />
         <xsl:text> </xsl:text>
      </xsl:if>
      <xsl:variable name="str">
         <xsl:value-of select="substring-after(@staff, 's')" />
      </xsl:variable>
      <xsl:value-of select="translate($str, 's', ',')" />
      <xsl:text>:</xsl:text>
      <xsl:value-of select="@tstamp" />
      <xsl:text> &quot;</xsl:text>
      <xsl:value-of select="." />
      <xsl:text>&quot;;</xsl:text>
   </xsl:template>

   <!-- Templates to encapsulate common functionality -->

   <!-- Print out augmentation dots, which can appear on notes, rests, etc -->
   <xsl:template name="augment-dots">
      <xsl:param name="num" />
      <xsl:if test="$num != 0">
         <xsl:text>.</xsl:text>
         <xsl:call-template name="augment-dots">
            <xsl:with-param name="num" select="$num - 1" />
         </xsl:call-template>
      </xsl:if>
   </xsl:template>

   <!-- Print out the beginning and end of a beam group. This entails placing a
   space, followed by 'bm' to start a beam and 'ebm' to end it. -->

   <xsl:template name="beam-groups">
      <xsl:if test="position() = 1">
```

```
904            <xsl:text> bm</xsl:text>
905         </xsl:if>
906         <xsl:if test="position() = last()">
907            <xsl:text> ebm</xsl:text>
908         </xsl:if>
909      </xsl:template>
910
911   </xsl:stylesheet>
```

Appendix I

Code Listing: Mary Had a Little Lamb

This appendix includes a code listing of "Mary Had a Little Lamb," the simplest piece of music encoded for this project. To see the resulting notation , refer to Figure 6.3 on page 44.

```
<?xml version="1.0"?>
<!DOCTYPE mei SYSTEM "http://dl.lib.virginia.edu/bin/dtd/mei/mei.dtd" [
]>
<?xml-stylesheet type='text/xsl' href='mei-mup.xslt' ?>
<mei>
<meihead>
   <meiid>MaryHadaLittleLamb</meiid>
   <filedesc>
      <pubstmt>
         <agent>Perry Roland</agent>
         <rights>Electronic edition copyright &copy; 2002 Perry Roland.
                 All rights reserved.</rights>
      </pubstmt>
   </filedesc>
</meihead>
<body>
   <mdiv type="children's song">
      <score>
         <scoredef meter.count="4" meter.unit="4" key.tonic="Eb"
         key.mode="major" key.sig="3f" beam.group="4,4,4,4">
            <staffdef id="s1" octave.default="4" />
         </scoredef>
```

```
23          <bar n="1">
24             <staff def="s1">
25                <!-- accidentals required by the key signature not encoded. -->
26                <note pname="g" dur="4" dots="1" />
27                <note pname="f" dur="8" />
28                <note pname="e" dur="4" />
29                <note pname="f" dur="4" />
30             </staff>
31          </bar>
32          <bar n="2">
33             <staff def="s1">
34                <note pname="g" dur="4" />
35                <note pname="g" dur="4" />
36                <note pname="g" dur="2" />
37             </staff>
38          </bar>
39          <bar n="3">
40             <staff def="s1">
41                <note pname="f" dur="4" />
42                <note pname="f" dur="4" />
43                <note pname="f" dur="2" />
44             </staff>
45          </bar>
46          <bar n="4">
47             <staff def="s1">
48                <note pname="g" dur="4" />
49                <note pname="b" dur="4" />
50                <note pname="b" dur="2" />
51             </staff>
52          </bar>
53          <bar n="5">
54             <staff def="s1">
55                <note pname="g" dur="4" dots="1" />
56                <note pname="f" dur="8" />
57                <note pname="e" dur="4" />
58                <note pname="f" dur="4" />
59             </staff>
60          </bar>
61          <bar n="6">
62             <staff def="s1">
63                <note pname="g" dur="4" />
64                <note pname="g" dur="4" />
65                <note pname="g" dur="4" />
66                <note pname="g" dur="4" />
67             </staff>
68          </bar>
69          <bar n="7">
```

```
70            <staff def="s1">
71               <note pname="f" dur="4" />
72               <note pname="f" dur="4" />
73               <note pname="g" dur="4" />
74               <note pname="f" dur="4" />
75            </staff>
76         </bar>
77         <bar n="8" rrend="dbl">
78            <staff def="s1">
79               <note pname="e" dur="2" dots="1" />
80               <rest dur="4" />
81            </staff>
82         </bar>
83         <!-- written-out repeat, double time -->
84         <bar n="9">
85            <staff def="s1">
86               <note pname="g" dur="8" dots="1" />
87               <note pname="f" dur="16" />
88               <note pname="e" dur="8" />
89               <note pname="f" dur="8" />
90               <note pname="g" dur="8" />
91               <note pname="g" dur="8" />
92               <note pname="g" dur="4" />
93            </staff>
94         </bar>
95         <bar n="10">
96            <staff def="s1">
97               <note pname="f" dur="8" />
98               <note pname="f" dur="8" />
99               <note pname="f" dur="4" />
100              <note pname="g" dur="8" />
101              <note pname="b" dur="8" />
102              <note pname="b" dur="4" />
103           </staff>
104        </bar>
105        <bar n="11">
106           <staff def="s1">
107              <note pname="g" dur="8" dots="1" />
108              <note pname="f" dur="16" />
109              <note pname="e" dur="8" />
110              <note pname="f" dur="8" />
111              <note pname="g" dur="8" />
112              <note pname="g" dur="8" />
113              <note pname="g" dur="8" />
114              <note pname="g" dur="8" />
115           </staff>
116        </bar>
```

```
         <bar n="12" rrend="end">
            <staff def="s1">
               <!-- explicit beam which overrides the beam.group attribute -->
               <beam>
               <note pname="f" dur="8" />
               <note pname="f" dur="8" />
               <note pname="g" dur="8" />
               <note pname="f" dur="8" />
               </beam>
               <note pname="e" dur="4" tie="i" />
               <note pname="e" dur="8" tie="t" />
               <rest dur="8" />
            </staff>
         </bar>

      </score>
   </mdiv>
</body>
</mei>
```

References

[1] *The Apache XML Project* (n.d.). Retrieved March 10, 2003 from http://xml.apache.org.

[2] *Website for Beyond Midi: The Handbook of Musical Codes*. (n.d.). Retrieved April 7, 2003 from http://www.ccarh.org/publications/books/beyondmidi/.

[3] Bourret, Ronald. *rpbourret.com – XML programming, writing, and research* (n.d.). Retrieved April 10, 2003 from http://www.rpbourret.com/.

[4] Brabec, Jeffrey and Todd Brabec. *Music, Money, and Success*. New York: Schirmer, 1994.

[5] Covey, Stephen R. *The 7 Habits of Highly Effective People*. New York: Simon & Schuster, 1989.

[6] *Cascading Style Sheets* (n.d.). Retrieved March 10, 2003 from http://www.w3.org/Style/CSS/.

[7] *CVS: Concurrent Versions System* (n.d.). Retrieved March 10, 2003 from http://cvshome.org.

[8] Roger B. Dannenberg: "The Canon Score Language," *Computer Music Journal*, 12:1. Spring 1989. 47-56.

[9] *The WC3 Document Object Model homepage* (n.d.). Retrieved March 10, 2003 from http://www.w3.org/DOM/.

[10] Downie, J. Stephen. 2003. "Music information retrieval," *Annual Review of Information Science and Technology*, 37: 295-340. Available from http://music-ir.org/downie_mir_arist37.pdf

[11] *Extensible Markup Language (XML)* (n.d.). Retrieved March 10, 2003 from http://www.w3.org/XML/.

[12] *Ghostscript, Ghostview and GSview* (n.d.) Retrieved March 10, 2003 from http://www.cs.wisc.edu/ ghost/.

[13] *GNU: GNU's Not Unix* (n.d.). Retrieved March 10, 2003 from http://www.gnu.org.

[14] Good, Michael. "MusicXML for Notation and Analysis." *The Virtual Score: Representation, Retrieval, Restoration*. Ed. Hewlett, Walter B., and Eleanor Selfridge-Field. Cambridge: MIT Press, 2001. 113-124.

[15] Gourlay, John. "Spacing a Line of Music." OSU-CISRC-10/87-TR35. Department of Computer and Information Science. The Ohio State University. 1987.

[16] Hegazy, Wael A. and John S. Gourlay. "Optimal Line Breaking in Music." OSU-CISRC-8/87-TR33. Department of Computer and Information Science, The Ohio State University. 1987.

[17] Hewlett, Walter B. "MuseData: Multipurpose Representation." Selfridge-Field, Eleanor, Ed. *Beyond MIDI: The Handbook of Musical Codes*. Cambridge: MIT Press, 1997. 402-445.

[18] Hewlett, Walter B., Eleanor Selfridge-Field, et. al. "MIDI." ibid. 41-72.

[19] Howard, John. "Plaine and Easie Code: A Code for Music Bibliography." ibid. 362-371.

[20] *HTML 4.01 Specification* (n.d.). Retrieved March 10, 2003 from http://www.w3.org/TR/html4/.

[21] Huron, David. "Humdrum and Kern: Selective Feature Encoding." ibid. 375-398.

[22] Huron, David. "Design Principles in Computer-based Music Representation." *Computer Representations and Models of Music*. Alan Marsden and Anthony Pople, eds. New York: Academic Press, 1992.

[23] *The Humdrum Toolkit: Software for Music Research* (n.d.). Retrieved March 10, 2003 from http://www.music-cog.ohio-state.edu/Humdrum/.

[24] Kay, Michael. *XSLT Programmer's Reference, 2nd Edition*. Birmingham, UK: Wrox Press, 2001.

[25] *The K Desktop Environment* (n.d.). Retrieved March 10, 2003 from http://www.kde.org.

[26] *LaTeX Typesetting system* (n.d.). Retrieved March 10, 2003 from http://www.latex-project.org/.

[27] *Libxml: The XML C library for Gnome* (n.d.). Retrieved March 10, 2003 from http://xmlsoft.org/.

[28] *GNU LilyPond* (n.d.). Retrieved March 10, 2003 from http://www.lilypond.org/.

[29] *Linux: Linux Is Not UniX* (n.d.). Retrieved March 10, 2003 from http://www.linux.org.

[30] Martin, Worthy. Private communication. 2001-2003.

[31] Mosterd, Eric. *Developing A New Way To Transfer Sheet Music Via The Internet*. 1999. Retrieved April 7, 2003 from University of South Dakota Web site: http://www.usd.edu/csci/research/theses/graduate/sp2001/emosterd.pdf.

[32] *The Music Encoding Initiative* (n.d.). Retrieved April 7, 2003 from http://dl.lib.virginia.edu/bin/dtd/mei/.

[33] *MrProject* (n.d.). Retrieved April 7, 2003 from http://mrproject.codefactory.se/.

[34] *Mup: Music Publisher* (n.d.). Retrieved April 7, 2003 from http://www.arkkra.com.

[35] Mup developers. Private communication. 2002-2003.

[36] *MusicXML Definition* (n.d.). Retrieved April 7, 2003 from http://www.recordare.com/xml.html.

[37] *MusiXTEX* (n.d.). Retrieved April 7, 2003 from http://icking-music-archive.sunsite.dk/software/indexmt6.html

[38] National Music Publisher's Association (NMPA). *Tenth Annual International Survey of Music Publishing Revenues*. 1994. Retrieved April 7, 2003 from http://www.nmpa.org/nmpa/survey10/base.html.

[39] *NoteEdit: — Musical Score Editor* (n.d.). Retrieved April 7, 2003 from http://rnvs.informatik.tu-chemnitz.de/ jan/noteedit/noteedit.html.

[40] *OASIS Open Office XML Format TC* (n.d.). Retrieved April 7, 2003 from http://www.oasis-open.org/committees/office/.

[41] *OFX: Open Financial Exchange* (n.d.). Retrieved April 7, 2003 from http://www.ofx.net/ofx/default.asp.

[42] *OpenType* (n.d.). Retrieved April 7, 2003 from http://www.adobe.com/type/opentype/main.html.

[43] Parish, Allen, Wael A. Hegazy, John S. Gourlay, Dean K. Roush, and F. Javier Sola. "MusiCopy: An Automated Music Formatting System." Department of Computer and Information Science, The Ohio State University. 1987.

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

[45] Renz, Kai and Holger H. Hoos. *GUIDO/MIR - an Experimental Musical Information Retrieval System based on GUIDO Music Notation*. Proceedings of the 2nd International Symposium on Music Information Retrieval. 2001.

[46] Roland, Perry. "MDL and MusiCat: An XML Approach to Musical Data and Meta-Data." *The Virtual Score: Representation, Retrieval, Restoration*. Ed. Hewlett, Walter B., and Eleanor Selfridge-Field. Cambridge: MIT Press, 2001. 125-134.

[47] Roland, Perry. *The Music Encoding Initiative (MEI)*. Proceedings of the First International Conference on Musical Applications Using XML. 2002.

[48] Roland, Perry. Private communications. 2002-2003.

[49] *SAX: the Simple API for XML* (n.d.). Retrieved April 7, 2003 from http://www.saxproject.org/.

[50] Schottstaedt, Bill. "MuTEX, MusicTEX, and MusiXTEX." Selfridge-Field, Eleanor, Ed. *Beyond MIDI: The Handbook of Musical Codes*. Cambridge: MIT Press, 1997. 217-221.

[51] Selfridge-Field, Eleanor. "Introduction: Describing Musical Information." ibid. 3-37.

[52] Selfridge-Field, Eleanor. "DARMS, Its Dialects, and Its Uses." ibid. 163-172.

[53] *Music Notation Software - Sibelius* (n.d.). Retrieved April 7, 2003 from http://www.sibelius.com/.

[54] *Database System Concepts, Fourth Edition*. Silberschatz, Abraham, Henry F. Korth, and S. Sudarshan. Boston: McGraw-Hill, 2002.

[55] Sloan, Donald and Steven R. Newcomb. "HyTime and Standard Music Description Language: A Document-Description Approach." ibid. 469-489.

[56] Sola, F. Javier and Dean K. Roush. "Design of Musical Beams." OSU-CISRC-10/87-TR30. Department of Computer and Information Science. The Ohio State University. 1987.

[57] *Standard Music Description Language (SMDL). ISO/IEC DIS 10743* (n.d.). Retrieved April 7, 2003 from ftp://ftp.ornl.gov/pub/sgml/WG8/SMDL/10743.pdf.

[58] Clark, James. *Comparison of SGML and XML*. 1997. Retrieved April 7, 2003 from http://www.w3.org/TR/NOTE-sgml-xml.html.

[59] *Sunhawk.com - Digital sheet music on-line* (n.d.). Retrieved April 7, 2003 from http://www.sunhawk.com/.

[60] *The Text Encoding Initiative (TEI)* (n.d.). Retrieved April 7, 2003 from http://www.tei-c.org/.

[61] Sperberg-McQueen, C.M.. and Burnard, L. (eds.) (2002). *TEI P4: Guidelines for Electronic Text Encoding and Interchange*. Text Encoding Initiative Consortium. XML Version: Oxford, Providence, Charlottesville, Bergen.

[62] Tidwell, Doug. *XSLT: Mastering XML Transformations*. Cambridge: O'Reilly, 2001.

[63] *Scalable Vector Graphics (SVG) 1.0 Specification* (n.d.). Retrieved April 7, 2003 from http://www.w3.org/TR/SVG/.

[64] *Vim: Vi IMproved* (n.d.). Retrieved April 7, 2003 from http://www.vim.org.

[65] Wall, Larry et al. *Programming Perl, Third Edition*. Cambridge: O'Reilly, 2000.

[66] *World Wide Web Consortium* (n.d.). Retrieved April 7, 2003 from http://www.w3.org/.

[67] Walmsley, Priscilla. *Definitive XML Schema*. New Jersey: Prentice-Hall, 2002.

[68] *XFIG Drawing Program for the X Window System* (n.d.). Retrieved April 7, 2003 from http://www.xfig.org.

[69] *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)* (n.d.). Retrieved April 7, 2003 from http://www.w3.org/TR/xhtml1/.

[70] *XML Path Language (XPath) Version 1.0* (n.d.). Retrieved April 7, 2003 from http://www.w3.org/TR/xpath.

[71] *XSL Transformations (XSLT)* (n.d.). Retrieved April 7, 2003 from http://www.w3.org/TR/xslt.