A FRAMEWORK FOR THE REAL-TIME ANALYSIS OF MUSICAL EVENTS

By

John Bryan Ibbotson

A thesis submitted for the degree of Doctor of Philosophy

School of Electronics and Computer Science,

University of Southampton,

United Kingdom.

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS

SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

A FRAMEWORK FOR THE REAL-TIME ANALYSIS OF MUSICAL EVENTS

by John Bryan Ibbotson

In this thesis I propose a framework for the real-time creation of a harmonic structural model of music. Unlike most uses of computing in musicology which are based on batch processing, the framework uses publish/subscribe messaging techniques found in business systems to create an interconnected set of collaborating applications within a network that process streamed events of the kind generated during a musical performance. These applications demonstrate the transformation of data in the form of MIDI commands into information and knowledge in the form of the music's harmonic structure represented as a model using semantic web techniques.

With such a framework, collaborative performances over the network become possible with a shared representation of the music being performed accessible to all performers both human and potentially software agents. The framework demonstrates novel real-time implementations of pitch spelling, chord and key extraction algorithms interacting with semantic web and database technologies in a collaborative manner. It draws on relevant research in information science, musical cognition, semantic web and business messaging technologies to implement a framework and set of software components for the real-time analysis of musical events, the output of which is a description of the music's harmonic structure. Finally, it proposes a pattern based approach to querying the generated model which suggests a visual query and navigation paradigm.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor Prof. Dave De Roure for his supervision, encouragement and direction over the last seven years. Other members of staff at Southampton University I have worked with have always been encouraging and supportive of my endeavours.

I'd also like to thank colleagues and friends at IBM Hursley for their support, advice and comments whilst my research progressed. In particular Colin Bird for his unofficial mentoring.

Finally, my thanks go to my wife Sharon and children Sophie and Will who have put up with my decision to start a PhD in the first place. The incentive of becoming a Doctor before one's children is a powerful one; particularly when one of them is studying medicine.

Chapter 1 Introduction

Where is the Life we have lost in living?

Where is the wisdom we have lost in knowledge?

Where is the knowledge we have lost in information?

T. S. Eliot. “The Rock”, Faber and Faber, 1934

1.1 Motivation

Writing in 1934, T. S. Eliot could not have anticipated the debates triggered by his poem “The Rock” that have taken place in the Information Science research community. In the lines quoted above, he links wisdom to knowledge and knowledge to information with the suggestion that they are interdependent and interlinked. However, in spite of the substantial research activities that have taken place in the intervening years, the terms wisdom, knowledge and information and the relationships between them are still unclear and often misunderstood.

Information enrichment, or the extraction of information and knowledge from low level data, is a skill that musicians and musicologists apply to the performance and analysis of music; musicians to inform their performance and musicologists to understand the structure of music. In this thesis, I intend to investigate the creation of knowledge in the form of a semantic model of musical harmony from low level data using MIDI commands streamed during a musical performance either live or replayed from a stored file. This motivation leads to the application of algorithmic, semantic and messaging technologies in the enrichment of musical data in the form of MIDI message streams into a semantic representation of musical knowledge in the form of an ontology that models harmonic structure.

Real-time in the context of this thesis means that there is minimal delay introduced by processing applications within the proposed framework. Unlike conventional music analysis applications which operate on a batch processing principle, the application architecture developed in this thesis responds to events propagated throughout a network by a messaging system. These events trigger analysis applications which in turn contribute other events. The structure of the analysis algorithms is such that any processing delays are minimal.

Introducing an event messaging infrastructure to interconnect algorithmic and semantic software applications within a network leads to a number of advanced motivational use cases:

1. A shared knowledge of the structure of a performance amongst collaborating performers interconnected via a network. Performers may include both human and software agents that improvise in a particular musical style. This leads to the concept of a musical Turing test to identify which performers are human and which are machines.
2. A more substantial set of encoded music information retrieval (MIR) datasets for research in musicology. Currently, these datasets are created manually by expert musical researchers. If the automated generation of harmonic structure from performance data is possible, then the creation of these datasets becomes easier.
3. Note extraction from digital audio, though not within the scope of this thesis, would allow additional publishing applications to contribute to the framework. This then makes audio files available as sources of data for generating harmonic structural descriptions.

A former colleague at IBM who was working with digital library technologies in the 1980s added an extra line to the T. S. Eliot poem:

Where is the information we have lost in the library?

But that’s another story.

1.2 Contributions

This thesis proposes an analysis framework based on publish/subscribe messaging techniques commonly found in business quality messaging systems. The musical events discussed in this thesis form a particular class of message event that may be processed and analysed using applications built onto such a distributed framework.

In particular, the research contributions developed in this thesis are:

1. The use of publish/subscribe messaging technology to create a framework where independent, collaborating applications can be developed for the real-time analysis of musical events.
2. An implementation of an event based version of Chew's spiral array note naming algorithm with improved results over those previously published.
3. A musical key and chord extraction algorithm using sampling based on metrical analysis.
4. An ontology is presented that models musical harmony.
5. A visual pattern query paradigm is presented that allows users to construct queries that may be applied to both relational and semantic models of the harmonic structure.

1.3 Thesis Structure

Following this chapter, chapter 2 provides an overview of background work in information theory and musical cognition. Chapter 3 introduces the publish/subscribe messaging model and compares it with other messaging patterns. This is followed by example applications of the model to MIDI command streams. Chapter 4 reviews current research in note naming followed by the development of an event based implementation of Chew's spiral array note naming algorithm. Chapter 5 reviews key, chord and meter extraction research followed by their application in a chord and key extraction component of the framework. Chapter 6 describes the development of the Harmony ontology for representing musical harmonic structure. Chapter 7 discusses the navigation of the harmonic structure model using a visual pattern query which can be transformed to either a relational or semantic database representation of the model. Chapter 8 provides conclusions and identifies areas for further research.

1.4 Declaration

I declare that this thesis and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as:
    - Ibbotson, J., DeRoure D. (2004). Record and Reuse using Publish/Subscribe Messaging. Equator Record and Reuse Workshop, February 2004. www.crg.cs.nott.ac.uk/~sdb/r&rworkshop/ibbotson.pdf
    - Ibbotson, J. (2004). Using publish/subscribe messaging to distribute MIDI commands, December 2004. Available from http://www.ibm.com/developerworks/websphere/library/techarticles/0412_ibbotson/0412_ibbotson.html

Chapter 2 Background

2.1 Introduction

This chapter discusses background research work related to the topic of this thesis. It starts with a discussion of the DIKW hierarchy; a “philosophical framework” for information enrichment which sets the scene for the work in this thesis. Finally, it reviews issues related to musical cognition; in particular its position as a branch of cognitive psychology which leads to the identification of characteristics that are important in the human brain’s perception and understanding of music.

This chapter provides an overview of background research work. Subsequent chapters will provide a more detailed review of current research work related to the components of the architectural framework described in this thesis.

2.2 The DIKW Hierarchy

Recent research into the meaning of data, information, knowledge and wisdom is based on Russell Ackoff’s Presidential address to the ISGSR in June 1988 [Ackoff89] in which he discusses wisdom and what an information system that could generate it would look like. He describes wisdom as being located at the top of a hierarchy of *types of content of the human mind*; descending from wisdom, are understanding, knowledge, information and at the bottom, data. He observes that each level in the hierarchy includes the categories that lie below it. For example, there can be no wisdom without understanding and no understanding without knowledge. Ackoff provides the following definitions of data, information, knowledge and wisdom:

Data are symbols that represent properties of objects, events and their environmental context. They are products of observation or sensing but are of no use until they are in a useable or relevant form; the difference between data and information is functional not structural.

Information is inferred from data and is contained in descriptions and answers to questions that begin with such words as who, what, when and how many.

Knowledge is know-how and makes possible the transformation of information into instructions. It is obtained either by transmission from another who has it, by instruction, or by extracting it from experience.

Wisdom is the ability to increase effectiveness by adding value through the mental function we call judgement. The ethical and aesthetic values that this implies are inherent to humans and are unique and personal.

In parallel with the definition of wisdom, Ackoff defines intelligence as the ability to increase effectiveness. Note that there seems to be no consensus on whether wisdom and intelligence are synonymous. In Turing's classic paper on machine intelligence [Turing50], he refers to the *Argument from Consciousness* proposed by Professor Jefferson from his 1949 Lister Oration:

"Not until a machine can write a Sonnet or compose a concerto because of thoughts and emotions felt, and not by the chance fall of symbols, could we agree that machine equals brain-that is, not only write it but know that it had written it. No mechanism could feel (and not merely artificially signal, an easy contrivance) pleasure at its successes, grief when its valves fuse, be warmed by flattery, be made miserable by its mistakes, be charmed by sex, be angry or depressed when it cannot get what it wants."

The aesthetic nature of the quotation from Lister would seem to be more closely linked to Ackoff's definition of wisdom rather than Turing's views on intelligence.

In conclusion, Ackoff believes that wisdom is grounded on ethical and aesthetic values that are uniquely human and lead to the pursuit of ideals; characteristics that differentiate man from machines. Therefore he concludes that machine automata will never be able to generate wisdom; though the creation of *actionable information* or knowledge by machines is a realistic expectation.

Awad and Ghaziri [Awad04] amongst others consider the inclusion of understanding in Ackoff's hierarchy as un-necessary. The Data, Information, Knowledge, Wisdom (DIKW) hierarchy is illustrated as a pyramid in Figure 1 with data as the base. There is a common view that the higher elements in the hierarchy can be explained in terms of the lower elements by identifying appropriate transformation processes. A research challenge is still to understand and explain how these transformations can be described. Rowley [Rowley07] in her review of representations of the DIKW hierarchy suggests that an alternative representation would be to invert the pyramid illustrating that data becomes increasingly more concentrated and ultimately becomes wisdom.

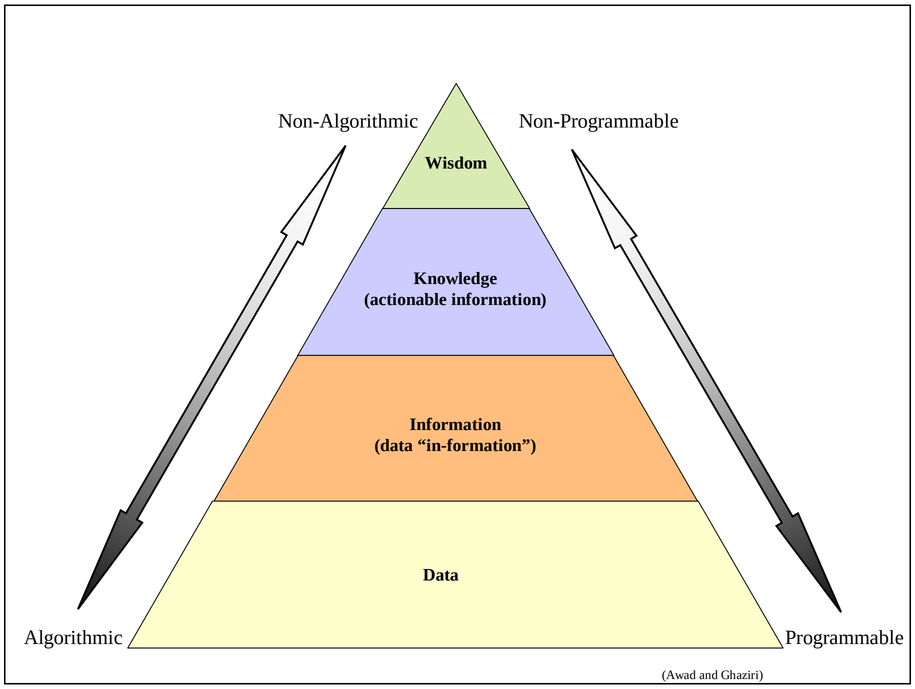


Figure 1: The DIKW Hierarchy

Bellinger et al [Bellinger07] suggest that understanding is not a separate level as proposed by Ackoff but is the process of transformation between layers of the DIKW hierarchy. They suggest that transforming data to information involves an understanding of the relationships that exist in the data. Similarly, knowledge is an understanding of the patterns that exist in information and wisdom is an understanding of the principles that exist in knowledge. Figure 2 illustrates Bellinger's proposition that understanding increases as the connectedness between objects in the hierarchy increases. It also includes Rowley's categorization of information system types that correspond to levels in the DIKW hierarchy.

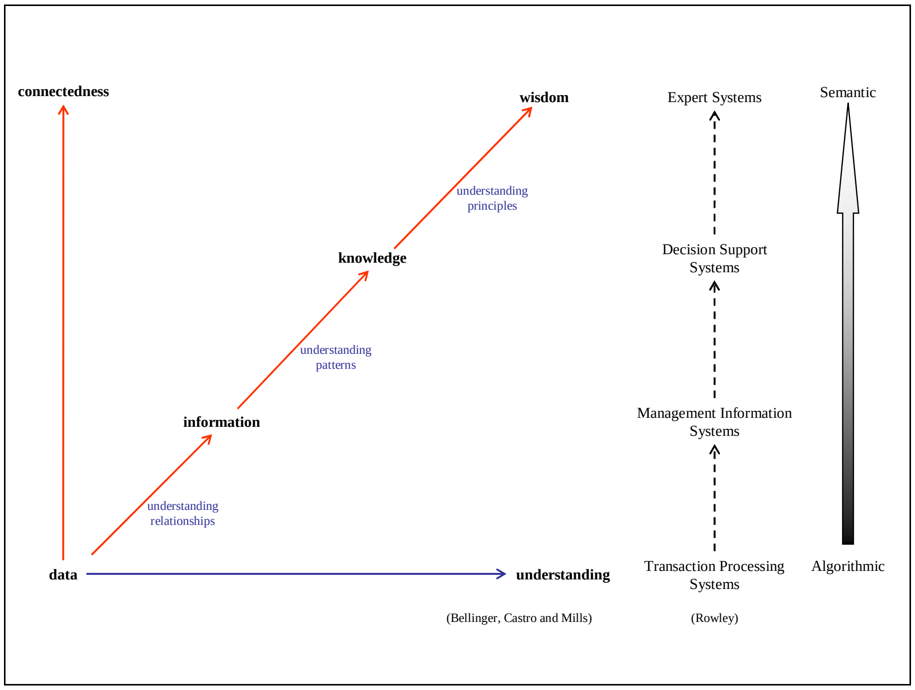


Figure 2: Transformation through Understanding

It is clear from this discussion of the DIKW hierarchy that the nature of the transformations moving up the hierarchy changes. Transformation of data into information is typically accomplished by algorithmic processing whereas the creation of knowledge is achieved by mapping facts into a model that represents the domain of knowledge being considered. This approach is becoming increasingly popular with the development of Semantic Web technologies. Therefore we can assume a spectrum of transformation techniques from the purely algorithmic to the purely semantic as we ascend the hierarchy.

2.3 Musical Cognition

Hargreaves [Hargreaves86] positions musical cognition as a branch of cognitive psychology. He defines this as:

The emphasis of cognitive psychology is upon the internalised rules, operations and strategies that people employ in intelligent behaviour just as much as on the external behavioural manifestations of these processes.

Musical cognition has been likened to information theory where music is compared to the transmitted message that has some degree of uncertainty. In cases of "simple" music, there is little uncertainty and too much redundancy. This type of music is perceived as uninteresting. In cases of complex or randomly generated music with a large amount of uncertainty, the listener perceives it as incomprehensible.

The methodology of cognitive psychology (and therefore musical cognition) is primarily experimental. Subjects are provided with some stimuli and then asked to perform a task or provide a report in response to the stimuli. The psychological processes that take place in the set of subjects are then inferred from the experimental results. Cognitive processes established through experimental procedures may then be modelled computationally. Modelling may be at the neurological level where the biological processes are mimicked in software. Alternatively, higher-level abstract computational models of behaviour may be developed which ignore the detailed neurological processes. Assuming the behavioural models exhibit the same response to stimuli as the neurological models or experimental results, this is a valid approach.

2.3.1 Competence, Knowledge and Language

Musical competence combines knowledge of its mechanisms together with the language of expression. Brinner [Brinner95] defines musical competence as:

Individualised mastery of the array of interrelated skills and knowledge that is required of musicians within a particular tradition or musical community and is acquired and developed in response to and in accordance with the demands and possibilities of general and specific cultural, social and musical conditions

Brinner writes as an ethnomusicologist, so his definition of musical competence embraces anthropological context in addition to purely technical skills. He classifies the knowledge required for musical competence into two forms: declarative and procedural. Declarative knowledge, or “Know that”, consists of hard musical facts such as tuning and scale structure whereas procedural, or “Know how”, consists of the knowledge that lets a musician apply declarative knowledge in a particular musical context. Examples of these knowledge classifications are:

- Procedural
    1. Sound Quality – vocal and instrumental techniques
    2. Sound Patterns - structure
    3. Social, religious, ritual performance contexts
    4. Repertoire and ensemble context, rules of interaction
    5. Meaning and symbolism
    6. Symbolic representation
    7. Transformation e.g. transposition, augmentation, variation
- Declarative
    1. Sound Quality - loudness, timbre, manner of production, and pitch
    2. Sound Patterns - brief ornaments, stock phrases to complete pieces
    3. Orientation - tonal centre, chord progressions, time line, metrical cycle

There is also a view [Sloboda85] that similarities exist between musical structure and natural language. Noam Chomsky, whose work in the 1960s formed the foundation of computational linguistics, believed that all natural languages have the same inherent structure and that understanding this structure informed us about the nature of human intellect. In contrast, Heinrich Schenker an early 20th century musicologist believed that all good musical compositions have the same basic structures and that this structure informed us about the nature of musical intuition. His work became known as *Schenkerian Analysis*, which provided a mathematical technique for identifying fundamental structures in music through a fusion of harmony and counterpoint. Apparently, there is no reference to Schenker within Chomsky’s published work.

Sloboda [Sloboda85] also comments that we cannot assume that music and language can be treated in the same way; that music is not just another form of natural language. The analogies between them can be exploited in metaphysical ways such as “Music is the language of the emotions”, which is not supportable from a scientific viewpoint. He concludes that the analogy is something that can be evaluated, but not assumed. A reasonable conclusion is that there is no problem in adopting techniques developed by language researchers for use in analysing music. If so, techniques such as finite state machines and automata developed for natural language analysis and processing may be investigated for their applicability to music.

The cognitive abilities of the human brain provide clues as to how human musicians understand musical performances.

2.3.2 Musical Cognition

Through research in musical cognition, Dowling [Dowling86] concludes that there are four psychological qualities of sounds that are especially important in music: pitch, duration, loudness and timbre. These fundamental qualities may be combined to form second-order qualities such as rhythm (a combination of duration and loudness). Also, connecting with the views of Schenker on the universality of music, he identifies three underlying properties that are common to all musical traditions:

1. Discrete pitch intervals together with the concept of harmonic tonal centre
2. Octave equivalence
3. The presence of four to seven focal pitches in an octave

These properties and assumptions can form the motivational basis of a framework for the analysis of musical event streams that includes the creation of a description of the music’s harmonic structure..

2.3.3 Musical Analysis

Lerdahl and Jackendoff [Lerdahl83] proposed a set of preference rules for music in their *Generative Theory of Tonal Music* (GTTM). Preference rules are criteria for forming a preferred analysis of some input that may be static or dynamic. During the analysis, many possible interpretations are considered and each preference rule expresses an opinion as to how well it is satisfied by a given interpretation. The opinions of each rule are then combined to form a preferred analysis.

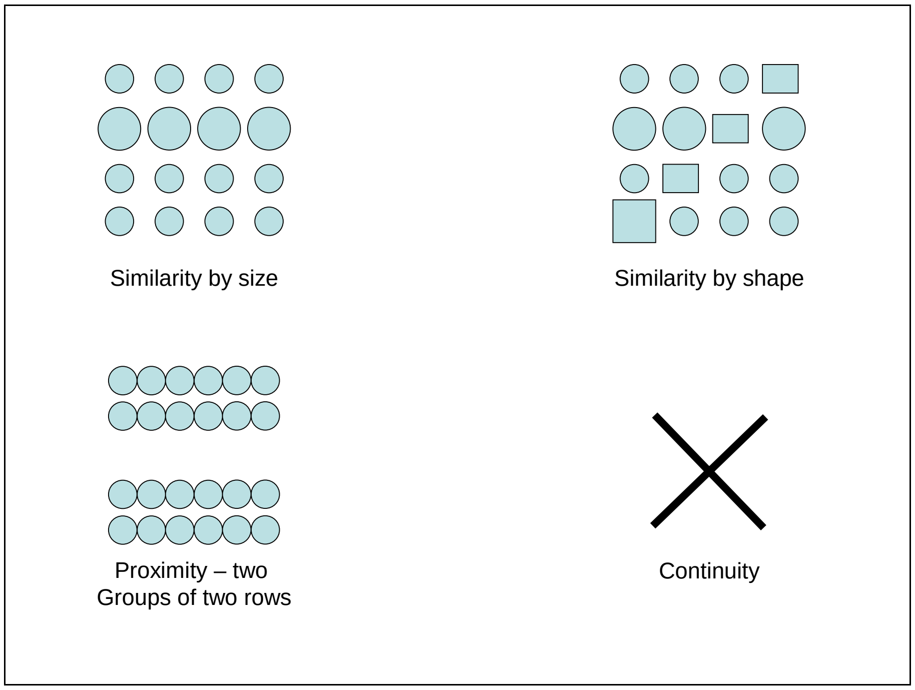


Figure 3 Gestalt rules of perception

Preference rules owe some of their ancestry to the Gestalt rules of perception first proposed by researchers in psychology in the 1920s. The Gestalt rules of particular relevance are the similarity, proximity and continuity rules illustrated in Figure 3. The similarity rule suggests that we tend to group things together that are similar. For example given a set of geometric shapes, we would form groups of similar shapes (squares, circles, triangles etc) or group similar shapes by their size. The proximity rule suggests we group things that are close together (either in space or some other dimension). Finally the continuity rule states that elements that follow each other in a linear pattern can be grouped together. For example, we perceive an X as two crossing lines rather than as a V over an inverted V. These rules are particularly useful when segmenting a musical performance into phrases and other structural sections.

In their book, Lerdahl and Jackendoff proposed a set of structures and well-formedness rules that identified legal structures. However, whilst proposing the structures and rules, they offered no implementations that could be tested. Temperley [Temperley01] however, has taken this work further by quantifying and implementing a set of rules and expanding their applicability from Western Classical music to include Rock and World music. Temperley proposes a set of six preference rule systems that match and extend the fundamental psychological properties identified by Dowling. These rule systems are:

1. Metrical Structure
2. Phrase Structure
3. Contrapuntal Structure
4. Tonal-Pitch-Class Structure
5. Harmonic Structure
6. Key Structure

Implementations of each of the rule based analysis components are available from his Website. Temperley's implementation of the rule systems makes use of dynamic programming techniques. In dynamic programming, rather than evaluating all possible interpretations then backtracking to evaluate the most optimal, the "best-so-far" solution is evaluated and retained at each step in a forward direction. What constitutes a "best-so-far" solution is established by the set of preference rules. Once the input has been completely traversed, the final analysis or solution is complete.

The prime motivation for Temperley's work [Temperley01] and that of others [Huron97] is musicological analysis. For this, they have access to the complete piece of music in some encoded form. This approach is not applicable for real-time analysis of musical events since in this case at a given point in time, only historical data is available.

Chew [Chew00] identifies computational music analysis as an interdisciplinary study that links human perception and cognition, mathematical modelling and computation with music theory. She identifies the need to determine the tonal centres and their progression as of critical importance in the analysis and perception of music since this forms the basis of an understanding of harmonic structure. Chew's model of tonal perception is grounded in the work of Temperley and Krumhansl [Krumhansl78] which themselves are grounded in cognitive psychology.

2.3.4 Discussion

For a real-time analytical framework, analysis concentrates on classification in Brinner's declarative domain. Temperley's work proposes a set of six rule systems that provide techniques for extracting Metrical, Phrase, Contrapuntal, Tonal-Pitch-Class, Harmonic and Key Structures from music. His rule sets provide a realisation of the rules proposed, but not implemented, by Lerdahl and Jackendoff in their *Generative Theory of Tonal Music*.

Chew has developed this further by building on the work of Temperley and Krumhansl to provide a mathematical model of musical tonality that can form the basis of note identification and subsequent harmonic analysis. Her work and its development forms an important part of the real-time framework proposed by this thesis and will be discussed in more detail in later chapters.

Chapter 3 Publish/Subscribe Messaging

3.1 Introduction

This chapter describes the Publish/Subscribe messaging model (Pub/Sub) which underpins the framework described in this thesis. It first describes the messaging pattern which allows communication between applications to be decoupled in time, space and synchronisation. This is then followed by a comparison with other messaging patterns and descriptions of the basic publish and subscribe applications developed; file publishers, MIDI input and output applications.

3.2 The Publish/Subscribe Messaging Pattern

The pub/sub messaging pattern is one that is widely used in business event messaging since it provides a loosely coupled form of interaction that can be scaled to large numbers of applications distributed throughout a network. In this pattern, subscribing applications express their interest in a set of events using topics within a shared namespace and are notified of any event, generated by a publisher, which matches their registered interest. Events are asynchronously propagated to all subscribers that registered interest in the type(s) of event. The advantage of this type of interaction style lies in the decoupling in space, time and synchronisation between the publishing and subscribing applications.

The pub/sub pattern is illustrated in Figure 4. The pattern relies on an event notification service (otherwise known as a publish/subscribe broker) which provides the storage and management of subscriptions and the efficient delivery of events to subscribers.

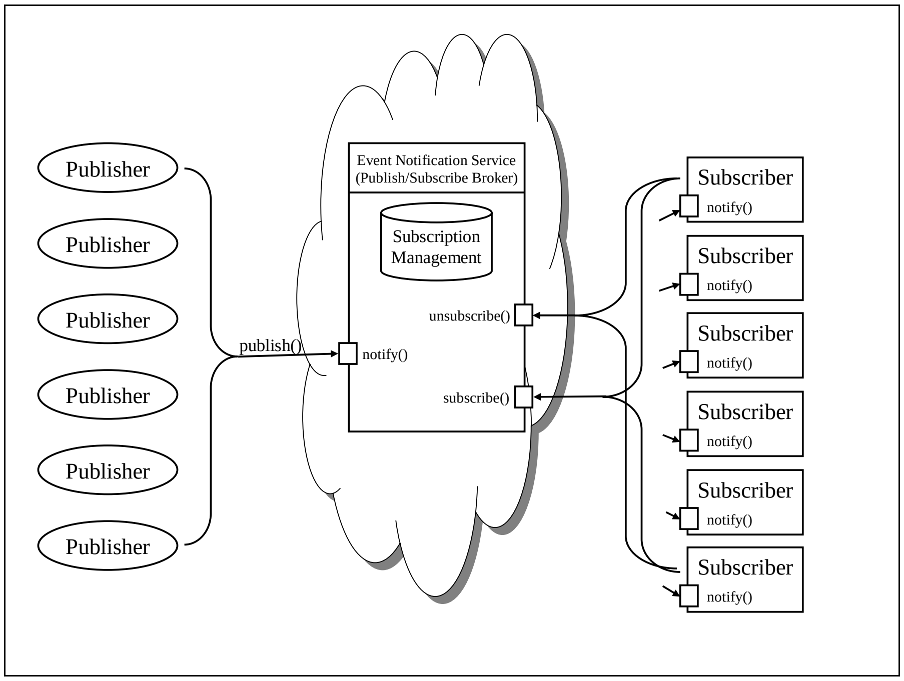


Figure 4: Components of the Publish/Subscribe Messaging Model

In the simple (or topic based) model, the broker acts as a neutral intermediary between publishers, acting as producers of events, and subscribers acting as consumers of events. Subscribers register their interest in particular sets of events by invoking a *subscribe()* operation on the event service providing a set of *topics* within a global namespace that identifies the events they are interested in. This subscription information is stored by the event service and used to distribute events received by the broker from publishers. A subscription is terminated by a subscriber invoking the *unsubscribe()* operation on the event service.

To generate events, a publishing application calls a *publish()* operation for an event. The event consists of two parts; message content and an associated topic contained within the global namespace. The publication notifies the event service that an event is available which it then propagates to all relevant subscribers that have registered an interest in events with the associated topic. The event service is acting as a proxy for the set of subscribers. Every subscriber will be notified of every event conforming to its interest.

The introduction of an event service between publishers and subscribers provides decoupling between publishers and subscribers that can be described in three dimensions and is illustrated in the Figure 5.

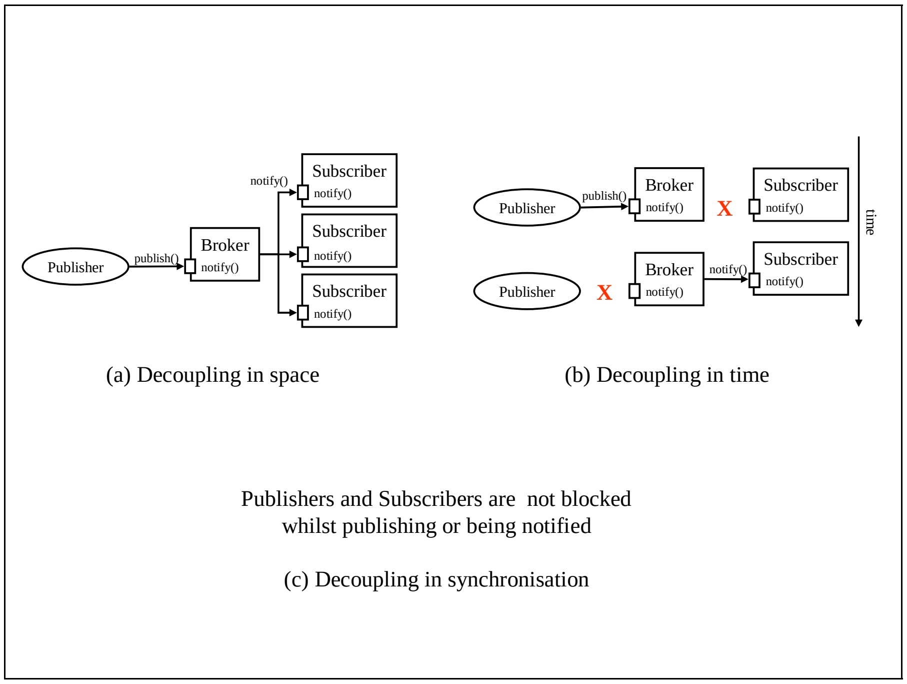


Figure 5: Decoupling introduced by the Publish/Subscribe messaging model

Decoupling in space. Publishers and subscribers do not need to know that each other exists. The publishers publish events to the event service and the subscribers receive the events indirectly through the event service. Publishers do not hold references to the subscribers and do not know how many subscribers are participating in the interaction. Conversely, subscribers do not hold references to the publishers, nor do they know how many of them are participating in the interaction.

Decoupling in time. Publishers and subscribers do not need to be actively participating in an interaction at the same time. In particular, a publisher may publish events whilst subscribers are disconnected from the network. Conversely, a subscriber may be notified of an event whilst the publisher of that event is disconnected. The event is persisted by the event service and only forwarded to subscribers once they are connected to the broker. The sophistication of this *store-and-forward* functionality will vary depending on the implementation of the event service.

Decoupling in synchronisation. Publishers are not blocked whilst producing events and subscribers can be asynchronously notified (through a callback mechanism) of the availability of an event whilst performing other concurrent activities. The production and consumption of events do not happen within the main control flow of publishing and subscribing applications which results in a non-blocking, asynchronous communication model between publishers and subscribers. This decoupling increases the scalability of the messaging model by removing all dependencies between the participating publishers and subscribers and reduces the need for coordination and synchronisation between them.

Decoupling of publishers and subscribers results in a communications infrastructure that is well adapted to distributed environments that are asynchronous in nature such as mobile environments [Huang01] and wireless based sensor networks [Bergamaschi07].

3.3 Other Messaging Patterns

This section reviews other communications patterns; in particular, message passing, remote procedure calls (RPC), notifications, shared spaces and message queuing. From this review, it can be seen that they do not provide fully decoupled communication between participating applications of the kind provided by the publish/subscribe messaging model.

3.3.1 Message Passing

Message passing is a primitive communication pattern in which two participants (a sender and a receiver) communicate by sending messages to each other. This pattern is rarely used for developing distributed applications since physical addressing, data marshalling and flow control are made visible to the application layer. In this pattern, both sender and receiver are coupled in space and time since they must both be active for the communication to take place and the recipient of the message is known to the sender.

3.3.2 Remote Procedure Calls

Remote Procedure Calls (RPC) are a widely used pattern in distributed systems. They were first proposed for procedural languages [Birrell83] but have subsequently been widely used for remote method invocations for object-oriented systems. For example, they have formed the basis of the Java Remote Method Invocation [Sun00],CORBA [OMG02] and Microsoft’s DCOM [Horstmann97].

RPCs make remote interactions between applications appear in the same way as local interactions. Remote invocation of this kind cannot, however be made completely transparent to the application because it can give rise to potential failures such as in the communications layer that have to be handled explicitly by the application. Unlike the publish/subscribe model, RPCs introduce strong coupling in space, time and synchronisation between the participating applications; the invoking object has to maintain a reference the remote object and the interaction is synchronous with the invoking object being blocked for the duration of the interaction. Attempts have been made (particularly with CORBA) to introduce asynchronous remote invocation but this has led to invocations with weak reliability guarantees because the sender does not receive any success or failure response to the invocation. This is often referred to as a *fire-and-forget* interaction.

3.3.3 Notifications

To achieve decoupling in synchronisation, a synchronous remote method invocation can be split into two asynchronous invocations. In this pattern, the sender invokes the receiver with the invocation arguments and a reference to a callback to itself. The receiver then invokes the original sender via its referenced callback. This pattern can be extended to a *one-to-many* pattern by the sender invoking many receivers each of which responds via a separate callback. Notification based interactions of this kind are often used to ensure the consistency of web caches where proxies are notified of any changes that occur in content on a web server. This implements a limited type of publish/subscribe mode, where the web proxies act as subscribers and the web server acts as a publisher. However, in this case, the publisher directly manages a set of subscriptions and is also responsible for sending events which removes the decoupling in space and time. In addition, the publisher is responsible for managing communications which limits the scalability for this system pattern.

3.3.4 Shared Spaces

Shared spaces is a development of the *distributed shared memory* model proposed in [Li89]. In this model, distributed applications have a common view of a shared memory space with synchronisation and communication between participating applications taking place through operations on the shared data. Linda [Gelernter85] introduced the concept of *tuple spaces* which provided an abstraction for accessing shared memory. A *tuple space* is a collection of ordered tuples accessible to applications within a distributed system. Communication between applications takes place through the insertion and removal of tuples from the shared space.

The shared space pattern provides time and space decoupling in that the producers and consumers of tuples are anonymous with respect to each other. However, the model is not decoupled with respect to synchronisation since consumers remove tuples from the shared space in a synchronous manner. To compensate for this deficiency, some applications of the shared spaces model such as JavaSpaces [Sun02] and TSpaces [Lehman99] extend the tuple space model with asynchronous notifications.

3.3.5 Message Queuing

Message queuing is often used to describe a family of commercial products including those from IBM [IBM95], Digital [DEC94] and Oracle [Oracle02]. Message queuing systems usually include some form of pub/sub type interaction and are generically referred to Message Oriented Middleware (MOM). In message queuing, messages are stored in a First In First Out (FIFO) queue. Producers append messages asynchronously to the queue while consumers de-queue messages from the front of the queue. Message queues have many of the interaction properties of tuple spaces in that queues can be viewed as global spaces which are populated by messages from producers. Functionally, commercial queuing systems provide transactional, timing, ordering and reliable delivery guarantees which are not normally seen as properties of tuple spaces.

As with tuple spaces, producers and consumers are decoupled in both time and space in message queuing systems. However, message queues do not provide synchronisation decoupling since consumers remove messages from the queue synchronously. This lack of synchronisation decoupling is further reinforced when queuing is being performed as part of a distributed transaction. Some commercial systems do provide support for asynchronous message delivery but these mechanisms do not scale well to large distributed systems because of the additional interactions required to maintain transactional, timing and ordering guarantees.

3.3.6 Summary

In summary, the more traditional types of message passing do not have the same degree of flexibility and scalability as the publish/subscribe messaging model. This is due to their limited support for decoupling of producers and consumers in time, space and synchronisation.

3.4 Variants of the Publish/Subscribe Model

Subscribing applications are usually selective and only interested in individual or limited sets of events. Variants of the publish/subscribe model have emerged that provide alternative ways of specifying the events of interest when applications subscribe to the event service. The two major variants are *topic* and *content* based subscriptions.

3.4.1 Topic Based Publish/Subscribe

The earliest forms of publish/subscribe messaging were based on the notion of topics or subjects identified as a set of keywords associated with the event. Publishers and subscribers can publish events and subscribe to individual topics which are identified by the keywords. This variant of the publish/subscribe model has been widely exploited in commercial products including Java [Altherr99], Tibco [TIBCO99] and more recently, the Java Messaging Service [JMS02].

Practically, topic based publish/subscribe systems provide a programming model which maps individual topics into separate communications channels. The interfaces are similar to those for the event service with the topic being provided as an argument to the publish() method (in the case of publishers) or subscribe() (in the case of subscribers). The topic abstraction usually allows platform independence by relying on strings to identify the topics. Although a simple type, strings are used to group topics into hierarchies as a further improvement to the variant. For example, a MIDI short command Note On event may be identified using the topic *Midi/Short/NoteOn* to differentiate it from a MIDI meta event such as a track name using the topic *Midi/Meta/TrackName*. Wildcards may be used when subscribing to topics so that for example, an application that only processes MIDI short commands can subscribe to the topic *Midi/Short/*+ where + is a wildcard character. In this case, all sub-topics in the *Midi/Short* hierarchy will be subscribed to.

The publish/subscribe broker used in the framework described in this thesis is an example of this variant and more examples will be provided in later sections and chapters.

3.4.2 Content Based Publish/Subscribe

In a publish/subscribe system, events are expressed as messages which have two parts; content and an associated topic. The previous section has described a variant which uses topics alone as the basis for identifying events. To overcome this limited expressiveness, the *content based* [Rosenblum97] variant introduces a subscription scheme based on the actual content of the event messages; the properties of the events themselves rather than an externally assigned topic. These properties may be internal elements of data structures as in Gryphon [Banavar99] or meta-data associated with events as in the Java Messaging Service [Hapner02].

In this variant, subscribers specify filters using a subscription language which define constraints that identify valid events. Constraints can be logically combined to form more complex filter patterns. These patterns are used to identify sets of events of interest for a given subscriber and propagate them accordingly. The patterns may be specified as strings using some subscription grammar such as SQL or a proprietary language, a template matching object or as an executable code module.

Content-based publish/subscribe is widely used in commercial systems where the set of possible events are tightly designed and managed. The structure of an event message and its contents are well known and available to application developers in some standardised form such as XML Schemas through library systems. The complexity of the event service supporting a content-based system is greater since it now has the overhead of processing the content of each event message to apply the subscription filter. This overhead is deemed acceptable when balanced against the increased flexibility content-based systems provide for commercial infrastructures.

3.5 A Real-Time Analysis Framework

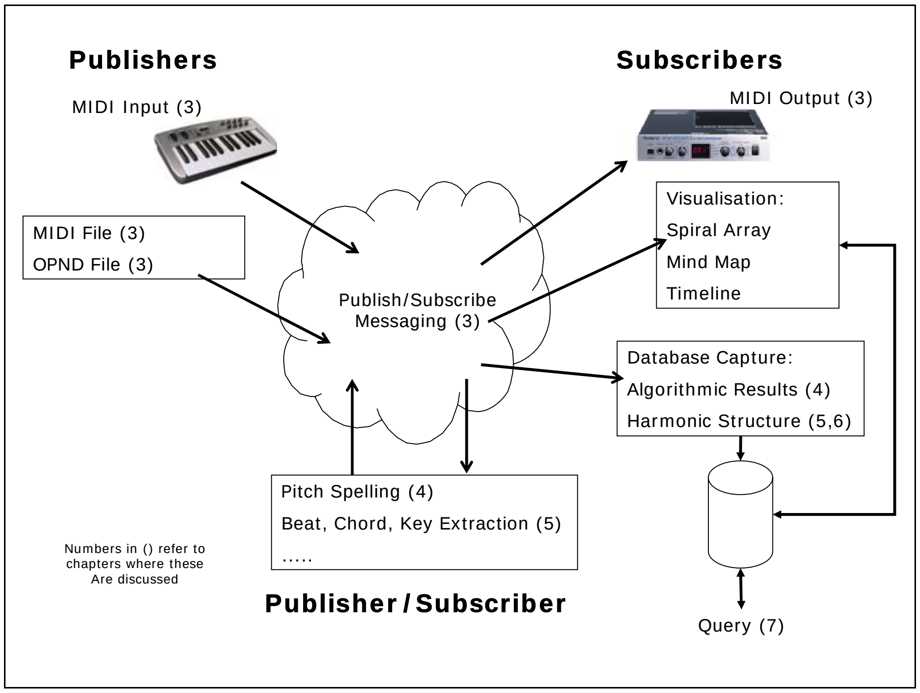


Figure 6: A real-time analysis framework

Using a publish/subscribe messaging infrastructure together with distributed applications, the kind of framework illustrated in Figure 6 becomes possible. MIDI input devices such as keyboards may publish events as a performance takes place. In addition, specialised file publishers can “play” MIDI or other formatted events into the framework either in real-time (the events are published at their correct timestamp) or with accelerated time for testing purposes.

Analysis applications are typically both subscribers and publishers. Applications for pitch spelling and beat, chord and key extraction will be demonstrated, but alternative algorithms may be implemented that contribute to the framework.

Subscribers to the framework may include MIDI output devices, database capture and visualisation applications. MIDI output will be via a subscribing application that transorms the event into commands output via a conventional MIDI interface. Visualisation applications may, for example, include specific widgets for representation of the spiral array pitch spelling algorithm, modified mind maps that show published events within a subset of the global topic space and timeline widgets to illustrate the relationship between objects within a Harmony ontology. The third class of applications include the persisting of events within databases; these may be either conventional relational databases or semantic databases or triple stores. With persisted data, opportunities exist for further navigation and enrichment of the captured events together with augmented visualisation tools to allow users to interact more fully with musical events captured from the framework.

These types of applications are described in the following chapters.

3.6 The MQTT Broker

The publish/subscribe broker used in the framework described in this thesis is the IBM Lotus Expeditor microbroker which supports the MQ Telemetry Transport (MQTT) publish/subscribe protocol. Developed by colleagues [O'Connell07] at IBM and used within UK e-Science projects [Robinson06], this broker is a small footprint (less than 2MB of Java code) and is intended to be deployed on small devices such as PDAs which may be communicating over low-bandwidth networks such as satellite communications links.

In addition to providing implementations of the abstract methods described in previous sections, the microbroker also supports different qualities of delivery service between itself and connected applications. These are:

At most once **delivery (QoS 0)**. Messages are delivered according to the best efforts of the underlying TCP/IP network. No response is expected. No retry semantics are defined in the protocol. Consequently, the message will arrive at the destination broker either not at all or once. This is also known as *fire and forget*.

At least once **delivery (QoS 1).** The arrival of a QoS 1 message at the broker, including its successful placement in a persistent store is acknowledged. In the event of identifiable failure of the communications link, or of the sending device, or after some period of time of non-receipt of the acknowledgement message, the sender will resend a duplicate message. Consequently, the message is certain to arrive, but could arrive more than once.

***Exactly once* delivery (QoS 2).** For QoS 2, additional protocol flows are employed above QoS 1 to ensure that duplicate messages are not delivered to the receiving application. This is the highest level of service, and is used when duplicate messages are undesirable. Of course, there is a price to be paid in terms of network traffic, but often this is acceptable because of the importance of the message content.

The next sub-sections describe aspects of the microbroker use of relevance to the framework described in this thesis.

3.6.1 The Framework Topic Space

All topics used within the framework are defined as Java Strings within the *EventConstants* class. For example, the definitions for MIDI short events are shown in the following code segment using the wildcard character as shorthand when subscribing to all MIDI short events:

```
// Midi Short Events
String shortTopic       = "Midi/Short/+";
String noteOnTopic      = "Midi/Short/NoteOn";
String noteOffTopic     = "Midi/Short/NoteOff";
String chanPressTopic   = "Midi/Short/ChannelPressure";
String cntlChangeTopic  = "Midi/Short/ControlChange";
String pitchBendTopic   = "Midi/Short/PitchBendChange";
String pgmChangeTopic   = "Midi/Short/ProgramChange";
String afterTouchTopic  = "Midi/Short/AfterTouch";
String sysMessageTopic  = "Midi/Short/SystemMessage";
```

The following table lists all MIDI topics within the framework together with the parameters used to construct the event message content to be published. A full set of topics and event contents can be found in Appendix A.

Topic	Event Contents	Description
Midi/Short/NoteOn	Timestamp, Channel, Note, Velocity	Note depressed
Midi/Short/NoteOff	Timestamp, Channel, Note, Velocity	Note released
Midi/Short/PolyKeyPressure	Timestamp, Channel, Note, Pressure	Note aftertouch – pressing a note after “bottoming out”
Midi/Short/ControlChange	Timestamp, Channel, Controller, Value	Change in a controller value
Midi/Short/ProgramChange	Timestamp, Channel, Program (Patch)	Change to a program patch number
Midi/Short/KeyPressure	Timestamp, Channel, Pressure	Note aftertouch – pressing a note after “bottoming out”
Midi/Short/PitchWheelChange	Timestamp, Channel, Value	Change in pitch wheel setting
Midi/Short/SystemMessage	Timestamp, Channel, Message Text	System message
Midi/Meta/SeqNumber	Timestamp, Sequence Number	Change in a Midi sequence number
Midi/Meta/Text	Timestamp, Text String	Arbitrary text event

Midi/Meta/Copyright	Timestamp, Copyright Text String	Copyright text
Midi/Meta/TrackName	Timestamp, Track Name String	Track name
Midi/Meta/InstrumentName	Timestamp, Instrument Name String	Instrument name
Midi/Meta/Lyric	Timestamp, Lyric String	Lyric
Midi/Meta/Marker	Timestamp, Marker String	Marker
Midi/Meta/CuePoint	Timestamp, CuePoint String	Cue point
Midi/Meta/ChannelPrefix	Timestamp, Value	Channel prefix
Midi/Meta/EndOfTrack	Timestamp, Empty	The end of a Midi track
Midi/Meta/Tempo	Timestamp, Value	Time in microseconds per beat
Midi/Meta/SMPTE	Timestamp, Value[0..4]	SMPTE time information
Midi/Meta/TimeSig	Timestamp, Value[0..3]	Time signature and other timing information
Midi/Meta/KeySig	Timestamp, Value	Key signature including Major/Minor
Midi/Meta/Vendor	Timestamp, String	Vendor specific information
Midi/Meta/UnknownMeta	Timestamp, String	Unknown Meta Event
Midi/Sysex/SysExcl	Timestamp, Value	Extension and manufacturer specific info
Midi/Sysex/SysSpExcl	Timestamp, Value	Extension and manufacturer specific info

Table 1: MIDI Topic Space and Event contents

3.6.2 Connecting to the Broker

Publishing and subscribing applications connect to the MQTT broker using the *BrokerConnection* class which implements an *MqttCallback* interface. The constructor to this class includes the TCP/IP address (String) and port number (int) of the broker, the name of the connecting publishing or subscribing client application (String), if a subscriber, then the set of topics to subscribe to (String []) and the name of a Java queue to that provides an asynchronous connection between the client and its associated processing component. Typically, this queue is a Java *BlockingQueue* typed to enqueue *PubSubEvents*. The interactions between the components of a generic client application are illustrated in Figure 7.

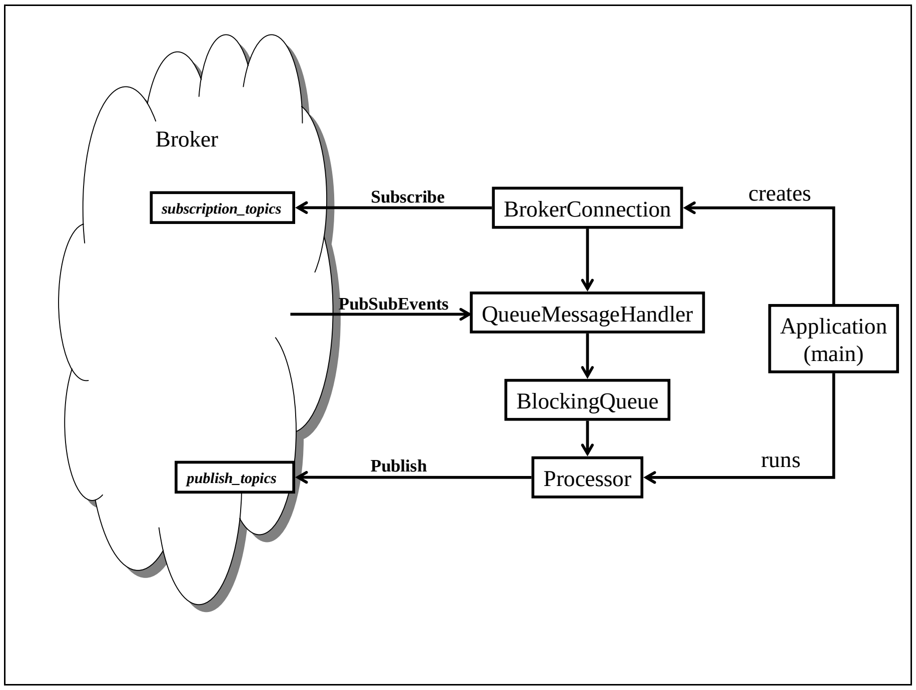


Figure 7: Generic application structure

When a client application instantiates a *BrokerConnection*, it connects to the broker which may be on a local or remote machine and subscribes to a set of topics. The broker then forwards all events that match the subscribed topic to the client. All events generated within the framework have a super-class of *PubSubEvent* from which all events are sub-classed. The *BrokerConnection* callback invokes the *QueueMessageHandler* class which casts the *PubSubEvent* into the appropriate sub-class based on the received topic. For example, a received *PubSubEvent* with a topic of *EventConstants.noteOnTopic* will be cast into an object of type *MidiNoteOnEvent*. This object will be placed on the *BlockingQueue*.

The purpose of the *BlockingQueue* is to synchronise the receipt of events by the client from the broker with the processing of the event by the client application. The application runs the processing class (such as note naming or metrical analysis) as a separate Java thread with the queue as an interface to the main client thread. The processing thread performs a blocking *get()* of objects from the queue which it can then process. A blocking *get()* means that if the queue is empty, the application waits until a new object is placed on the queue input. If the processor publishes events (such as the results of some analysis), it can use the *BrokerConnection.publish ()* method to publish an event without impacting the main client thread.

The BrokerConnection class also includes methods for unsubscribing, disconnecting from the broker and handling lost connections to the broker.

3.6.3 Framework Events

All events within the framework are subclassed from the *PubSubEvent* class. The class constructor takes two arguments; the topic used to publish the event and a URI that uniquely identifies the event instance. URIs are generated by a *URIGenerator* class and their use is described in later chapters. The topic and URI are saved as class variables which also include the Java *byte[]* array containing serialised event variables. This byte array contains the contents of the message to be published. The class also contains a *publish()* method which in turn invokes the *BrokerConnection.publish (String, byte[])* method to publish the event.

Events subclassed from the PubSubEvent class contain two constructors; one is intended for publishing applications contains the event parameters as arguments and the other for subscribing applications contains a single argument consisting of the serialised byte array. For example, the *NoteStartEvent* which identifies when a note has started during harmonic analysis has four arguments; the note's URI, the time it started (long), its name (String) and the octave it sounds in (int). These arguments are stored as class variables which are serialised into a byte array for publication. The code to serialise this set of variables is illustrated in the following code segment.

```
byte[] bytes;
ByteArrayOutputStream bos = new ByteArrayOutputStream(1024);
DataOutputStream dos = new DataOutputStream(bos);
try {
     dos.writeUTF(noteURIStr);
     dos.writeUTF(timelineURIStr);
     dos.writeLong(startTime);
     dos.writeUTF(noteName);
     dos.writeInt(octave);
     dos.close();
     bytes = bos.toByteArray();
} catch (IOException e) {
     e.printStackTrace();
}
```

The constructor used in subscribing applications takes a single argument; the byte array. This is deserialised in the constructor to recover the event variables. Using the above example, the variable deserialising is illustrated below.

```
ByteArrayInputStream bis = new ByteArrayInputStream(bytes);
DataInputStream dis = new DataInputStream(bis);
try {
     noteURIStr = dis.readUTF();
     timelineURIStr = dis.readUTF();
     startTime = dis.readLong();
     noteName = dis.readUTF();
     octave = dis.readInt();
} catch (IOException e1) {
     e1.printStackTrace();
}
```

Therefore the general pattern for the lifecycle of an event is:

1. A publishing application constructs an event of the correct type providing a set of event variables as arguments
2. The event constructor serialises the arguments as a Java byte array and associates the correct topic for the event type
3. The constructor publishes the event using the event topic and the serialised array
4. Once the event has been received by a subscribing application as a PubSubEvent type, it is cast to the appropriate subclass based on its topic with a single constructor argument; the byte array
5. The byte array is deserialised into the set of event variables which may be retrieved by the subscribing application via a set of access methods.

3.7 Publishing Applications

This section describes two examples of publishing applications to illustrate how events may be published. The first section describes the publication of events from files – in particular from MIDI and OPND format files which form the testcase corpus for this thesis. The second section describes a MIDI input application which allows controllers such as a keyboard to be connected to the framework to allow real time events to be generated and published.

3.7.1 File Publishers

The Java programming language provides comprehensive support for the MIDI specification. Included in the specification are the *Transmitter* and *Receiver* classes which allow MIDI devices to be programmatically interconnected in a similar way to MIDI hardware using patch cables. Java MIDI *Transmitters* and *Receivers* are specified as interfaces; therefore code can be developed to mimic the operation of a receiver or transmitter. For event publication, a *Receiver* can be written which publishes the MIDI event. This is the role of the *MidiEventPublisher* class within the framework.

To support the Java *Receiver* interface, the class must implement the interface *send()* method which is invoked by a *Transmitter* to propagate a MIDI message from one device to another. The implementation of this method is illustrated below.

```java
public void send(MidiMessage message, long lTimeStamp) {
      if (message instanceof ShortMessage) {
            publishShortMessage(message);
      } else if (message instanceof SysexMessage) {
            publishSysExMessage(message);
      } else if (message instanceof MetaMessage) {
            publishMetaMessage(message);
      }
}
```

The Java MIDI specification subclasses the *MidiMessage* into one of three types with each message type having a different structure. The *MidiEventPublisher* class has separate methods to publish each type. For example, consider the publication of MIDI Short messages which are typically used to control hardware devices. This is implemented by the *publishShortMessage()* method which takes a single argument, the *MidiMessage*. The method identifies the type of message by extracting the MIDI command from the message structure, creates an appropriate event from the *ShortMessage* and publishes it with the correct topic. For *NoteOn* and *NoteOff* events that match the same note, the same URI is used to signify that these two events are connected. This is achieved by managing an array of 128 *MidiNoteOnEvents*. When a *MidiNoteOnEvent* occurs, its URI is inserted into the array at the index corresponding to its MIDI note number before publishing. When a *MidiNoteOffEvent* occurs, its MIDI note number is used to extract its *MidiNoteOnEvent* from the array and the URI is used to publish the *MidiNoteOffEvent*. The entry in the array is then cleared in preparation for the next event.

Given this implementation of the *Receiver* interface, applications such as a MIDI file player can be implemented using the *MidiEventPublisher* to publish the MIDI messages to subscribing applications.

The *MidiEventPublisher* class includes a second implementation of a *send()* method. This implementation takes a single argument of type *NoteEvent*. The *NoteEvent* class is used to publish note events that have been derived from other sources. In particular from OPND structure files. OPND files are text files containing note information in the format onset, pitch name and duration. Therefore a record in the file such as 0, Bf2, 325 denotes a B flat in the second octave starting at time 0 and lasting for 325 milliseconds. Given this information, a matching pair of *MidiNoteOnEvent* and *MidiNoteOffEvents* can be created by computing the MIDI number of the note named Bf2. Therefore the implementation of *send()* with a *NoteEvent* argument publishes two events corresponding to the MIDI note on and off events.

The OPND testcases used for this thesis have been derived from the Musedata[Hewlett97] corpus and include the name of the note together with its octave. The *MidiNoteOnEvent* and *MidiNoteOffEvents* can optionally include the name of the note they represent which is not available in a pure MIDI environment. For algorithm testing purposes, the name can be published as part of the note event and is used to verify the correctness of the note naming algorithm described later in this thesis.

3.7.2 MIDI Input

Given a *MidiEventPublisher* class of the type described in the previous section, an application which acts as a MIDI input can be easily created. An example is shown in the following code fragment.

```
// Broker address, port and client name
private static String brokerAddress = "127.0.0.1";
private static int brokerPortNumber = 1883;

// Topics to subscribe to
private static final String[] topics = {EventConstants.shortTopic};

// Queue to receive published events
private static BlockingQueue<PubSubEvent>
                subQueue = new  ArrayBlockingQueue<PubSubEvent>(1000);
```

```
public static void main(String[] args) {
     // Connect to the Broker
     try {
           brokerConnection = new BrokerConnection(brokerAddress,
                                    brokerPortNumber,
                                    "MidiIn", null, subQueue);
     } catch (MqttException e) {
            e.printStackTrace();
     }

     // Get a list of Midi output devices
     MidiDeviceManager midiDeviceManager = new MidiDeviceManager();
     ArrayList inputDeviceList = MidiDeviceManager.getInputDevices();
     for (int i = 0; i < inputDeviceList.size(); i++)
            // Add a read to allow MidiIn device to be selected
            int mSelect = 0;

            // Get the selected device .....
            MidiDevice inDevice = MidiDeviceManager.getMidiDevice((
                  (MidiDevice.Info)inputDeviceList.get(mSelect)));
            try {
                  // ..... open it .....
                  inDevice.open();

                  // ..... get its transmitter .....
                  Transmitter t = inDevice.getTransmitter();

                  // ..... and add the midiPublisher to it
                  MidiEventPublisher midiPublisher = new
                                    MidiEventPublisher(qTime);
                  t.setReceiver(midiPublisher);

                  // Need to set up a loop to allow events to be
                  // published
                  while (true) {}
            } catch (MidiUnavailableException e1) {
                  e1.printStackTrace();
            }
      }
}
```

The application first connects to the broker and then obtains a set of the currently installed MIDI devices which may include a MIDI Input Port. Following selection of the input device, it is opened and the device *Transmitter* is obtained. Following the creation of a *MidiEventPublisher*, it is attached to the *Transmitter* of the input port. The final while loop is a simple mechanism to allow the program to continue. Whilst the loop is executing, any events that appear on the MIDI input port are propagated via the *Transmitter/Receiver* connection and published by the *MidiEventPublisher*.

3.8 Subscribing Applications

Subscribing applications are more complex that publishing applications because they require an asynchronous queuing interface to buffer the reception of events from their processing. This is illustrated using a simple MIDI output application which subscribes to MIDI events and outputs them to a connected MIDI port.

3.8.1 MIDI Output

The MIDI output application is similar in structure to the previously described input application.

```
// Broker address, port and client name
private static String brokerAddress = "127.0.0.1";
private static int brokerPortNumber = 1883;

// Topics to subscribe to
private static final String[] topics = {EventConstants.shortTopic};

// Queue to receive published events
private static BlockingQueue<PubSubEvent>
      subQueue = new  ArrayBlockingQueue<PubSubEvent>(1000);

public static void main(String[] args) {
     // Connect to the Broker
     try {
            brokerConnection = new BrokerConnection(brokerAddress,
                                                   brokerPortNumber,
                                            "MidiOut", topics, subQueue);
     } catch (MqttException e) {
            e.printStackTrace();
     }

     // Get a list of Midi output devices
     MidiDeviceManager midiDeviceManager = new MidiDeviceManager();
     ArrayList outputDeviceList =
                  MidiDeviceManager.getOutputDevices();
     for (int i = 0; i < outputDeviceList.size(); i++)
           midiDeviceManager.printInfo((MidiDevice.Info)
                                         outputDeviceList.get(i));
            MidiDevice outDevice = MidiDeviceManager.getMidiDevice((
                         (MidiDevice.Info) outputDeviceList.get(0)));

            // Create the output thread .....
            MidiOutThread moThread = new MidiOutThread(outDevice,
                                                        subQueue);

            // ..... and run it
            new Thread(moThread).start();
      }
}
```

In this example, following the selection of an output port device (assuming it is the first in the retrieved list) the application creates a *MidiOutThread* class and runs it.

The code for the *MidiOutThread* implements the *Runnable* interface and is listed below.

```
public class MidiOutThread implements Runnable{
     private static MidiDevice midiDevice = null;
     private static Receiver outReceiver = null;
     private static boolean closing = false;
     private BlockingQueue<PubSubEvent>   subscriberQueue;

     public MidiOutThread(MidiDevice mD,BlockingQueue<PubSubEvent> q){
            // Save the device and queue
            midiDevice = mD;
            subscriberQueue = q;

            // Open the device and get its Receiver
            try {
                  midiDevice.open();
                  outReceiver = midiDevice.getReceiver();
            } catch (MidiUnavailableException e) {
                  e.printStackTrace();
            }
     }

     public void run() {
            PubSubEvent psE;
            int id, x, y, r;
            while (!closing) {
                  try {
                        psE = subscriberQueue.take();
                        if (psE instanceof MidiShortMessageEvent) {
                              MidiShortMessageEvent smE =
                                          (MidiShortMessageEvent)psE;
                              smE.sendToMidiReceiver(outReceiver);
                        }
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            }
     }
}
```

The class constructor saves the selected *MidiDevice* and the *BlockingQueue* arguments. It then opens the *MidiDevice* and gets its receiver. Once the thread has been started, the *run()* method is invoked. This method loops, halting on the *subscriberQueue.take()* method call until a *PubSubEvent* appears on the queue as a result of the BrokerConnection callback being invoked from the broker. Once this occurs, the *PubSubEvent* is cast to a *MidiShortMessageEvent* which then sends the message to the output receiver by invoking its *send()* method.

A combination of a MidiIn application publishing messages from a MIDI controller and a MidiOut application which subscribes to the published topics and sending the messages to a MIDI output port will allow MIDI input devices such as a keyboard and output devices such as a synthesiser to be connected to the framework. Note that the properties of the publish/subscribe messaging model mean that the MidiIn, MidiOut and Broker applications can each reside on separate machines thereby providing a mechanism for MIDI events to be distributed throughout a network.

3.9 Summary

This chapter has described the Publish/Subscribe messaging model and its variants and compared them with other messaging techniques. It has described how the messaging model has advantages in decoupling applications in space, time and synchronisation. It has described the structure of published messages in that they contain a topic used by subscribing applications to identify which messages they are interested in together with the contents of the message. Within the context of the framework described in this thesis, the components used to implement the publish/subscribe messaging have been described together with example publishing and subscribing applications to show how MIDI devices may be connected to the framework.

Chapter 4 Pitch Spelling

4.1 Introduction

A fundamental aspect of human interpretation and understanding of music is the concept of *tonality*. The term *tonality* was first introduced in the early 19th century and is currently used to describe the structured relationships that exist between musical pitches. These relationships exist between the pitch of a note and its associated key which defines its harmonic context. This chapter reviews the background literature on *tonality* leading to the introduction of Chew's Spiral Array model as an algorithm for pitch spelling. An implementation of this algorithm is then described using the publish/subscribe messaging model. From this implementation, a simplified version is described which has equivalent performance and accuracy when tested with a corpus of around 200,000 notes.

The MIDI specification identifies notes played on a keyboard or sounded by a synthesiser as an integer lying between 0 and 127. The first stage in producing a harmonic representation of the musical data represented by the MIDI stream is to assign a name to the note. Pitch spelling algorithms predict the note name (D♯4, B♭3, A♮2 etc) given only the MIDI note number, the start (or onset) time and possibly the duration of the note.

4.2 Tonality and Musical Analysis

Tonality refers to the underlying structures and principles of tonal music and is sometimes synonymous with musical key. It denotes the relationships between the subjective concept of musical pitches and more specifically a system of relationships between pitches having a tonic or central pitch as its most important element. The perceptual term *pitch* refers to a sound of some frequency and is usually expressed using a subjective quality such as high or low. A *note* is a symbolic representation of two properties; *pitch* and *duration*.

Pitched sounds usually consist of a complex waveform consisting of several components or harmonics. The frequency of each component is a multiple of the lowest (or fundamental) frequency. The complex waveforms are produced by some physical mechanism as in the case of acoustic instruments or as a synthesized waveform as in the case of electronic instruments. A spectral analysis of the complex waveform will provide information about the fundamental and harmonic components of the waveform but this is outside the scope of this thesis. For the purposes of this thesis, musical streams of data are represented by MIDI commands where *pitch* is represented as a number between 0 and 127. This represents (for example) which key has been depressed on a MIDI keyboard. For a review of MIDI and other musical codes and representations, see Appendix E,

Dowling has observed that music of all traditions has the properties of octave equivalence together with a fixed (depending upon musical tradition) set of tones within an octave. Since tones are perceived in a cyclic manner where the cycle repeats every octave, Shepherd [Shepherd82] visualizes this perceptually as a *pitch helix*. This represents pitch as two descriptors; *height* to show octave equivalence and *chroma* or *pitch class* which identifies the rotational position of a given pitch within the helix. This is illustrated in Figure 8.

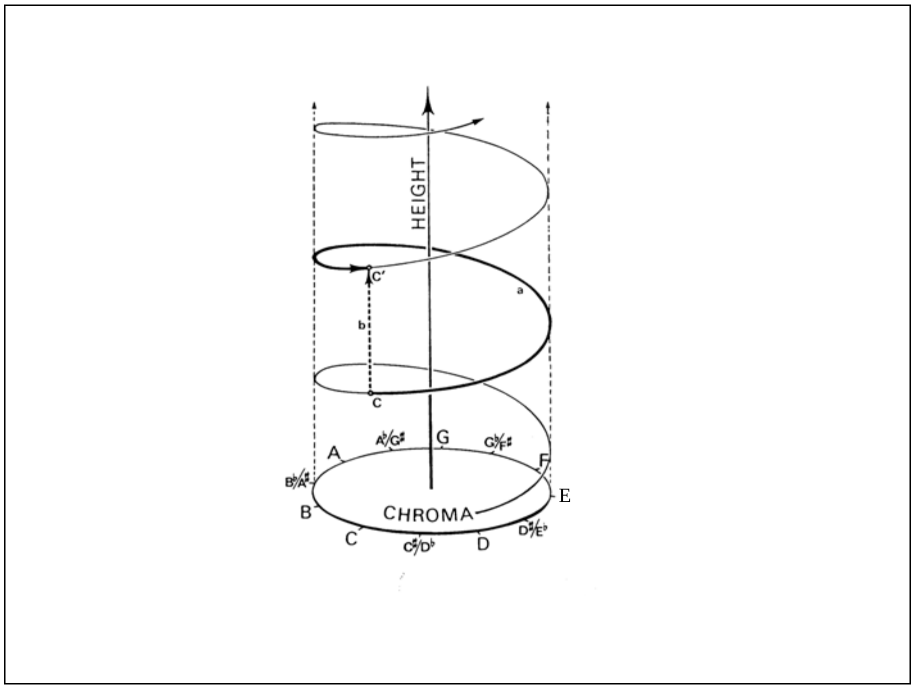


Figure 8: Shepherd's pitch helix

Given the definition of *pitch*, *melody* and *harmony* can now be defined. Britannica online defines melody as "..... *the aesthetic product of a given succession of pitches in musical time, implying rhythmically ordered movement from pitch to pitch*". This definition implies that melody is monophonic and consists of a sequence of pitches. *Harmony* by comparison is polyphonic. It denotes the simultaneous combination of notes into *chords* which over time change into *chord progressions*. In addition to describing *notes* and *chords*, *harmony* has its own corpus of knowledge embodied in musical theory. In the context of this thesis, harmony will only be considered in terms of harmonic content related to the combination of *notes* into *chords* and their relationship to *keys*. Although defining *melody* as monophonic and *harmony* as polyphonic, both refer to the combination of pitches into higher level musical structures and as such, they both influence each other.

Most of the research work on tonality has been carried out on western music [Krumhansl04]. Within this genre, a *key* is defined as a system of relationships between a series of pitches having a *tonic* or central pitch class as its most important element. Another important pitch class of a key is the *dominant* or fifth note of the scale starting at the tonic of the key. A key can have two *modes*; major and minor. Each of them has different orderings of the tones and semitones within their respective scales.

There exists a total of 24 keys when both major and minor modes are included; one of each mode for the 12 semitones within the chromatic octave and including enharmonic equivalence (where notes sound the same but have different names e.g. F♯ and G♭). These 24 keys can be arranged in a circle of fifths where a tonic on the circle is the fifth of the scale of the preceding tonic. This is shown in the following figure.

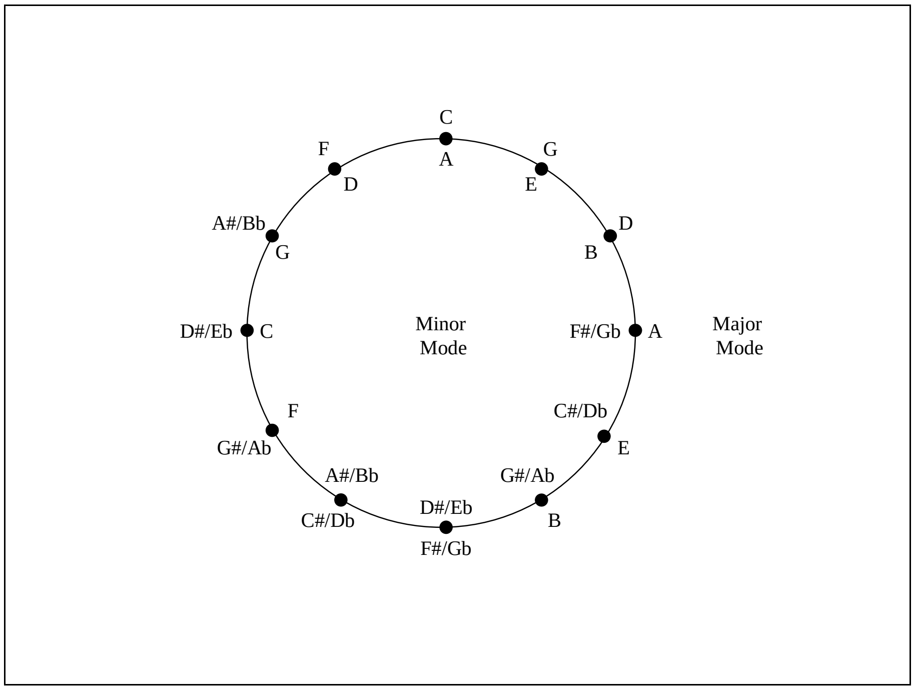


Figure 9: Circle of fifths

The circle of fifths also illustrates relationships between major and minor keys. A key has a signature which indicates the set of sharps or flats contained within its scale. A major key has a *relative* minor key indicated by the pairing within the circle of fifths. A relative minor key has the same key signature as its major key; for example A minor is the relative key to C major. *Parallel* keys share the same tonic but have different key signatures. For example C major has a different key signature to C minor.

Given this foundation, there has been a large amount of research effort in modeling human understanding of tonality. Early work included that by Winograd [Winograd68] who applied linguistic analysis techniques to automatic musical analysis. Later work has been influenced by the 19th century musicologist Riemann who observed that tonality derives from establishing significant relationships through chord functions. His theory asserted that the most significant intervals are the perfect fifth and major and minor thirds present in the triads based on the tonic of a key. Riemann mapped these relationships onto a harmonic network or table of tonal relationships known as a *Tonnetz*. Versions of this harmonic network have been traced to earlier theories of Euler [Cohn97].

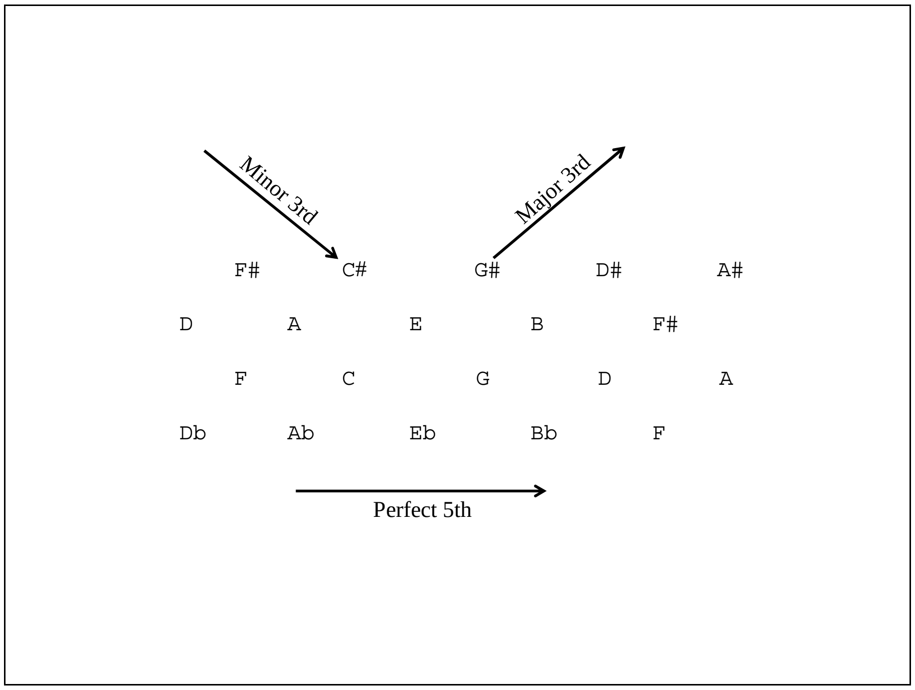


Figure 10: The Harmonic Network or Tonnetz

In this network representation, there is a horizontal relationship between entries of a perfect 5th (e.g. G is a perfect 5th above C) with diagonal relationships representing major and minor 3rds. It has been observed [Longuet-Higgins71] that pitches in a given key tend to cluster in a particular area of this network. They propose a key finder based on a shape matching algorithm. This was investigated further by Temperley [Temperley01] who proposed a simple key profile model which identifies the likelihood of the input set of pitches (comprising the musical score) matching a set of key profile vectors.

Based on the work of Longuet-Higgins, Chew [Chew00] has proposed a three dimensional representation of pitches formed from the harmonic network; this is referred to as the Spiral Array Model.

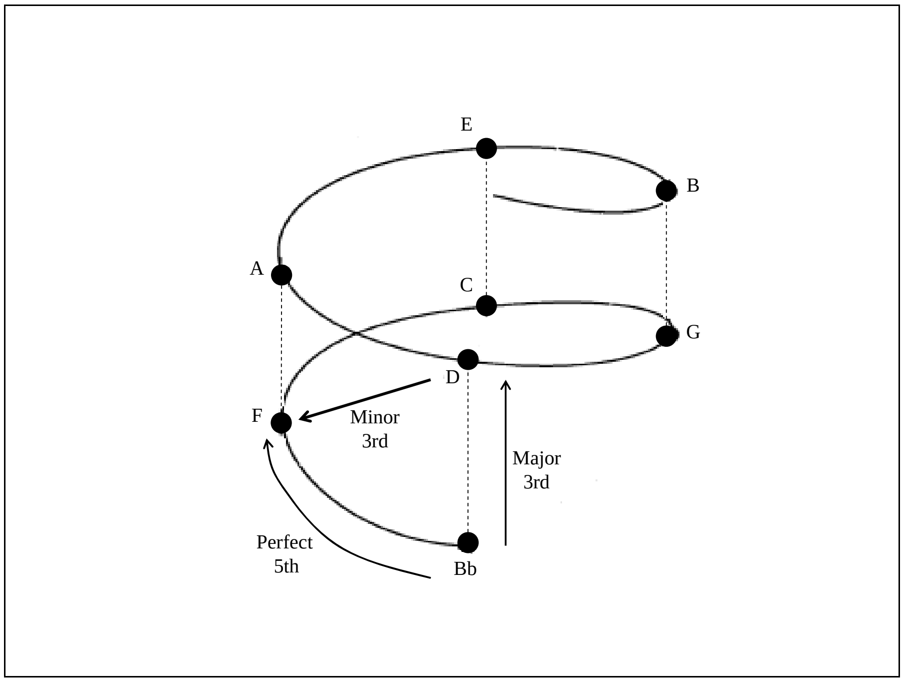


Figure 11: The Spiral Array

In this model, pitches are arranged along a spiral. Adjacent pitches are positioned at each quarter turn and are a perfect 5^{th} apart. The minor and major 3^{rd} intervals are shown as vertical and diagonal relationships in the diagram. Note that the ordering of pitches in the spiral array also reflects the ordering in the circle of fifths. Since the circle of fifths is closed, then the spiral array is closed and maps onto the surface of a toroid. However for computational purposes, the spiral array reflects linear distances which are not present in the toroidal case.

The advantage of the spiral array is that it introduces a spatial component which is not present in the two dimensional network array. In a later chapter, the use of the spiral array to identify chords and keys will be discussed. The next section describes Chew's note naming algorithm based on the spiral array together with a simplified version developed by the thesis author.

4.3 The Spiral Array Model

The note naming algorithm described in [Chew04] is based on the spiral array model described in [Chew00]. The spiral array is described mathematically as

$$P(k) = \begin{bmatrix} x_k \\ y_k \\ z_k \end{bmatrix} = \begin{bmatrix} r\sin(k\pi/2) \\ r\cos(k\pi/2) \\ kh \end{bmatrix} \tag{4.1}$$

Where r is the radius of the spiral and h is the vertical ascent per quarter turn and the aspect ratio $r/h = \sqrt{15/2}$. In the model, any collection of notes generates a *centre of effect* (CE). This is a point within the spiral array that is a convex combination of the pitch positions weighted by their durations. If S is a set of notes in a piece of tonal music, then the *centre of effect* CE(S) is defined as

$$CE(S) = \frac{\sum_{n \in S} d(n).p(n)}{\sum_{n \in S} d(n)} \tag{4.2}$$

Where $p(n)$ denotes the vector representing the position in the spiral array of note n which has duration $d(n)$. That is, the CE is the weighted centroid of the position vectors of the notes in the spiral array with each note being weighted by its duration. In the algorithm, the CE acts as a proxy for the key and that each note should be spelt so that it is as close as possible in the spiral array to the notes that preceded it. This approach is similar to that adopted by [Temperley01] except that Temperley uses a line of 5ths rather than a spiral array representation.

In the spiral array algorithm, it is assumed that the input data gives the MIDI note number together with the start time (onset) and duration of each note in milliseconds. The data is divided into equal time slices called chunks. The algorithm then names the notes one chunk at a time.

The algorithm adopts a bootstrapping and sliding window strategy whereby a set of preceding chunks are used to compute the current CE which is then used to name the notes in the current chunk. Naming consists of computing the nearest distance to the CE of the potential names for a note. For example a note identified by its MIDI note number may correspond to either B♯ C or D♭♭. Each of these three possible names occupies a point in 3-D space on the spiral array. Given the current CE, the distance between the CE and each of the three possible names is computed and the name with the nearest distance is assigned to the note.

According to [Chew04], there are two phases in the algorithm. Phase I consists of the following steps 1 and 2, Phase II consists of steps 3 to 5. The following variables are used in the description

$W(i, j)$	The set of notes that sound, start or are present in chunks *i* to *j*
$CE_{global,j}$	A global CE computed from the set of notes in a sliding global context window consisting of the w_s chunks preceding the *j*th chunk
$CE_{local,j}$	A local CE computed from the set of notes in a local context window consisting of the chunk *j* together with the $(w_r - 1)$ preceding chunks
$CE_{cum,j}$	A cumulative CE computed from all notes preceding chunk *j*
$CE_{hybrid,j}$	A hybrid CE which is a weighted version of the local and cumulative CEs

The following steps are executed:

1. The global CE is computed from the value

$$CE_{global,j} = CE(W(j - w_s, j - 1)) \tag{4.3}$$

2. The algorithm then names all notes in chunk j to be as close to $CE_{global,j}$ as possible in the spiral array

3. The local CE is computed from the value

$$CE_{local,j} = CE(W(j - w_r + 1, j)) \tag{4.4}$$

4. The algorithm computes a the cumulative CE from the value

$$CE_{cum,j} = CE(W(1, j - 1)) \tag{4.5}$$

5. The notes in chunk *j* are then re-spelt so that their names are as close as possible to the hybrid CE where f is a parameter between 0 and 1 that determines the relative weights

$$CE_{hybrid,j} = f.CE_{local,j} + (1-f).CE_{cum,j} \qquad (4.6)$$

The algorithm adopts an initialising strategy of spelling the first chunk using a CE corresponding to the Dn location on the spiral array. The reason for this is that it biases the notation towards fewer sharps and flats. This is identical to the initialisation strategy adopted by Temperley.

4.3.1 Sounding, Starts and Presence

The Chew Pitch Spelling algorithm describes two ways of computing the duration of a note within a chunk; this is whether the note starts or sounds within a chunk. This concept is illustrated using Figure 12.

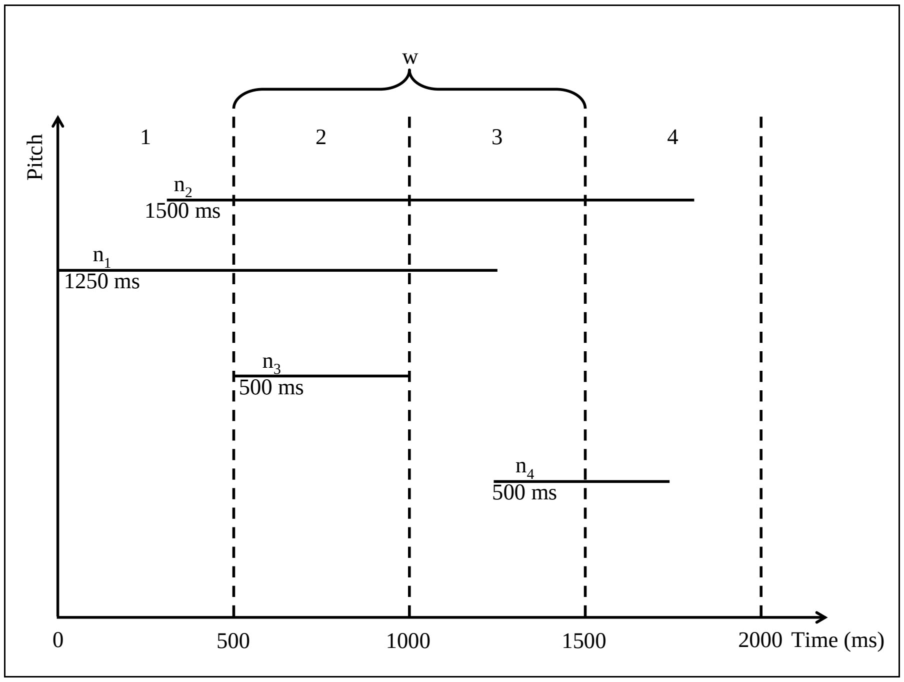


Figure 12: Piano-roll representation of four notes

In a piano-roll representation, the horizontal axis represents time in milliseconds, the vertical axis represents pitch. Each note is represented as a horizontal line (n_1 – n_4) illustrating their start (onset) and end (offset) times and duration. The segment of music is divided into four 500 ms chunks labelled 1 to 4. Taking note n_2 as an example, the note starts in chunk 1 and ends in chunk 4. Once it starts, the note continues to sound until it ends. Therefore we can say that the note *starts* in chunk 1 and *sounds* in chunks 2, 3 and 4. We can calculate *CE(w)* where *w* is the window consisting of chunks 2 and

3. We can further define a note that *starts* within a chunk as one whose start time is greater than or equal to the start time of the chunk and less than the end time of the chunk. We can also define a note that *sounds* in the chunk if its start time is less than the end of the chunk and its end time is greater than the start time of the chunk.

If we consider the notes that *start* in window *w*, then

$$CE(w) = CE(W_{start}(2,3)) = CE(\{n_3, n_4\})$$

Then from equation (4.2)

$$CE(w) = \frac{500.p(n_3) + 500.p(n_4)}{1000} \qquad (4.7)$$

Alternatively, if we consider the notes that *sound* in window *w*, then

$$CE(w) = CE(W_{sound}(2,3)) = CE(\{n_1, n_2, n_3, n_4\})$$

Equation (4.2) evaluates to

$$CE(w) = \frac{1250.p(n_1) + 1500.p(n_2) + 500.p(n_3) + 500.p(n_4)}{3750} \qquad (4.8)$$

In this evaluation, the note is weighted by its duration even if this is greater than the duration of the window *w*. In personal communication, Chew has confirmed however that in their implementation, the note is weighted by the duration for which it sounds within the window. Therefore equation (4.2) should evaluate to

$$CE(w) = \frac{750.p(n_1) + 1000.p(n_2) + 500.p(n_3) + 500.p(n_4)}{2500} \qquad (4.9)$$

These different evaluations highlight the importance of specifying in detail how the CE should be calculated in the algorithm.

A further issue is related to weighting notes by their duration either within the window or their complete duration. For a real time analysis, this requires a processing delay either for the duration of a chunk time or in the worst case for the complete duration of a note. An alternative is to ignore the duration and to weight simply on the *presence* of a note within the window. This only requires the detection of a note's onset and does not require any delay in processing. If this strategy is adopted, then equation (4.2) becomes

$$CE(S) = \frac{\sum_{n \in S} p(n)}{N} \tag{4.10}$$

Where N is the number of notes currently present in the window. Equation (4.9) then simplifies to become

$$CE(w) = \frac{p(n_1) + p(n_2) + p(n_3) + p(n_4)}{4} \tag{4.11}$$

This simplification of Chew's algorithm forms the basis of the novel implementation described in the following section.

4.4 The Implementation

This section describes the implementation of a pitch spelling application based on a simplification of Chew's spiral array algorithm as described in the preceding section. It also describes how the results of the pitch spelling are captured using a subscribing application linked to a relational database that persists the results for later analysis. It also introduces the need for synchronisation between applications within the framework and describes how this can be achieved by use of control messages.

4.4.1 The Pitch Spelling Application

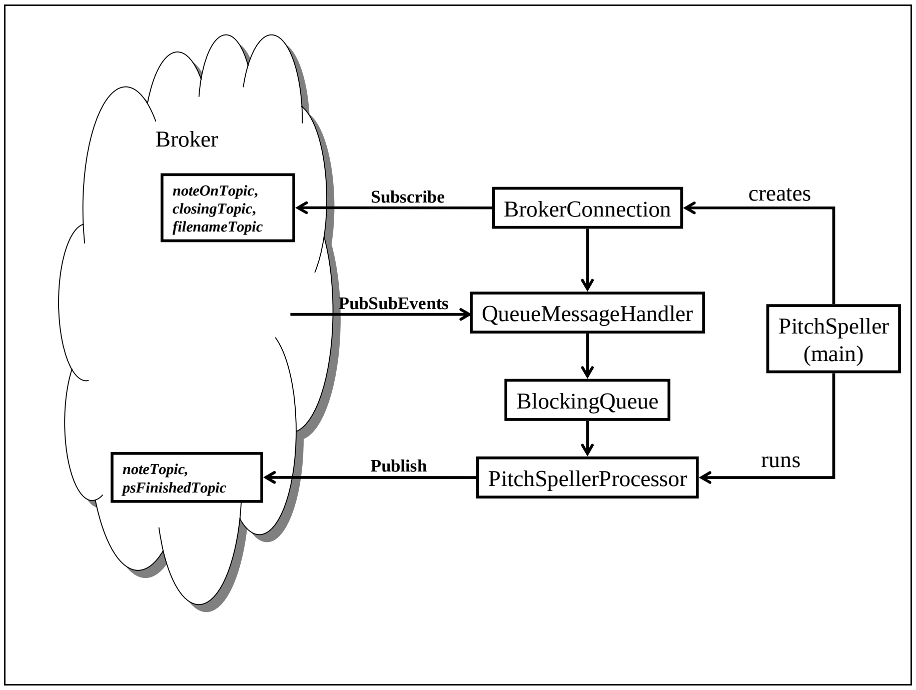


Figure 13: The Pitch Spelling application

The structure of the Pitch Spelling application is illustrated in Figure 13 which is an instance of the generic application structure illustrated in Figure 7 of section 3.5.2. Note that in the thesis figures, topics are referenced by their Java variable names. These are defined as Java Strings which resolve to the hierarchic topic names listed in Appendix A. The application subscribes to the *noteOnTopic* which identifies the start of a note and is published either by a MIDI device client or for testing purposes, a MIDI or OPND file publishing application. In addition, it subscribes to a *filename* topic which identifies which testcase is being published and a *closing* topic used to synchronise the pitch speller with other applications.

The pitch spelling application publishes two events using the *noteTopic* and *psFinishedTopic*. The *noteTopic* is used to publish instances of the *NamedNoteEvent*. This event contains the name of the note as a Java String in the format as the OPND input file; that is the String “Cn3” identifies a C natural in the third octave. In addition to the note name, the event includes the Chew algorithm Centre of Effect that was active when the note was named. This is expressed as a point in three dimension space using Java Float types for the x, y and z components. The event also includes a URI. This URI is the same one as the input *MidiNoteOnEvent* that resulted in the output

NamedNoteEvent allowing the correctness of the spelling to be established as described in a later section. The second event published using the *psFinishedTopic* generated by the pitch spelling application is a confirmation that the application has completed and its use is described in more detail in a later section.

The PitchSpellerProcessor Class

The *PitchSpellerProcessor* class implements the modified Chew pitch spelling algorithm. Its constructor takes a *BlockingQueue* parameter which has been initialised by the main thread and provides an asynchronous interface between the *PitchSpeller* client and the processing thread. The constructor builds an instance of the Spiral Array using the *SpiralArrayBuilder* class. In her algorithm, Chew identifies each pitch name on the array using an index from -15 to +19. The mapping between the indices and the pitch names are shown in the following table:

Index	Pitch Name	Index	Pitch Name	Index	Pitch Name	Index	Pitch Name	Index	Pitch Name
-15	Fff	-8	Ff	-1	F	6	Fs	13	Fss
-14	Cff	-7	Cf	0	C	7	Cs	14	Css
-13	Gff	-6	Gf	1	G	8	Gs	15	Gss
-12	Dff	-5	Df	2	D	9	Ds	16	Dss
-11	Aff	-4	Af	3	A	10	As	17	Ass
-10	Eff	-3	Ef	4	E	11	Es	18	Ess
-9	Bff	-2	Bf	5	B	12	Bs	19	Bss

Table 2: Spiral Array index versus note name

The *PitchSpellerProcessor* class constructor also initialises an array of possible note names. For each of the 12 pitch classes, the array contains the indices of the three possible names for the note. For example, the entry for pitch class 0, note C, contains the indices for its three possible names B# (index 12), C (index 0) and Dbb (index -12). The array is fully defined as:

```
private static final int[][]  noteIndexes = {
     { 12, 0, -12 },  // B#,  C,  Dbb
     { 7, -5, 19 },   // C#,  Db, B##
     { 14, 2, -10 },  // C##, D,  Ebb
     { 9, -3, -15 },  // D#,  Eb, Fbb
     { 16, 4, -8 },   // D##, E,  Fb
     { 11, -1, -13 }, // E#,  F,  Gbb
     { 18, 6, -6 },   // E##, F#, Gb
     { 13, 1, -11 },  // F##, G,  Abb
     { 8, -4, -999 }, // G#,  Ab
     { 15, 3, -9 },   // G##, A,  Bbb
     { 10, -2, -14 }, // A#,  Bb, Cbb
```

```
    { 17, 5, -7 }    // A##, B, Cb
};
```

The option of -999 for G# / Ab is because there is not a third optional name for this note. The purpose of this array is to augment the information for an input note. The note is updated with the three optional names given its pitch class before it is sent to the spelling algorithm. This reduces the search space from the 35 possible names to only the three possible options for each of the 12 pitch classes.

The algorithm described in section 4.3 spells notes contained within a chunk. This approach is adopted in the implementation with the *ChunkBuffer* class managing chunks consisting of a set of notes. Chunks are defined to be a given size in milliseconds (500ms). The *ChunkBuffer* class is initialised through its constructor with parameters that include the global and local context window sizes (8 and 2 respectively) and the relative weight applied between the local and cumulative centre of effect (0.5). The final parameter is the spiral array centre of effect coordinates for the key of D as an initial condition for the pitch spelling algorithm.

Having been initialised, the *PitchSpellerProcessor.addNoteEvent*() method assembles notes as they are input into a *Chunk* based on the note onset time. Once a chunk has been assembled (the current note onset is later than the end time of a *Chunk*), it is passed to the *ChunkBuffer* for naming via the *ChunkBuffer.addCurrentChunk()* method.

The *ChunkBuffer* class constructor initialises a pair of chunk windows corresponding to the global and local windows with sizes set by the constructor initialisation parameters. It also initialises an instance of the *CentreOfEffect* class which computes and maintains the current centre of effect used in note pitch spelling. The *CentreOfEffect* class is initialised to reflect the key of D; the initial condition for Chew’s algorithm.

The pseudo code for the pitch spelling algorithm implemented within the *ChunkBuffer.addCurrentChunk* method is as follows:

1. If the chunk is the first chunk, then name the notes in the chunk using the initial CE (assuming key of D)
    i. Compute the Global CE using the notes within the current chunk
    ii. Name the notes within the current chunk using this CE
    iii. Publish the chunk spellings as a set of *NamedNoteEvent*
2. If the chunk is not the first chunk
    i. Compute the cumulative CE and shift the chunk windows
    ii. Compute the global CE and name the current chunk using this CE
    iii. Compute the local CE using the current chunk and the local chunks window set
    iv. Compute the hybrid CE consisting of a fraction of the local and cumulative CEs
    v. Name the notes using the hybrid CE
    vi. Publish the chunk spellings as a set of *NamedNoteEvent*

When naming a note, the pitch spelling algorithm uses a CE represented as a Java *Point3f* type. This type represents a point in 3D space with the x, y and z coordinates represented as Float types. When a note is input to the *PitchSpellerProcessor* class, the three possible names for the note are assigned as described earlier. Each note name within the spiral array is represented as a point in 3d space. These points are computed during the initialisation of the *SpiralArrayBuilder* class and are themselves of type *Point3f*. Naming a note consists of computing the distance between each of the points on the spiral array for the three possible note names and the current CE. The note name with the minimum distance to the CE is then chosen as the name for the note. The name of the note is then published as a *NamedNoteEvent*.

All events have a URI associated with them. In the cases of raw data such as events published from MIDI input or file players, the URI is created by a *URIGenerator* class. For the *NamedNoteEvent*, the URI contained within the matching input *MidiNoteOnEvent* is reused. This allows the result of the pitch spelling to be matched to a separate MIDI event within the results database. This is discussed further in the next section.

4.4.2 The EventCapture Application

EventCapture is a database application that subscribes to events published by file publisher and analysis applications and archives the events within a relational database. The application is based on the Apache Derby database [Derby]. This is an open source Java database with a small footprint that occupies around 2 Mbytes for the base engine and embedded JDBC driver. It supports SQL standards and can operate in a client/server mode as a network server.

The *EventCapture* application structure is illustrated in Figure 14. The application subscribes to the following topics; *noteOnTopic*, *noteOffTopic*, *noteTopic* and *noteNameFinishedTopic*. The *EventDatabaseProcessor* class constructor creates a JDBC connection to an instance of the Derby database. Each event that needs to persist itself within the database includes an *insert()* method which takes a JDBC Connection object as an input parameter.

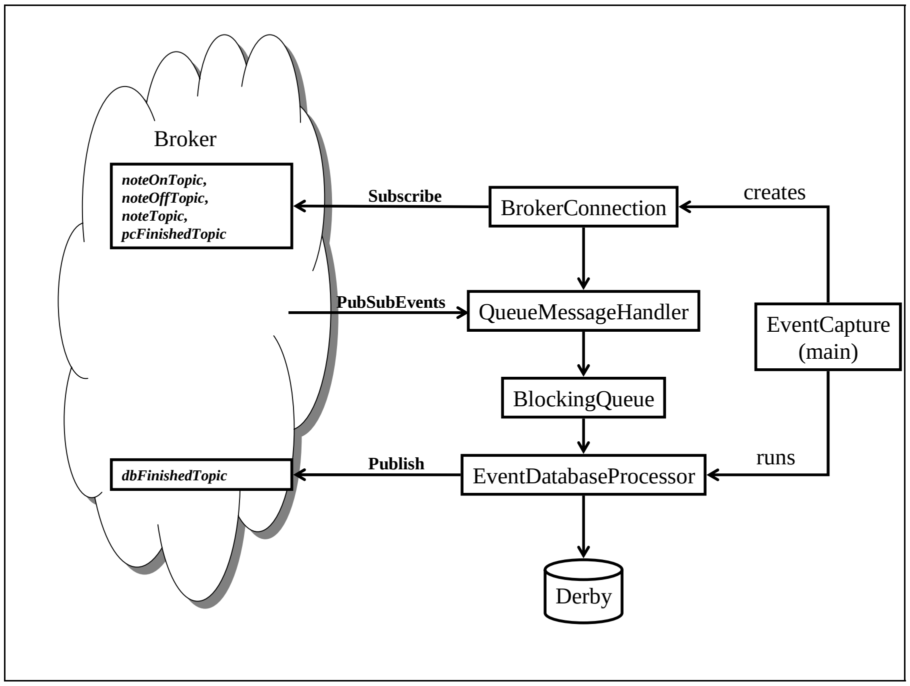


Figure 14: The Event Capture application

The database contains a NOTES table whose schema is created by the following SQL statement:

```
create table NOTES (
     URI          varchar(41) not null,
     TESTNAME     varchar(30) not null,
     ONTIME       bigint,
     OFFTIME      bigint,
     DURATION     bigint,
     MIDINUMBER   integer,
     VELOCITY     integer ,
     NAME         varchar(4),
     NOTENAME     varchar(4)
);
```

A complete row in the NOTES table is assembled by the application invoking the *insert()* method for the *MidiNoteOnEvent*, *MidiNoteOffEvent* and *NamedNoteEvent* as they each arrive asynchronously. The first two events are published by a testcase generator or MIDI input device connected to an input port application. The *NamedNoteEvent* event is generated by the pitch spelling application.

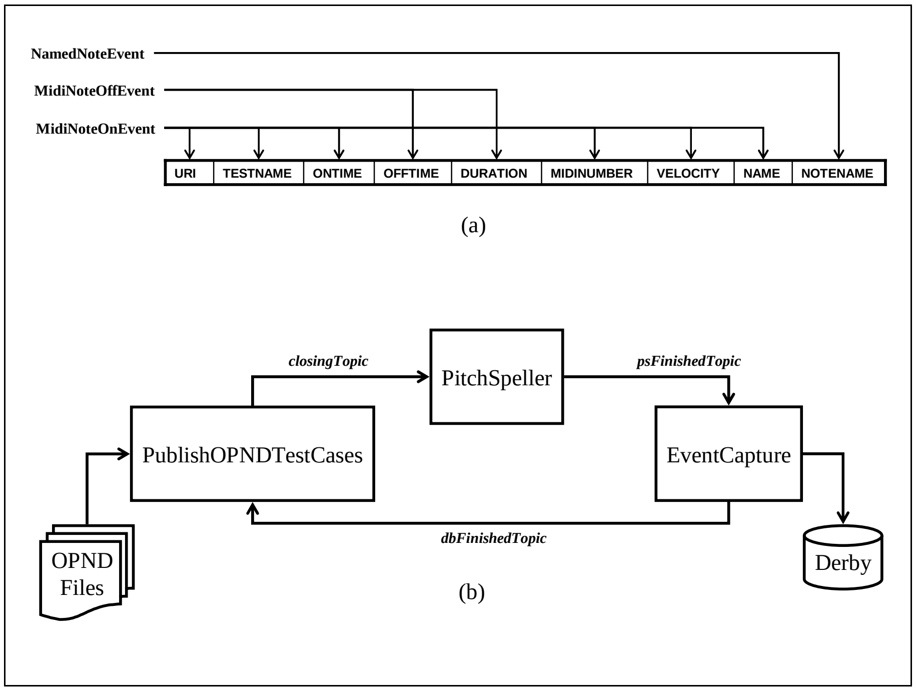


Figure 15: Event Capture and Synchronisation

Assembling results within a relational database

Figure 15(a) above illustrates how each event contributes columns to the NOTES table. For a given note, the first event to be published will be the *MidiNoteOnEvent* identified by the *noteOnTopic*. This will be followed in no particular order by the *MidiNoteOffEvent* identified by the *noteOffTopic* and the *NamedNoteEvent* identified by the *noteTopic*. The *MidiNoteOnEvent.insert()* method uses the SQL insert statement to insert the URI, TESTNAME, ONTIME, MIDINUMBER, VELOCITY and NAME columns using the following code example.

```
public void insert(Connection conn) {
      PreparedStatement pStmt = null;
      String stmtStr = "insert into notes values
                       (?,?,?,0,0,?,?,?,'NONE')";
      try {
            pStmt = conn.prepareStatement(stmtStr);
            pStmt.setString(1, eventURI.toString());
            pStmt.setString(2, testName);
            pStmt.setLong(3, timeStamp);
            pStmt.setInt(4, midiNumber);
            pStmt.setInt(5, velocity);
            pStmt.setString(6, name);
            pStmt.execute();
            pStmt.close();
      }
      catch (SQLException sqlExcept) {
            sqlExcept.printStackTrace();
      }
}
```

Subsequently, the SQL update statement is used to update the row when a *MidiNoteOffEvent* and a *NamedNoteEvent* arrive with the same URI.

The code for the *MidiNoteOffEvent.insert()* method is:

```
public void insert(Connection conn) {
      PreparedStatement pStmt = null;
      String stmtStr = "update notes set OFFTIME=?, DURATION=?-ONTIME
                        WHERE URI=?";
      try {
            // Update notes table
            pStmt = conn.prepareStatement(stmtStr);
            pStmt.setLong(1, timeStamp);
            pStmt.setLong(2, timeStamp);
            pStmt.setString(3, eventURI.toString());
            pStmt.executeUpdate();
            pStmt.close();
      } catch (SQLException sqlExcept) {
            sqlExcept.printStackTrace();
      }
}
```

Note that the DURATION is computed from the inserted ONTIME and the OFFTIME to be inserted. The code for the *NamedNoteEvent.insert()* method is:

```
public void insert(Connection conn) {
      String stmtStr = "update notes set NOTENAME=? WHERE
                        URI=?";
      PreparedStatement pStmt = null;
      try {
            pStmt = conn.prepareStatement(stmtStr);
            pStmt.setString(1, noteName);
            pStmt.setString(2, eventURI.toString());
            pStmt.executeUpdate();
            pStmt.close();
        }
        catch (SQLException sqlExcept) {
            sqlExcept.printStackTrace();
        }
}
```

The reason for the two VARCHAR(4) columns called NAME and NOTENAME will be discussed in the next section.

Application Synchronisation

The testcase set for pitch spelling consists of some 216 files. Whilst it is possible to construct a script consisting of 216 invocations of the OPND file publisher, the pitch spelling and event capture applications, an alternative approach is to use additional control messages to synchronise the three applications. In particular this is necessary for the pitch spelling application since the algorithm must be initialised before a new testcase can be processed.

For all the applications, the processing classes implement the Java Runnable interface; part of the Java Threads support. Inserting a *wait(condition)* statement in the application's *run()* method will cause the application to halt until the condition is satisfied. The Figure 15 (b) illustrates how the three applications are synchronised:

1. The *PublishOPNDTestCases* application publishes the contents of an OPND file terminating the publication with a *closingTopic* event. Following the publication of the closing event, it will halt with the wait condition set to false.
2. The *PitchSpeller* application will process the set of events from the first OPND file and publishes the appropriate note naming events. When it has received the closing event, it completes the spelling by publishing any remaining note information and initialises the algorithm to wait for the next set of OPND generated events. It also publishes an event using the *psFinishedTopic*.
3. The *EventCapture* application uses events from the *PublishOPNDTestCases* and *EventCapture* applications to assemble the NOTE table within the database. Once it has received an event with the *psFinishedTopic,* it can then perform any cleanup processes and wait for the next set of events it has subscribed to and publish an event using the *dbFinishedTopic* to signal that it has completed.

4. The publication of an event using the *dbFinishedTopic* by the *EventCapture* application causes the wait condition halting the *PublishOPNDTestCases* application to be set to true causing the next OPND file to be published.

4.5 Testing and Results

Testing and verification of the pitch spelling algorithm used 216 test files containing nearly 200,000 notes. The files were in the OPND format which consists of a set of triples for the note onset, pitch name and duration. The following opening of the bachbgcant000905m testcase illustrates the format for the first 2 seconds. The first triple 0, An2, 243 encodes an A natural in the second octave starting at time 0 and lasting for 243 milliseconds.

```
0,An2,243
0,En5,118
125,Fs5,118
250,En5,118
375,Dn5,118
500,Cs3,243
500,En5,118
625,Cs5,118
750,Bn4,118
875,An4,118
1000,Dn3,243
1000,An4,118
1000,Fs5,1243
1125,Bn4,118
1250,An4,118
1375,Gs4,118
1500,Dn4,243
1500,An4,118
1625,Fs4,118
1750,En4,118
1875,Dn4,118
2000,Gs3,243
2000,Bn4,1243
```

The testcase set originates from the Musedata corpus and has been generated from the Humdrum **kern format by David Meredith. They are the same testcase set used in his PhD thesis [Meredith07] allowing the performance of this implementation of the spiral array algorithm to be compared with the results contained within his thesis. The initial part of the **kern description from which the OPND file was generated is shown below.

```
!!!COM: Bach, Johann Sebastian
!!!OPR: Es ist das Heil uns kommen her
!!!OTL: Duetto
!!!OMV: 5
!!!SCT: BWV 9
!!!SCA: Thematisch-systematisches Verzeichnis der musikalischen Werke Johann Sebastian
Bach: Bach-Werke-Verzeichnis (Schmieder)
!!!YOR: Bach Gesellschaft Edition i,9
!!!EED: Steven Rasmussen
!!!ENC: Steven Rasmussen
!!!CDT: 1685/3//-1750/7/28/
!!!OCY: Deutschland
!!!YEC: Copyright (c) 1994, 2000 Center for Computer Assisted Research in the
Humanities
!!!YEM: Rights to all derivative editions reserved
!!!YEM: Refer to licensing agreement for further details
!!!YEN: United States of America
**kern  **dynam **kern  **dynam **kern  **silbe **kern  **silbe **kern  **dynam
*I:Flauto traverso      *I:Flauto traverso      *I:Oboe d'amore         *I:Oboe d'amore
        *I:SOPRANO      *I:SOPRANO      *I:ALTO *I:ALTO *I:Continuo     *I:Continuo
*clefG2 *       *clefG2 *       *clefG2 *       *clefG2 *       *clefF4 *
*k[f#c#g#]      *       *k[f#c#g#]      *       *k[f#c#g#]      *       *k[f#c#g#]      *
        *k[f#c#g#]      *
*A:     *       *A:     *       *A:     *       *A:     *       *A:     *
*M2/4   *       *M2/4   *       *M2/4   *       *M2/4   *       *M2/4   *
=1-     =1-     =1-     =1-     =1-     =1-     =1-     =1-     =1-     =1-
(16ee\LL        f       2r      .       2r      .       2r      .       8AA/    f
16ff#\  .       .       .       .       .       .       .       .       .
16ee\   .       .       .       .       .       .       .       8r      .
16dd\JJ)        .       .       .       .       .       .       .       .       .
(16ee\LL        .       .       .       .       .       .       .       8C#/    .
16cc#\  .       .       .       .       .       .       .       .       .
16b\    .       .       .       .       .       .       .       8r      .
16a\JJ) .       .       .       .       .       .       .       .       .
=2      =2      =2      =2      =2      =2      =2      =2      =2      =2
[2ff#\  .       (16a/LL f       2r      .       2r      .       8D\     .
.       .       16b/    .       .       .       .       .       .       .
.       .       16a/    .       .       .       .       .       8r      .
.       .       16g#/JJ)        .       .       .       .       .       .       .
.       .       (16a/LL .       .       .       .       .       8d\     .
```

Given an OPND triplet, it is possible to compute the MIDI note number for the note in a given octave. This is combined with the note onset to generate a *MidiNoteOnEvent* and the duration to compute the note off time to generate a *MidiNoteOffEvent* both of which are published. Both events use the same generated URI to signify they refer to the same note. The *MidiNoteOnEvent* contains an additional variable. This is a String containing the name of the note as contained in the OPND triplet. It is this value that is inserted into the relational database row in the NAME column. The pitch spelling application subscribes to the *MidiNoteOnEvent*, spells the note given the MIDI note number contained within the event and publishes a corresponding *NamedNoteEvent* containing the calculated spelling together with the URI of the input note event. The *NamedNoteEvent* provides the NOTENAME column entry in the database table. A row in the NOTES table now includes both the expected (NAME) and calculated (NOTENAME) spellings for each note in the testcase. Statistics can then be extracted from the NOTES table using appropriate SQL queries.

Initialisation settings within the pitch spelling implementation allow the algorithm to behave as described by pseudo code in Meredith's thesis which itself is an implementation of Chew's work. When running in this mode, the implementation described in this thesis gave the same set of results as reported by Meredith and tabulated in the table below. The table lists the notes that were not spelled correctly. Meredith does not supply sufficient detail to establish whether the same notes were incorrectly spelled. With the simplified algorithm (section 4.3.1), there was an improvement in the spelling accuracy of 0.09% over the published results for the same testcase corpus.

	Bach **24505**	**Beethoven** **24493**	**Corelli** **24493**	**Handel** **24500**	**Haydn** **24490**	**Mozart** **24494**	**Telemann** **24500**	**Vivaldi** **24497**	**Total** **195972**
Meredith	175	311	152	136	365	230	149	147	1665 (99.15%)
Simplified	117	317	100	125	366	206	127	127	1485 (99.24%)

Table 3: Test results for Pitch Spelling Application

4.6 Summary

This chapter has addressed the real-time spelling of notes given only their MIDI note number using a simplified version of Chew's algorithm based on the spiral array. An event based implementation has been described together with applications to capture the published events in a relational database for later analysis. Synchronisation of the applications using control events has been discussed.

The implementation of the pitch spelling algorithm has been tested using a corpus contributed by a previous researcher and the simplified algorithm has been shown to perform marginally better than previously published results.

Current implementations of pitch spelling algorithms by Temperley, Meredith and others are based on batch methods whereby the entire encoded piece of music is analysed in its entirety. The benefit of a real-time, event based implementation is that the algorithm only requires historical data; the notes that have been played up to the current point in time. In addition, the implementation described in this chapter does not rely on the duration of the notes being played thereby eliminating any additional processing delay. These two properties, historical data and delay elimination, lead to an effective real-time implementation of the spiral array pitch spelling algorithm.

Chapter 5 Meter, Key, and Chord

5.1 Introduction

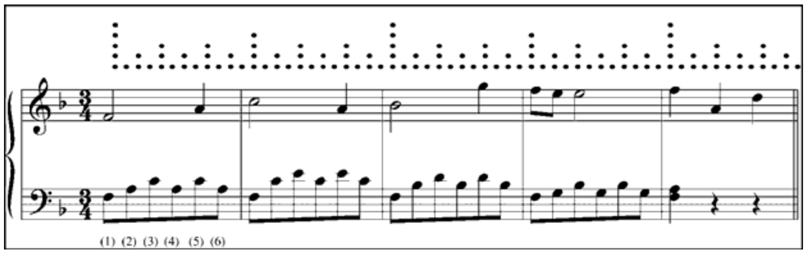
Given the naming of notes described in the previous chapter, the next stage in developing a harmonic model of music is the identification of chord structures within the music and their relationship to the sounding notes. In order to identify and correctly name the chords, it is necessary to detect the underlying key in which the chord sounds. Since the key signature may change during a musical performance, significant points within the music must be identified to delineate musical segments within which the key can be extracted; metrical analysis of the music is used to identify these significant points. Having identified the current key and sounding chords, these provide data for published events which form the basis of the harmonic description described in later chapters.

In this chapter I first review current literature in metrical analysis, key and chord extraction. This is followed by a description of the implementation of the key and chord extraction component of the publish/subscribe framework which includes metrical analysis to segment the music, key and chord identification. Details of the events generated by the component are also provided. Finally, the results obtained are analysed and the chapter summarised.

5.2 Metrical Analysis

Metrical analysis is the identification of the metrical structure of a musical piece. This structure consists of a series of points in time, beats, which relate to events in the music. Beats do not necessarily coincide with events in the music but it is widely accepted that all cognitively significant events within a musical piece occur at a beat.

Perception of metrical structure as an area of research draws contributions from music theory, cognitive psychology and artificial intelligence. Computation models of metrical structure can be distinguished between those that assume their input is taken from a symbolic input and those that operate directly on audio. Temperley [Temperley04]lists some 25 studies that present models of metrical analysis; 4 assume input is from audio, the rest are symbolic This thesis assumes that input to the metrical analysis contains symbolic information; a set of notes which have already been named.



Notelist	Beatlist	Note-Address List
Note 2882 3935 65	Beat 2882 4	ANote 2882 3935 65 10000
Note 2903 3159 53 (1)	Beat 3060 0	ANote 2903 3159 53 10000 (1)
Note 3132 3402 57 (2)	Beat 3132 1	ANote 3132 3402 57 10010 (2)
Note 3392 3652 60 (3)	Beat 3262 0	ANote 3392 3652 60 10100 (3)
Note 3645 3877 57 (4)	Beat 3392 2	ANote 3645 3877 57 10110 (4)
Note 3888 4452 69	Beat 3500 0	ANote 3888 4452 69 10200
Note 3900 4125 60 (5)	Beat 3635 1	ANote 3900 4125 60 10200 (5)
Note 4133 4385 57 (6)	Beat 3760 0	ANote 4133 4385 57 10210 (6)
Note 4412 5477 72	Beat 3888 2	ANote 4412 5477 72 11000
Note 4413 4694 53	Beat 4010 0	ANote 4413 4694 53 11000
Note 4689 4948 60	Beat 4133 1	ANote 4689 4948 60 11010
Note 4936 5164 63	Beat 4270 0	ANote 4936 5164 63 11100
Note 5194 5448 60	Beat 4412 3	ANote 5194 5448 60 11110
Note 5447 6011 69	Beat 4550 0	ANote 5447 6011 69 11200
Note 5456 5677 63	Beat 4689 1	ANote 5456 5677 63 11200
Note 5688 5954 60	Beat 4810 0	ANote 5688 5954 60 11210
.	Beat 4936 2	.
.	Beat 5060 0	.
.	Beat 5194 1	.
	Beat 5320 0	
	Beat 5447 2	
	Beat 5570 0	
	Beat 5688 1	
	Beat 5810 0	
	.	
	.	
	.	

Figure 16: The opening bars of Mozart's Sonata KV332 - from [Temperley04]

Figure 16 illustrates the objective of metrical analysis. Metrical analysis produces a representation of beats aligned with the music that was given as input. Beats occur at points in time and can have different strengths corresponding to the perception of their strength by a listener. Perceptively strong beats exist at higher levels, whereas weaker beats occur at lower levels. In the figure above, the metrical structure is shown above the music as dots representing 5 strength levels (0 – 4). Further reference to this figure will be made later in this chapter.

Metrical analysis has historically been divided into two problems; quantisation and higher-level meter extraction. In quantisation, notes of a performance are adjusted so that the start and end times of notes are multiples of a common time interval thereby eliminating any timing jitter between notes that sound at (nearly) the same time. The second problem starts with a quantised set of notes and develops a multi-level metrical model consisting of a set of different strength beats.

Desain and Honing [Desain92] provide the most important grounding in quantisation. They propose a model which consists of basic units with activation levels representing the inter-onset interval (the time interval between the start of one note and the start of the following note). Interaction units connect adjacent basic units and adjust their relative activation levels so that they are related by simple ratios. This causes the activation levels to converge to multiples of a common value. They introduce the concept of a “sum cell” which sums the activation levels of several basic units which allows a single unit to represent a time interval containing several notes.

More research focus has been on the second problem of metrical analysis; that of developing a multi-level model consisting of a set of different strength beats. Lee [Lee91] proposes a model that ignores note pitch and only considers the music’s rhythmic pattern. The model initialises by finding the interval between the first two note start times. It then generates a second interval of the same length to provide a hypothesis rhythmic level which can be adjusted if necessary. Lee’s model in common with many other ones, only considers a single analysis which is adjusted or discarded as necessary. Other approaches have considered multiple hypotheses of an entire excerpt which are evaluated by different criteria. Simple examples of this type of approach have been reported by Povel and Essens [Povel85] and Parncutt [Parncutt94].

Recent studies incorporate both quantisation and metrical analysis. Examples of these include Chafe [Chafe82] which forms part of a digital audio editing suite, Rosenthal [Rosenthal92] who ranks multiple hypotheses by salience criteria and Large [Large94] who introduces a connectionist model driven by an oscillator which tracks the period and phase of the input.

The work of Temperley and Sleator [Temperley99] forms the basis for the metrical analysis implementation described in this thesis. Their work on preference rules builds on the earlier work by Lerdahl and Jackendoff [Lerdahl83]. The input to this analysis is in the form of a notelist. This is illustrated in Figure 16 and for each note there is an entry which includes its pitch (assuming MIDI note numbers, middle C

= 60) together with the note’s start and end times. Lerdahl and Jackendoff’s model assumes that the metrical structure consists of several beat levels and propose four well-formedness rules that define the set of permissible metrical structures. They are:

1. Every note onset (start) must be marked with a beat
2. Every beat at one level must also be a beat at a lower level
3. Every second or third beat must be a beat at the next level up
4. Beats must be evenly spaced at the tactus level. *This pulse is typically what listeners entrain to as they tap their foot or dance along with a piece of music (Handel, 1989), and is also colloquially termed the 'beat,' or more technically the 'tactus'* [Lerdahl83]

They also propose a set of preference rules that state the criteria where listeners infer the correct structure; for example whether there is a duple or triple relationship between the beats. These are:

1. Prefer structures that align strong beats with note onsets (Event Rule)
2. Prefer structures that align strong beats with the onset of longer notes (Length Rule)
3. Prefer beats at each level to be maximally evenly spaced (Regularity Rule)
4. Prefer to locate strong beats near the beginning of groups (Grouping Rule)
5. Prefer duple over triple relationships between levels (Duple Bias Rule)

Sleator’s implementation strategy for preference rules is described in [Temperley01] and uses principles of dynamic programming to produce an efficient search strategy for the preference rule models used for metrical analysis in their Melisma music analyser. Their algorithms, implemented in C, read an entire notelist file which encodes the music to be analysed. The output of the analysis is a beat list containing a list of beat strengths and the times they occur together with a note address list. The note address list contains the same information as the input note list but with the addition of a note address; a number representing the note’s position in a metrical grid. These are illustrated in the earlier figure.

The dynamic programming approach allows an efficient left-to-right realisation of a preference rule system. This means that at each point in time, the system has a preferred analysis of the music heard so far. This approach of reaching a preferred analysis based only on “historical” notes heard is analogous to the process of human appreciation of music. The implementation developed in Java for this thesis is a modification of this approach to provide an event based implementation. This is described in a later section.

5.3 Key Extraction

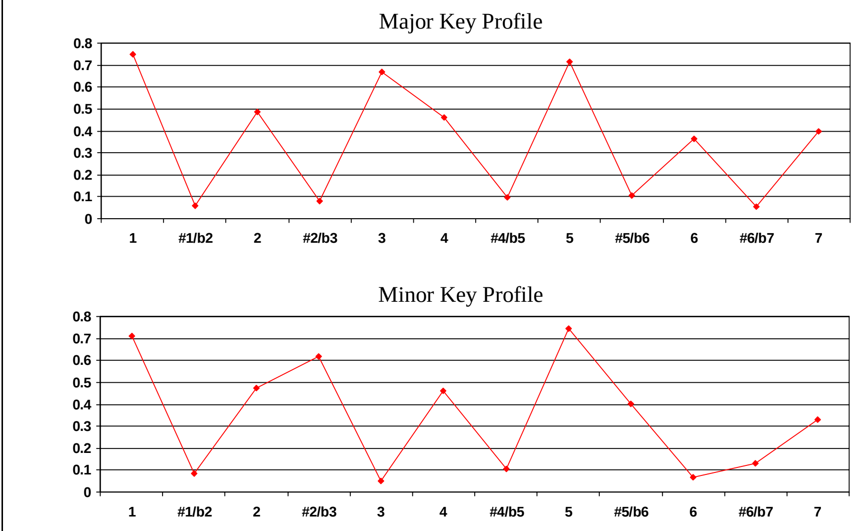
Musical key provides the contextual framework for western tonal music within which notes, chords and harmonies are understood. Musical key supports the perception of elements within a performance providing stability of note pitches and chords. It also increases the perception of melody and through modulation conveys a sense of motion and drama. When analysing music, identifying the key is a precursor to accurately naming the sounding notes and the chords they are part of. In the previous chapter on note naming, the *centre of effect* within Chew’s spiral array algorithm acts as a proxy for the musical key. In this section, the key has to be accurately identified so that the harmonic structure in the form of named chords can be successfully documented.

Models for identifying musical keys have long been a subject of research by the computational musicology community. An early model was proposed by Longuet-Higgins and Steedman [Longuet-Higgins71] for monophonic music. This algorithm processes music in a left-to-right manner eliminating all scales that do not include any pitch it encounters. When it is left with only one key, that is the preferred key. The algorithm has the capability of back-tracking and re-evaluating for cases when all possible keys are eliminated. A similar algorithm was developed by Holtzmann [Holtzmann77] which worked solely on melodies instead of using individual pitches and eliminating keys.

The previously cited work did not however handle modulation when more than one key is present in a musical work. Vos and Van Geenen [Vos96] proposed a model for monophonic music which supports modulation with limited success. More recent research work in modulated key finding has concentrated on a two stage process; segmentation and key finding by the use of key-profiles.

Key profiles are a prototypical pattern of pitches expected in major and minor keys and are represented as a vector of 12 values. Profiles have been proposed based on psychological hearing assessment tests or analysis of large music corpuses. For example, if a piece of music is in C major, then one would expect the notes within the piece to be made up of notes from the C major scale (C,D,E,F,G,A,B) and not expect many notes outside of the scale (C#,D#,F#,G#,A#). The Krumhansl-Schmuckler algorithm [Krumhansl90] uses profiles based on data from experiments by Krumhansl and Kessler [Krumhansl82] where subjects were asked to rate how well a pitch class fitted with a previously defined key established by either a cadence or scale.

Instead of deriving a set of key profiles from psychological tests, an alternative approach is to use actual compositions. The Kostka-Payne corpus is one such set of compositions consisting of 46 excerpts (9057 notes) from the common practice repertoire. It is taken from a workbook accompanying their textbook [Kostka95]. Using its associated instructors manual, Temperley [Temperley07] has derived profiles for major and minor keys using the corpus. The generic profiles are illustrated in Figure 17. The data is interpreted as for example, the note of scale degree 1 (the tonic) occurs in 0.748 (74.8%) of segments in major keys and 0.712 (71.2%) of segments in minor keys. Temperley proposes a Bayesian process to match the set of notes within a music segment to one of 24 major and minor profiles – one of each of the 12 major and minor keys.



Degree	1	#1/b2	2	#2/b3	3	4	#4/b5	5	#5/b6	6	#6/b7	7
Major	0.748	0.060	0.488	0.082	0.670	0.460	0.096	0.715	0.104	0.366	0.057	0.400
Minor	0.712	0.084	0.474	0.618	0.049	0.460	0.105	0.747	0.404	0.067	0.133	0.330

Figure 17: Major and Minor key profiles from the Kostka-Payne corpus

Key finding using profiles assumes that there is a single key within the segment of music being analysed. In a longer piece, this may not be the case due to keys modulating as the music progresses. To overcome this problem, the approach is to segment the music into shorter sections with the assumption that these shorter sections contain a single key allowing the key finding algorithm to be applied. Examples of segmentation approaches include Pardo and Birmingham [Pardo00] who propose a segmentation scheme based on matching tonal structures to a template based on Forte's pitch class representation of chords.

A second segmentation strategy is the one proposed by Temperley [Temperley07] based on the metrical analysis described in an earlier section. Temperley observes that all events within a musical piece occur on a beat and that the more significant events such as modulation occur at stronger beats. Therefore if a metrical analysis is performed on the music, then the times at which higher level beats (levels 3 and 4) occur are the boundaries of meaningful segments. These segments can then be analysed to identify the key within that segment.

5.4 Chord Extraction

Within western music, a chord is a set of different notes that sound simultaneously or occur within a time interval. The identification of chords within a piece of music is fundamental to understanding its harmonic structure and forms the basis of any subsequent processing in the form of arranging, accompaniment and phrasing.

Numerous techniques have been investigated for performing harmonic analysis of music including linguistics [Winograd68], expert systems [Maxwell92], neural networks [Laden89], [Tsui02], hidden Markov models [Raphael03] and structural analysis [Smaill93]. Usually, this work does not consider the segmentation of a musical piece preferring to work with notations that are already segmented. Segmentation prior to harmonic analysis has been addressed by [Pardo02] and [Temperley99].

The work cited so far in this section has concentrated on western tonal music. Tonal music is the organisation of music and harmony around a single, central pitch or key that grew out of renaissance modal music in the 17th century. Music that lacks this tonal centre is referred to as atonal music and became a feature of 20th century composition with composers such as Bartok, Hindemith and Prokofiev creating compositions that musicologists have described as atonal. The analysis of atonal music builds on the work of Forte [Forte73] who proposed the use of a Pitch Class Set (PCS) notation for identifying chords in atonal music. In western music there are 12 pitch classes (0..11) in an octave that can be used to identify pitches independently of octave displacement or enharmonic spelling (C#, Db equivalent in pitch).

A PCS is a list of pitch class numbers enclosed in square brackets that represent the set of sounding pitches; the chord. For example, the PCS [0,3,7] represents a C minor triad and [7,11,2] a G major triad. Some PCS are similar, for example [0,1,4] is a transposed version of [3,4,7]. PCSs that are related through inversion and transformation belong to the same class and can be converted to a prime or canonical form. Therefore the PCS for a C minor triad [0,3,7] is the canonical form for all minor triads irrespective of the key or chord root. Solomon [Solomon82] has proposed a simpler approach to PCSs based on Forte's original work and has generated tables of relationships and properties for all PCSs which is available on the web [Solomon].

Nelson [Nelson04] provides further background to the mathematics underpinning pitch class sets. Table 4 illustrates the first 20 entries in Solomon's table of pitch class sets. The complete table contains 351 entries and includes all chords, not just the unison, intervals and triads listed in Table 4. The table lists Forte's set name, the PCS prime form, its interval vector (a set of numbers that summarise the intervals within the chord) and a description of the chord.

#	Forte cross-referenced Set-name	Prime	Interval Vector	Descriptive name/properties
0	0-1	Empty	000000	Null set
1	0-1*	0	000000	Unison
2	2-1*	01	100000	Semitone
3	2-2*	02	010000	Whole-tone
4	2-3*	03	001000	Minor Third
5	2-4*	04	000100	Major Third
6	2-5*	05	000010	Perfect Fourth
7	2-6*(6)	06	000001	Tritone
8	3-1*	012	210000	BACH /Chromatic Trimirror
9	3-2	013	111000	Phrygian Trichord
10	3-2B	023	111000	Minor Trichord
11	3-3	014	101100	Major-minor Trichord.1
12	3-3B	034	101100	Major-minor Trichord.2
13	3-4	015	100110	Incomplete Major-seventh Chord.1
14	3-4B	045	100110	Incomplete Major-seventh Chord.2
15	3-5	016	100011	Rite chord.2, Tritone-fourth.1
16	3-5B	056	100011	Rite chord.1, Tritone-fourth.2
17	3-6*	024	020100	Whole-tone Trichord
18	3-7	025	011010	Incomplete Minor-seventh Chord
19	3-7B	035	011010	Incomplete Dominant-seventh Chord.2

Table 4: Section of Solomon's table of Pitch Class Sets

The prime form of a PCS may produce confusing results. For example, the prime form for both a major triad [0,4,7] and a minor triad [0,3,7] both map to the same prime form (037). However, both triads are inversions of each other so that they do indeed map to the same prime form. Solomon observes that if the final matrix inversion is omitted when computing the prime form, then the major and minor triads together are distinguishable as are other inversion equivalences. This fact is used in the implementation described in the following section.

5.5 The Implementation

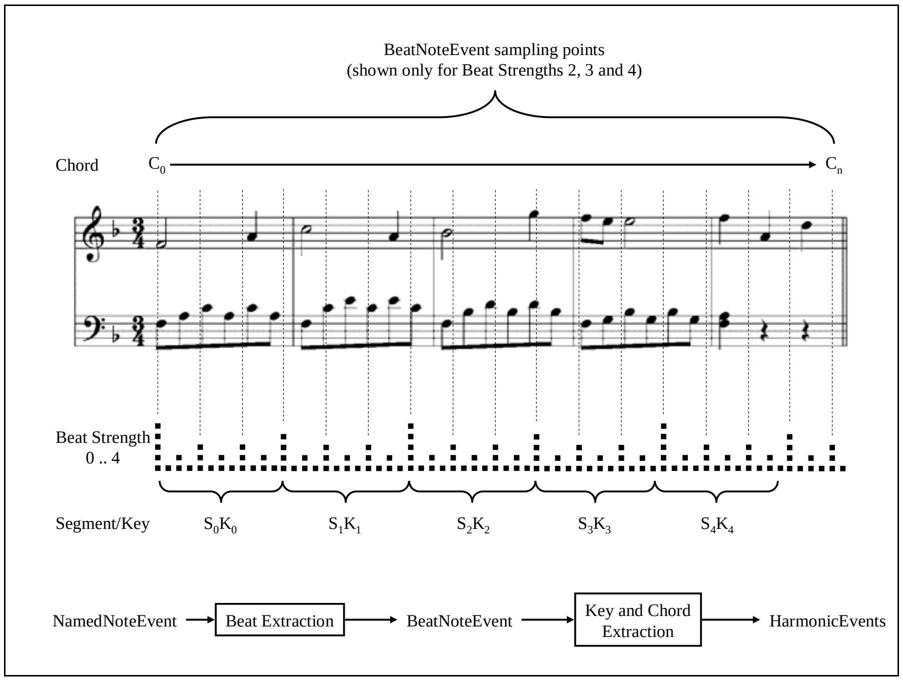


Figure 18: Key and Chord extraction

Extracting key and chord information is a two-step approach involving two applications. The first is an event based version of Temperley's meter application which subscribes to *NamedNoteEvents* published by a pitch spelling application or simulated by the OPND testcase publisher. The application publishes *BeatNoteEvents* which contain three pieces of information:

1. The beat time in milliseconds from the start of the piece
2. The strength of the beat in a range from 0 to 4
3. The set of notes that are sounding when the beat occurs

The *BeatNoteEvent* may therefore be thought of as a sample at a point in time containing all the notes that are currently sounding in the piece. *BeatNoteEvents* are published for each beat though for clarity, only the sampling points for beats of strength 2 to 4 are shown in Figure 18. The key and chord extraction application subscribes to the published *BeatNoteEvents*.

The second application establishes the key(s) and identified chords. As described in section 5.3, key extraction is a two stage process. Firstly the piece must be partitioned into segments with the assumption that the key does not change within a segment. Given a segmented piece, the key can then be identified for each segment using an appropriate algorithm. The segmentation strategy adopted in this implementation is that proposed by Temperley [Temperley07] with segments identified by high beat strength. A high beat strength (strength >= 3) assumes that the beat occurs at a cognitively important event; one which may signify a modulation.

With reference to Figure 18, the structure of the key and chord extraction implementation is as follows:

1. Segment S_0 starts at time t = 0 and ends when a *BeatNoteEvent* arrives with strength >= 3. The final segment S_{final} ends when the beat extraction application signals the end of the piece by publishing a *ClosingEvent*. Otherwise segments start and end when a *BeatNoteEvent* arrives with strength >= 3.
2. A segment S_n is assembled by storing the set of *BeatNoteEvents* contained within it. This will include all notes sounding within the segment. An empty segment S_n is also initialised with the key identified for the previous segment S_{n-1}, K_{n-1} and the chord sounding at the end of the previous segment.
3. When the end of a segment S_n is identified by the arrival of a *BeatNoteEvent* with strength >= 3 or the end of the piece, the segment is processed to identify the segment key K_n and the set of chords present within the segment.

Key extraction uses the Bayesian matching technique proposed by Temperley of the notes present within a segment to key profiles generated from the Kostka-Payne corpus. The start and end times for a key are identified by detecting when the current key changes and *KeyStartEvent* and *KeyEndEvents* are published. A key may exist for a period longer than a single segment so initialising a segment with the key identified in the previous segment allows only changes in key within the piece to generate published events.

Chord identification uses a modified PCS approach as described by Forte. A *PitchSet* class has been developed which given a vector of note pitch classes, computes the Forte normal form. In addition, the semi-normal form (the Forte normal form without the final inversion) is also computed to allow for example, major and minor triads to be differentiated. The *PitchSet* class also includes a method to return the root pitch class of the chord. The operation of the *PitchSet* class is illustrated by the following test code.

```
public class Test3 {
     public static void main(String[] args) {
            int[] notes = new int[]{2,5,10};

            printArray("notes = ", notes);

            PitchSet ps = new PitchSet(notes);
            PitchSet nForm = ps.getNormalForm();
            PitchSet pForm = ps.getPrimeForm();
            PitchSet spForm = ps.getSemiPrimeForm();
            int root = ps.getRoot();
            printArray("normal form = ", nForm.noteArray);
            printArray("prime form = ", pForm.noteArray);
            printArray("semi-prime form = ", spForm.noteArray);
            System.out.println("root pc = " + root);

     }

     private static void printArray(String preA, int[] iA) {
            int aL = iA.length;
            System.out.print(preA);
            for (int i = 0; i < aL; i++) {
                  System.out.print(iA[i] + " ");
            }
            System.out.println();
     }
}
```

Consider a Bb major triad consisting of the notes Bb, D and F but re-ordered into D, F, Bb. The set of pitch classes for these three notes is (2, 5, 10). Given this triplet, represented as a Java int array, we would like to know the different forms of the chord and its root pitch class. Running this test produces the following output:

```
     notes = 2 5 10
     normal form = 10 2 5
     prime form = 0 3 7
     semi-prime form = 0 4 7
     root pc = 10
```

Given the input array, it identifies the normalized form (10 2 5), the prime form [0,3,7] which identifies a minor triad and the semi-prime form [0,4,7] which disambiguates the pitch set to a major triad. The root of the triad is identified as pitch class 10, the Bb. Given the semi-prime form and the root for set of notes within a segment, the chord name can then be identified.

Chord names are identified using the *ChordDescriptionTable* class. This Java class is a lookup on a hash table keyed on the semi-prime form. The hash table contains entries of the Java *ChordDescription* type whose private variables are strings containing the chord’s semi-prime form, Forte number, interval vector, name, quality and a description generated from Solomon’s table published on the web.

The output of the chord identification can be illustrated by the following segment information generated by the application. Note that this is a single segment from a longer running testcase.

```
Start = 19985 End = 21000 # notes = 13 Key = An +
19985(4) Fs2 Cs5          [1,6] (0,5) <0,0,0,0,1,0>  2-5 Cs           Perfect Fourth
20125(0) Gs2 Bn4 En5      [4,8,11] (0,4,7) <0,0,1,1,1,0>  3-11B En    Major Triad
20265(1) Bn2 Dn5          [2,11] (0,3) <0,0,1,0,0,0>  2-3 Bn          Minor Third
20370(0) Bn2 Dn5          [2,11] (0,3) <0,0,1,0,0,0>  2-3 Bn          Minor Third
20510(2) En2 Gs5          [4,8] (0,4) <0,0,0,1,0,0>  2-4 En           Major Third
20615(0) En2 Gs5          [4,8] (0,4) <0,0,0,1,0,0>  2-4 En           Major Third
20755(1) Gs5 Dn3          [2,8] (0,6) <0,0,0,0,0,1>  2-6 Dn           Tritone
20860(0) Gs5 Dn3          [2,8] (0,6) <0,0,0,0,0,1>  2-6 Dn           Tritone
21000(3) Gs5 Cs3 En4      [1,4,8] (0,3,7) <0,0,1,1,1,0>  3-11 Cs      Minor Triad
```

This segment starts at time 19985ms and ends at 21000ms and contains 13 different notes whose profile indicates a key for the segment of A natural major. The plus sign following the key shows that this segment’s key is a continuation of the key from the previous segment. There then follows a detailed listing of each beat within the segment. Each beat has strength from 0 to 4. This is shown within brackets following the beat time in milliseconds; the segment is defined from the higher strength beats at times 19985ms (strength = 4) and 21000ms (strength = 3).

Taking the beat at 20125ms as an example we see that three notes are sounding at the beat time. These notes define the pitch class set [4,8,11] which resolves to the semi-prime form (0,4,7) and interval vector <0,0,1,1,1,0> corresponding to the Forte number 3-11B. Although the note name includes an octave, this is irrelevant when identifying the chord and we can establish that the root of the chord is E natural. Using the semi-prime form as the hash lookup into the *ChordDescriptionTable*, this chord is identified as a major triad. Re-ordering the three sounding notes into En, Gs and Bn shows that this is the case.

The application together with other publishers results in a set of events that identify:

1. The start and end of the piece of music
2. The start and end of notes including the name of the note
3. The start and end of keys including the name and mode
4. The start and end of chords including its type, root and links to the notes contained within the chord.

These events will be described in more detail in the next chapter where the semantic description of the music's harmonic structure is addressed.

5.6 Summary

In this chapter I have discussed the relevant supporting work for metrical analysis, key identification and chord extraction. These form the basis of two applications to extract metrical beats and from these determine the keys within the piece and associated chords. The applications use a novel sampling approach based on the extracted beats to establish which notes are sounding simultaneously thereby allowing chords to be identified. Techniques for manipulating pitch class sets have been discussed which allow hash table lookups to be used to identify the properties of the sounding chords.

There is a problem in identifying high quality test data to validate the key and chord identification algorithms developed in this chapter. For testing, test data that includes the notes together with chord and key information annotated by an expert musicologist is needed so that the algorithm results may be compared with expert analysis.

Chapter 6 A Semantic Representation of Musical Harmony

6.1 Introduction

Previous chapters have described the naming of notes and the identification of keys and chords. This chapter describes how these musical objects and their relationships are captured and represented within a semantic model of musical harmony. Starting with a description of semantic web technologies, this chapter continues with a review of current techniques for representing musical structure using semantic web languages; notably the Resource Description Framework (RDF) and the Web Ontology Language (OWL). It then develops a semantic representation of musical harmony that is constructed from events generated by the applications described in earlier chapters of this thesis.

6.2 The Semantic Web and the representation of Musical Harmony

The World Wide Web has evolved from its original form as a web of documents to be consumed by humans into a web of data that can be consumed by machines. This evolving Web, the Semantic Web [Berners-Lee01], can now be thought of as a repository of knowledge rather than as an information store consisting of different document types. Berners-Lee, Hendler and Lassila have described the Semantic Web as *an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.*

From a knowledge management point of view, current web technologies, when applied to weakly structured documents such as text, audio and video, suffer from limitations[Antoniou08]:

1. Searching for information is based on keywords; a technique with known limitations
2. Extracting information is time consuming since any retrieved documents have to be examined for relevant information
3. Maintaining information is problematic due to inconsistencies in terminology and failure to remove outdated information
4. Uncovering information that implicitly exists in documents can be extracted using text and data mining but this is difficult for information in weakly structured collections of documents
5. Viewing information by groups of users is easily managed when the information is contained within a database but is harder to manage over a web based infrastructure

The objectives of the Semantic Web through its tools, organisation and descriptive languages are to enable more advanced knowledge management through:

1. Organising knowledge within semantic, or conceptual, domains according to its meaning
2. Developing tools to automate the extraction of knowledge and checking for conceptual inconsistencies
3. Replacing keyword based searching by query answering whereby knowledge can be retrieved and presented in a way that is semantically meaningful. This will be through semantically relevant entities (e.g. notes, keys and chords in the musical domain) and their inter-relationships
4. Developing techniques for inferencing and reasoning across multiple semantic domains. These domains are represented as ontologies.

The emergence of the Semantic Web like other research domains has given rise to its own terms, technologies and tools. Many of these terms will be used later in this thesis so for completeness they are defined here:

1. An *Ontology* is a semantic model consisting of entities and the relationships between them. Both entities and relationships may have attributes assigned to them.
2. A *Triple* is a fact consisting of a subject, predicate and object. The predicate defines the relationship between the subject and object. For example “John knows Dave” is a *Triple*; John is the subject, Dave is the object and knows is the predicate.
3. A *Triple Store* is a database of facts. The term *Triple Store* has wider meaning in the context of the semantic web than simply a database of triples in that it is also able to process the rules inherent in a semantic representation such as OWL. In this thesis, a relational database is used to store triples generated by the music analysis. This database will be referred to as a *triple database* to differentiate it from a *Triple Store*.
4. *Inference* or *Entailment* is a logical process in which rules are applied to a set of facts in order to deduce (or infer) additional facts. For example, if “John knows Dave”, then it may be inferred that there is an inverse relationship “Dave knows John”.
5. The *Resource Description Framework* (RDF) [RDF04] is an activity by the World Wide Web Consortium W3C which completed in 2004. The RDF specifications provide syntax for a lightweight ontology system to support the exchange of knowledge on the Web. Included in the framework is the RDF Vocabulary Description Language *RDF-Schema* (RDFS) which forms the basis of other ontology languages and is expressed in XML.
6. The *Web Ontology Language* (OWL) [OWL04] builds on RDF and RDF Schema and provides a more extensive vocabulary for describing properties and classes: among others, relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes.
7. The SPARQL Protocol And RDF Query Language (*SPARQL*) [SPARQL07] can be used to express queries across diverse information sources, whether the data is stored natively as RDF or viewed as RDF via middleware. *SPARQL* contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. The results of *SPARQL* queries can be results sets or RDF graphs.
8. Logical rules may be expressed in a number of different formats including the Semantic Web Rules Language (SWRL) [SWRL04].

9. Several tools frameworks have been developed in Java to support the Semantic Web. The most popular in research communities are the *Protégé Ontology Editor* [Protege] and the *Jena Semantic Framework* [Jena].
10. To support the tools frameworks, there are a number of inferencing (or reasoning engines) in widespread use. The most popular ones being *Pellet* [Pellet], *FaCT* [FaCT] and *Racer* [Racer].

With the maturity of semantic web technologies, their use for annotating music through metadata is becoming widespread. Musicbrainz [Swartz02] provides an open source repository of information about artists, albums and song titles using RDF. The openness of the RDF representation allows the metadata to be re-purposed so that other applications and websites can link to the Musicbrainz metadata to enhance their own information which could include where to purchase the albums.

A more widespread approach to music metadata using ontologies has been described in [Raimond2006] and developed further in [Raimond2007a] and [Raimond2007b] as the Music Ontology. This addresses metadata which includes editorial, cultural and acoustic information. The ontology contains three levels of expressiveness which cater for the wide range of granularity required to document musical events. Level 1 deals with purely editorial material matters such as relating artists to albums and albums to tracks. Level 2 introduces the concept of an event which allows the workflow associated with a musical work such as its composition, arrangement, performance and recording to be documented. Finally, level 3 introduces event decomposition to allow finer granularity events and sub-events to be documented. These events could include characteristics of an audio signal (waveform, spectral etc), settings used throughout a recording session or harmonic events resulting from an analysis and encoding of the musical score. To support levels 2 and 3, specific Event [Raimond07d] and Timeline[Raimond07c] ontologies are defined together with the use of the Friend-of-a-friend (FOAF) ontology [FOAF05] to reuse its concepts of person and group. The representation of musical harmony presented in this chapter could be considered as part of level 3 in the music ontology.

6.3 The Representation of Time

The need to represent time in a consistent way is a recurring problem in any application domain. The music ontology timeline adopts Allen's [Allen83] approach and is a compromise that makes a formalism which reflects the way time is actually used in natural language. It assumes that events are expressed as time intervals of various sizes rather than explicit time points.

Relation	**Symbol**	**Inverse Symbol**	**Endpoint Relationships**	**Representation**
X *before* Y	<	>	X+ < Y-	XXXX YYYY
X *equal* Y	=	=	(X- = Y-) & (X+ = Y+)	XXXX YYYY
X *meets* Y	m	mi	X+ = Y-	XXXXYYYY
X *overlaps* Y	o	oi	(X- < Y-) & (X+ > Y-) & (X+ < Y+)	XXXX YYYY
X *during* Y	d	di	((X- > Y-) & (X+ <= Y+)) | ((X- >= Y-) & (X+ < Y+))	XXXX YYYYYYYY
X *starts* Y	s	si		XXXX YYYYYYYY
X *finished* Y	f	fi		XXXX YYYYYYYY

Table 5: Base Temporal Relationships

It also assumes that our perception of temporal knowledge is relative and identifies thirteen possible relationships; these are identified in tabular form.

Relationships exist between two events X and Y. Each event has a start time and an end time denoted by – and +. In the table, an event X starts at time X- and ends at time X+. A relationship and its inverse are each assigned a symbol. An expression in terms of the event endpoints is provided together with a pictorial example. For example, in the case of the X *before* Y relationship where the start of event Y occurs after the end of event X, the inverse may be interpreted as Y *follows* X. The three *during* relationships *during*, *starts* and *finishes* can be collapsed into a single *during* relationship. This can be represented by a single expression as illustrated in the table. However this does not preclude the three separate relationships being used for finer granularity qualification of event relationships. The inverse of the three *during* relationships provide three *containment* relationships which may be collapsed into a single *contains* relationship.

6.4 The Representation of Musical Events

The temporal relationships identified in the previous section can be used to formally express the relationships between harmonic objects in a musical piece.

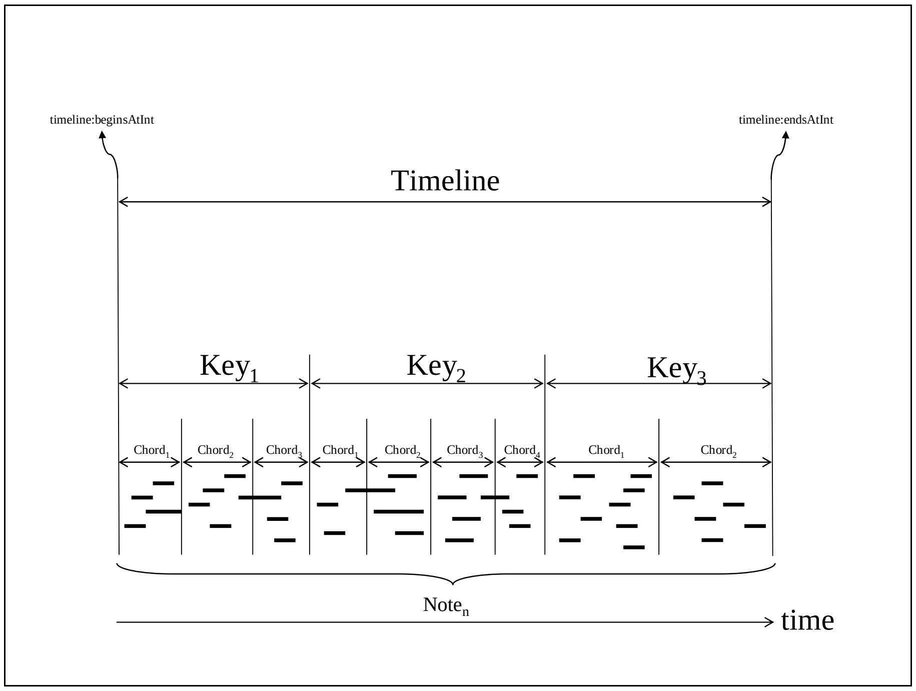

timeline:beginsAtInt
timeline:endsAtInt
Timeline
Key_1
Key_2
Key_3
$Chord_1$ $Chord_2$ $Chord_3$ $Chord_1$ $Chord_2$ $Chord_3$ $Chord_4$ $Chord_1$ $Chord_2$
$Note_n$
time

Figure 19: Objects of a Harmonic Ontology

Each musical piece is represented as a Timeline with specified starting and ending times. It consists of an ordered set of Keys, Key_k where k = 1..K. The relationships between these Keys and the Piece are defined as:

Piece *starts* Key_1		(5.2)
Piece *finishes* Key_K		(5.3)
Key_k *during* Piece	$(1 < k < K)$	(5.4)
Key_k *meets* Key_{k+1}	$(1 <= k < K)$	(5.5)

These describe the temporal relationships between a Piece and its constituent Keys. The first Key starts at the same time as the Piece (5.2) with the final Key ending at the same time as the Piece (5.3). All other Keys are contained within the Piece (5.4). The final relationship (5.5) states that Key changes occur instantaneously at an explicit time point; the end of Key_k occurs at the start of Key_{k+1}. The start and end times for each Key are defined as events occurring within the Piece Timeline (startsAtInt and endsAtInt expressed as integer time points).

A Key can contain an ordered set of Chords Chord_c where c = 1..C; it is assumed that an instance of a Chord cannot be contained within more than one Key. If so, then a similar set of relationships (5.2 to 5.5) exist between a Key and its constituent Chords.

Key *starts* Chord_1		(5.6)
Key *finishes* Chord_C		(5.7)
Chord_c *during* Key	$(1 < c < C)$	(5.8)
Chord_c *meets* Chord_{c+1}	$(1 <= c < C)$	(5.9)

Notes do not have such strict containment relationships with their parent Chords and Pieces. We can assert that a Piece consists of an ordered set of Notes, Note_n where n = 1..N. There are no *starts* or *finishes* relationship between a Note and a Piece since rests may occur at the start and end of a Piece. Similarly the relationships do not exist between a Note and its parent Chord since a Note may be sustained between multiple Chords. A single composite relationship between a Note and a Chord is expressed as:

(Note_n *equal* Chord_c) |
(Note_n *overlaps* Chord_c) |
(Note_n *during* Chord_c) (5.10)

Expression (5.10) states that a Note can exist for the same time as a parent Chord through the *equal* relationship, be shared with another Chord through the *overlaps* relationship or sound during a Chord through the *during* relationship. A similar set of relationships can be asserted between Notes to establish their temporal links.

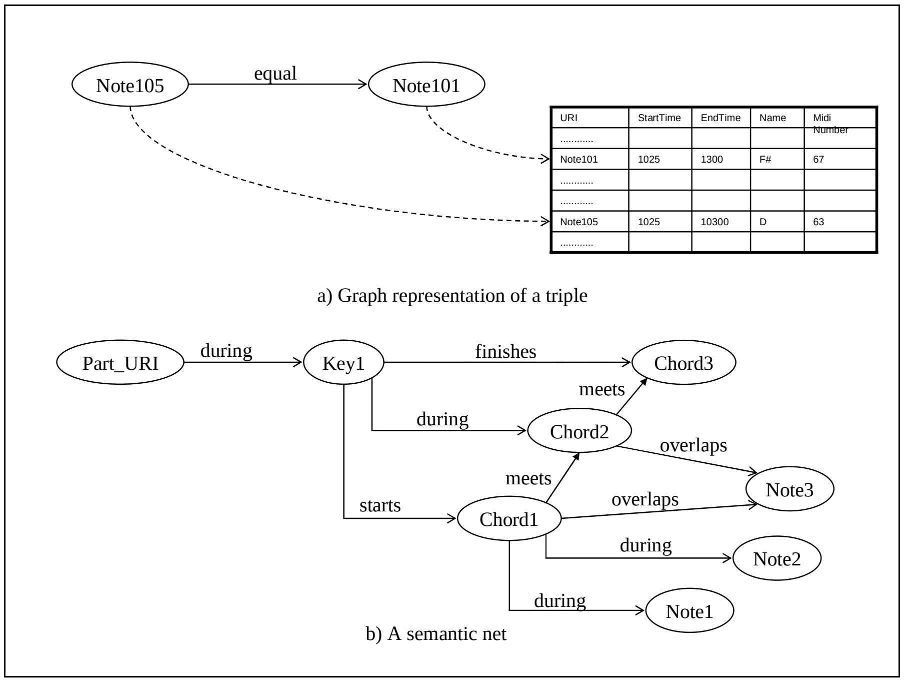
(Note_m *equal* Note_n) |

$(\text{Note}_m \textit{ overlaps } \text{Note}_n)$ |

$(\text{Note}_m \textit{ during } \text{Note}_n)$ (5.11)

6.5 Semantic Representation

In the previous discussion, the objects having relationships can be thought of as resources, each having a unique identifier. This identifier is typically a URI or Universal Resource Identifier. A URI does not imply any access mechanism to the resource in a web context; it is simply an identifier for a resource. In the context of the Semantic Web, the URI is the identifier of a Web resource.



URI	StartTime	EndTime	Name	Midi Number
....				
Note101	1025	1300	F#	67
....				
....				
Note105	1025	10300	D	63
....				

Figure 20: Representation of a Semantic Net

The building block of a semantic description is the relationship between two resources. This is illustrated in the Figure 20 (a). In this example, two Notes start and finish at the same times; therefore they are related by the temporal *equal* relationship. The Notes are each identified by a URI *Note101* and *Note105* which are represented as nodes on a graph. The *equal* relationship is the arc connecting the two nodes. This triple *(x, P, y)* can be thought of as a logical formula *P(x,y)* where the binary predicate *P* relates the resource *x* to the resource *y*. The graph illustrated is known as a Semantic Net and forms the basis of the Resource Description Framework; an XML vocabulary for describing Semantic Nets.

As stated previously, each resource is identified by a URI. In the model developed as part of this project, the URI corresponds to a unique key in the data model that describes the different harmonic objects that have been extracted from the MIDI stream. In the illustrated example, *Note101* and *Note105* are unique database keys into the URI column of a table of *Notes*. By selecting the rows corresponding to these URIs, further information about the *Notes* can be accessed such as its name, MIDI number, start and end times.

Part b) of the figure illustrates how a more complex graph can be assembled that describes the relationships between a *Part*, and its constituent *Keys*, *Chords* and *Notes*. The graph can be interpreted as:

> *A Part contains a single Key (Key1) that occurs within the start and end time of the Part. Key1 contains three Chords (Chord1, Chord2 and Chord3) with Chord1 starting at the same time as the Key, Chord 2 follows immediately then Chord3 which finishes at the end of the Key. Chord1 contains three Notes (Note1, Note2 and Note3). Note3 is sustained so that it also contributes to Chord2. No Notes are shown for Chord3.*

The representation of harmonic objects using RDF can permit queries of the graph to identify particular harmonic structures. For example, a Plagal cadence is a IV chord followed by a I (root) chord. This may be represented by a query of the form

(Chord1 *meets* Chord2) *where* ((Chord1 *type* IV) and (Chord2 *type* I))

6.6 The Harmony Ontology

The Harmony OWL ontology developed in this thesis uses simpler temporal relationships between Keys and Chords.

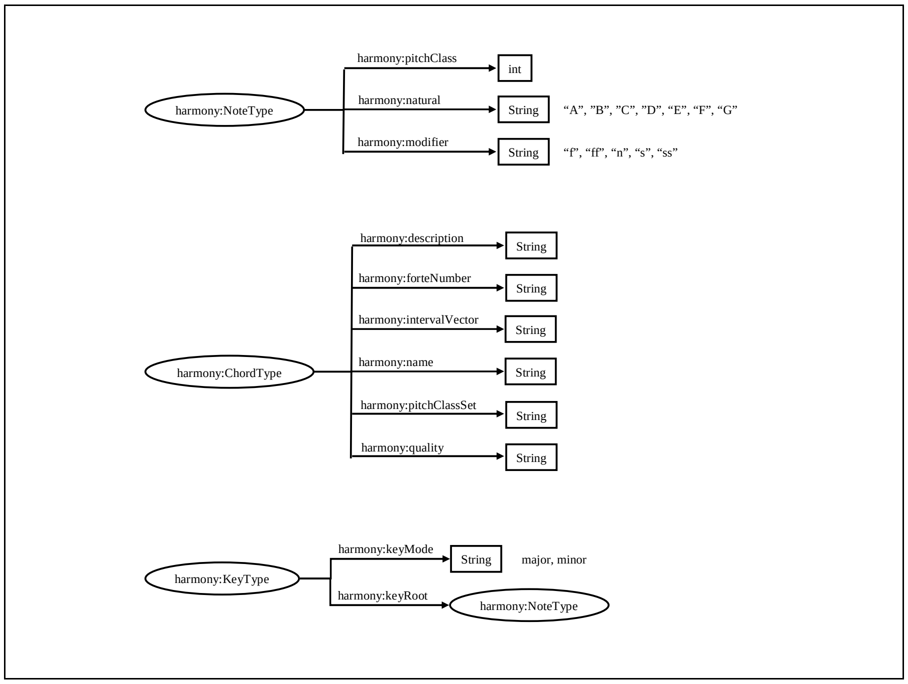


Figure 21: Harmony object types

It has been developed using the Protégé ontology editor and consists of six classes separated into three type classes, *NoteType*, *ChordType* and *KeyType* and three objects Note, Chord and Key. The three type classes are shown in the figure above.

The *NoteType* class has three data properties; *pitchClass*, *natural* and *modifier*. The *pitchClass* property contains the pitch class of the note which is an integer ranging from 0 to 11 with C = 0 and B = 11. Note that this does not take into account whether for example the B is harmonically a Cb or A##. The *natural* property is the name of the note as a String restricted to the values “A” to “G”. The third property is the note *modifier* which identifies whether the note is a flat, double flat, natural, sharp or double sharp. The ontology defines 35 individuals (or instances of) *NoteTypes*; one for each of the 7 naturals with 5 modifiers. For example, the C natural *NoteType* individual has its properties set to (0, “C”, “n”) and the G sharp individual has the properties (8, “G”, “s”).

Chords have a more complex set of properties reflecting the Forte number, pitch class set and interval vector described in the previous chapter. To these are added the description, name and quality properties to further describe the chord. As with the *NoteType*, a set of *ChordType* individuals have been defined in the ontology. One *ChordType* individual has been defined for each of the 351 chord types identified by

Solomon. Each *ChordType* has been named using its pitch class set prefixed with C so that the *ChordType* for a major triad (pitch class set “0,4,7”) is named C047. Therefore this individual will have the properties *forteNumber* = “3-11B”, *intervalVector* = “001110”, *name* = “Triad”, *pitchClassSet* = “0,4,7”, *quality* = “Major” and *description* = “Major Chord”. Only a common subset of the chord types has been fully defined; chords such as C02348, an Augmented Pentacluster, have still to have their name and quality assigned.

The *KeyType* has a simpler set of properties. The *keyRoot* property is a *NoteType* with the *keyMode* property set to either “major” or “minor”. The ontology does not include any individuals for *KeyTypes*.

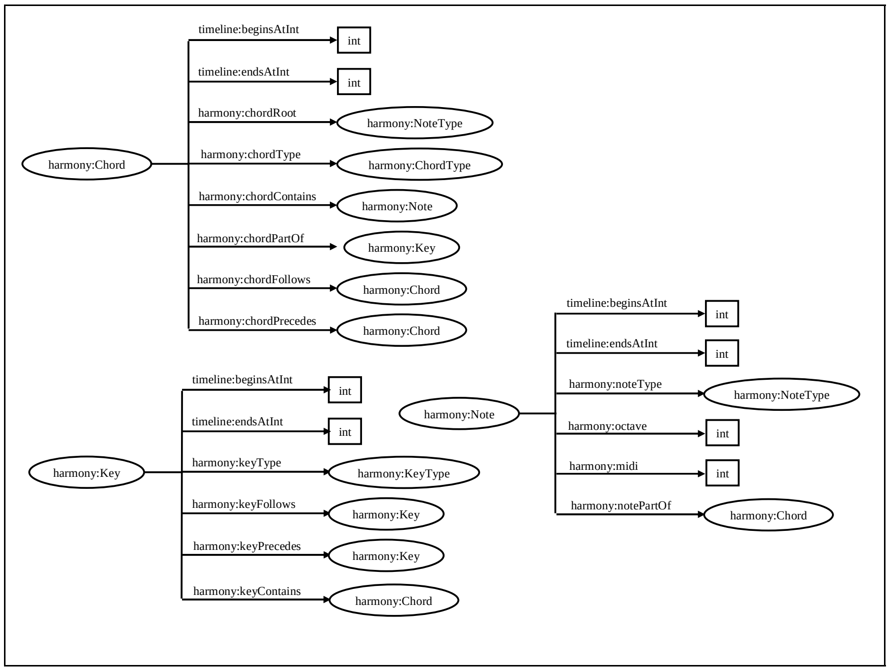


Figure 22: Harmony Note, Chord and Key classes

The Note, Chord and Key classes are subclassed from the Event class cited in section 6.2 allowing them to be tied to a Timeline. Each Event’s start and end times are defined by their *timeline:beginsAtInt* and *timeline:endsAtInt* properties. Also, each of the Note, Chord and Key classes have their own property type as described earlier.

Notes have a *midi* property which specifies the note’s MIDI pitch and an *octave* property in addition to their type. This allows for example, a Note to be defined as A flat in the 5th octave. The Note class also includes a notePartOf property which links the Note to a particular Chord. The complex temporal relationships between Notes are not reflected in this ontology though inferencing based on Allen's approach is a subject for future research.

Chords and Keys have simpler temporal relationships. The ontology assumes that key changes (or modulation) in a way that one key follows another. This is reflected by the *keyFollows* property and its inverse *keyPrecedes*. Similarly, chord changes are reflected by the *chordFollows* and *chordPrecedes* properties. Keys and Chords are related through containment relationships. A Key will have a set of *keyContains* properties to define the set of Chords present in that Key. The inverse property *chordPartOf* links the Chord to its parent Key. Similar relations exist between Chords and Notes. The set of *chordContains* properties identify the constituent Notes in a Chord with the inverse *notePartOf* property of Note identifying its parent Chord.

6.7 Inferencing using the OWL Model

Inferencing (or reasoning) means that we can derive additional facts from instance descriptions and an associated ontology. The Harmony ontology contains relationships that connect objects temporally or through containment. An example of the temporal relationship is the *chordFollows* relationship which links two chords. The relationship is expressed as:

Chord2 *chordFollows* Chord1

There is an inverse relationship within the model called *chordPrecedes*. Therefore the inverse relationship is expressed as:

Chord1 *chordPrecedes* Chord2

In OWL, the two relationships are connected using the *owl:inverseOf* construct as shown in the following snippet of the ontology file.

```
owl:ObjectProperty rdf:about="urn:x-phd:harmony/ChordPrecedes">
    <owl:inverseOf>
      <owl:ObjectProperty rdf:about="urn:x-phd:harmony/ChordFollows"/>
    </owl:inverseOf>
    <rdfs:domain rdf:resource="urn:x-phd:harmony/Chord"/>
    <rdfs:range rdf:resource="urn:x-phd:harmony/Chord"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="urn:x-phd:harmony/ChordFollows">
    <owl:inverseOf rdf:resource="urn:x-phd:harmony/ChordPrecedes"/>
    <rdfs:domain rdf:resource="urn:x-phd:harmony/Chord"/>
    <rdfs:range rdf:resource="urn:x-phd:harmony/Chord"/>
</owl:ObjectProperty>
```

This means that an application only needs to assert one of the relationships by adding it to the model. The inverse relationship is inferred from the ontology and returned in response to a query against the model.

Inferencing using the harmonic model can be illustrated using the following code sample. In this sample, the harmony ontology is used to generate a Jena schema model (1) in addition to the model to be used for the instance data (2). An OWL reasoner is created (3) which is specialised to the schema and applied to the data model to obtain an inference model (4). A URI generator is created and some URIs to identify the key and three chords are produced (5).

In the inference model we create three chords and one key as resources (6) together with two properties to represent the c*hordFollows* and *keyContains* relationships (7). The example then asserts that chord c1 follows chord c0 (8), chord c2 follows chord c1 (9) and key k0 contains chord c0, c1 and c2 (10). The resources representing chord c1 and key k0 are then retrieved from the model (11) and printed using the *printStatements()* method (12). Finally, the model is serialized and output as an RDF file (13).

```
public class ITest3 {
    private final static String rdfDir =
        "C:\\EclipseWorkspace\\RealTimeMusicAnalyser\\RDF\\";
    private final static String ontFName =
        "C:\\EclipseWorkspace\\RealTimeMusicAnalyser\\Harmony.owl";
    private final static String rdfName = rdfDir + "itest3.rdf";

    public static void main(String[] args) {
        // Create the Jena schema model                          (1)
        Model schema = FileManager.get().loadModel(ontFName);

        // Create the Jena model for the data                    (2)
        Model data = ModelFactory.createOntologyModel();

        // Create the OWL reasoner specialised to the schema     (3)
        Reasoner reasoner = ReasonerRegistry.getOWLReasoner();
        reasoner = reasoner.bindSchema(schema);

        // Create the inference model applying the reasoner to the
        // data                                                 (4)
        InfModel infModel = ModelFactory.createInfModel(reasoner,
                                                        data);

        // Create the URI Generator and some URIs               (5)
        new URIGenerator("jenatest");
        String c0Str = URIGenerator.getChordURI().toString();
        String c1Str = URIGenerator.getChordURI().toString();
        String c2Str = URIGenerator.getChordURI().toString();
        String k0Str = URIGenerator.getKeyURI().toString();

        try {
            // Create some chords and the key                   (6)
            Resource c0 = infModel.createResource(c0Str,
                                              HARMONY.Chord);
            Resource c1 = infModel.createResource(c1Str,
                                              HARMONY.Chord);
            Resource c2 = infModel.createResource(c2Str,
                                              HARMONY.Chord);
            Resource k0 = infModel.createResource(k0Str, HARMONY.Key);

            // Create relationship properties                   (7)
            Property cFollows =
              nfModel.createProperty(HARMONY.ChordFollows.toString());
            Property kContains =
              nfModel.createProperty(HARMONY.KeyContains.toString());

            // c1 follows c0 .....                              (8)
            c1.addProperty(cFollows, c0);

            // c2 follows c1 .....                              (9)
            c2.addProperty(cFollows, c1);

            // k0 contains c0, c1, c2                           (10)
            k0.addProperty(kContains, c0);
            k0.addProperty(kContains, c1);
            k0.addProperty(kContains, c2);

            // Now query the inference model ......
            // Get Resource for c2 and k0.....                  (11)
            Resource rC1 = infModel.getResource(c1Str);
            Resource rK0 = infModel.getResource(k0Str);

            // Print statements for c1 and k0                   (12)
            System.out.println("Chord " + c1Str + ":");
            printStatements(infModel, rC1, null, null);
            System.out.println("\nKey " + k0Str + ":");
            printStatements(infModel, rK0, null, null);

            // Output the model as RDF                          (13)
            FileOutputStream fos = new FileOutputStream(rdfName);
            data.write(fos);
            fos.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
    }
}
```

The following example shows the RDF serialization of the model written from line (13). It correctly shows the three chords and one key with the *ChordFollows* relationships expressed for chords 1 and 2 in addition to the *KeyContains* relationship for the key and the contained chords. Note that the model serialization does not show the inverse relationships *ChordPrecedes* and *ChordDuring*.

```
<rdf:RDF
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:j.0="urn:x-phd:harmony/"
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
    <rdf:Description rdf:about="urn:x-phd:harmony/jenatest/Chord/0">
        <rdf:type rdf:resource="urn:x-phd:harmony/Chord"/>
    </rdf:Description>
    <rdf:Description rdf:about="urn:x-phd:harmony/jenatest/Chord/1">
        <j.0:ChordFollows rdf:resource="urn:x-
phd:harmony/jenatest/Chord/0"/>
        <rdf:type rdf:resource="urn:x-phd:harmony/Chord"/>
    </rdf:Description>
    <rdf:Description rdf:about="urn:x-phd:harmony/jenatest/Chord/2">
        <j.0:ChordFollows rdf:resource="urn:x-
phd:harmony/jenatest/Chord/1"/>
        <rdf:type rdf:resource="urn:x-phd:harmony/Chord"/>
    </rdf:Description>
    <rdf:Description rdf:about="urn:x-phd:harmony/jenatest/Key/0">
        <j.0:KeyContains rdf:resource="urn:x-
phd:harmony/jenatest/Chord/2"/>
        <j.0:KeyContains rdf:resource="urn:x-
phd:harmony/jenatest/Chord/1"/>
        <j.0:KeyContains rdf:resource="urn:x-
phd:harmony/jenatest/Chord/0"/>
        <rdf:type rdf:resource="urn:x-phd:harmony/Key"/>
    </rdf:Description>
</rdf:RDF>
```

If we now look at the output for chord 1 and key 0 generated by the *printStatements()* method at (12) we can now see additional inferred relationships for chord 1; that it *precedes* chord 2 and is *during* key 0.

```
Chord urn:x-phd:harmony/jenatest/Chord/1:
 - (urn:x-phd:harmony/jenatest/Chord/1 urn:x-phd:harmony/ChordFollows
urn:x-phd:harmony/jenatest/Chord/0)
 - (urn:x-phd:harmony/jenatest/Chord/1 rdf:type urn:x-
phd:harmony/Chord)
 - (urn:x-phd:harmony/jenatest/Chord/1 rdf:type owl:Thing)
 - (urn:x-phd:harmony/jenatest/Chord/1 urn:x-phd:harmony/ChordPrecedes
urn:x-phd:harmony/jenatest/Chord/2)
 - (urn:x-phd:harmony/jenatest/Chord/1 rdf:type rdfs:Resource)
 - (urn:x-phd:harmony/jenatest/Chord/1 urn:x-phd:harmony/ChordDuring
urn:x-phd:harmony/jenatest/Key/0)
 - (urn:x-phd:harmony/jenatest/Chord/1 owl:sameAs urn:x-
phd:harmony/jenatest/Chord/1)
```

```
Key urn:x-phd:harmony/jenatest/Key/0:
 - (urn:x-phd:harmony/jenatest/Key/0 urn:x-phd:harmony/KeyContains
urn:x-phd:harmony/jenatest/Chord/2)
 - (urn:x-phd:harmony/jenatest/Key/0 urn:x-phd:harmony/KeyContains
urn:x-phd:harmony/jenatest/Chord/1)
 - (urn:x-phd:harmony/jenatest/Key/0 urn:x-phd:harmony/KeyContains
urn:x-phd:harmony/jenatest/Chord/0)
 - (urn:x-phd:harmony/jenatest/Key/0 rdf:type urn:x-phd:harmony/Key)
 - (urn:x-phd:harmony/jenatest/Key/0 rdf:type rdfs:Resource)
 - (urn:x-phd:harmony/jenatest/Key/0 rdf:type owl:Thing)
 - (urn:x-phd:harmony/jenatest/Key/0 owl:sameAs urn:x-
phd:harmony/jenatest/Key/0)
```

Therefore with a sufficiently rich description such as the Harmony ontology it is unnecessary to assert all facts about a music sample's structure. Additional facts can be asserted from the model structure which may also be extended by use of other rules engines. This is currently outside the scope of this thesis.

6.8 Creating the Harmony Model

The SemanticCapture application structure follows the structure described earlier in the thesis for subscribing applications. This is illustrated in Figure 23.

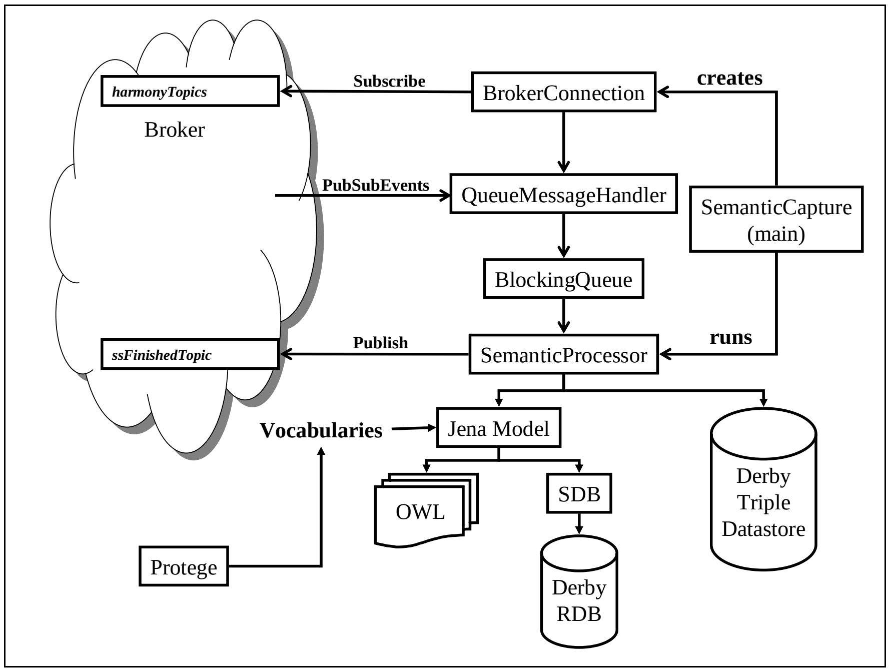


Figure 23: Semantic model capture

The set of *harmonyTopics* subscribed to by the application correspond to a topic for each of the significant events published by the key and chord extraction application described in the previous chapter. In addition, the application subscribes to the *filenameTopic* that identifies the testcase (or other filename) so that the model can be linked to the file or musical piece that created it. The *psFinished* event is used for synchronisation. The set of Harmony topics is shown in the following Java code segment.

```
// Harmony Ontology
String timelineStartTopic     = "Harmony/Ontology/TimelineStart";
String timelineEndTopic       = "Harmony/Ontology/TimelineEnd";
String noteStartTopic         = "Harmony/Ontology/NoteStart";
String noteEndTopic           = "Harmony/Ontology/NoteEnd";
String chordStartTopic        = "Harmony/Ontology/ChordStart";
String chordEndTopic          = "Harmony/Ontology/ChordEnd";
String chordContainsTopic     = "Harmony/Ontology/ChordContains";
String keyStartTopic          = "Harmony/Ontology/KeyStart";
String keyEndTopic            = "Harmony/Ontology/KeyEnd";
String[] harmonyTopics = {
     EventConstants.timelineStartTopic,
     EventConstants.timelineEndTopic,
     EventConstants.noteStartTopic,
     EventConstants.noteEndTopic,
     EventConstants.chordStartTopic,
     EventConstants.chordEndTopic,
     EventConstants.keyStartTopic,
     EventConstants.keyEndTopic,
     EventConstants.filenameTopic,
     EventConstants.closingTopic,
     EventConstants.chordContainsTopic,
     EventConstants.psFinishedTopic};
```

The Harmony ontology was created using the Protege ontology editor. The editor can export a vocabulary definition for the ontology which complies with the Jena ontology and model interfaces. This allows a Jena model to be constructed for each testcase which is then serialised as an OWL file. Jena models can be persisted to a relational database using the semantic database (SDB) component of Jena or serialised from the model as an OWL file.

The serialised model is verbose, so for brevity, the three main entities Key, Chord and Note descriptions are illustrated in the following OWL segments:

```
<rdf:Description rdf:about="harmony:mozartbhqrtetsk08003mKey562">
    <timeline:endsAtInt
     rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
        43015
    </timeline:endsAtInt>
    <harmony:keyMode>major</harmony:keyMode>
    <harmony:keyRoot rdf:resource="harmony:Gn"/>
    <timeline:beginsAtInt
     rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
        0
    </timeline:beginsAtInt>
    <timeline:onTimeLine>
        mozartbhqrtetsk08003mTimeline143
    </timeline:onTimeLine>
    <rdf:type rdf:resource="harmony:Key"/>
</rdf:Description>

<rdf:Description rdf:about="harmony:mozartbhqrtetsk08003mChord42700">
    <timeline:endsAtInt
     rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
        43015
    </timeline:endsAtInt>
    <harmony:chordContains>
        harmony:mozartbhqrtetsk08003mNote134681
    </harmony:chordContains>
    <harmony:chordPartOf>
        harmony:mozartbhqrtetsk08003mKey562
    </harmony:chordPartOf>
    <harmony:chordType rdf:resource="harmony:C0"/>
    <harmony:chordRoot rdf:resource="harmony:Dn"/>
    <timeline:beginsAtInt
     rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
        42840
    </timeline:beginsAtInt>
    <timeline:onTimeLine>
        mozartbhqrtetsk08003mTimeline143
    </timeline:onTimeLine>
    <rdf:type rdf:resource="harmony:Chord"/>
</rdf:Description>

<rdf:Description rdf:about="harmony:mozartbhqrtetsk08003mNote134790">
    <timeline:endsAtInt
     rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
        51664
    </timeline:endsAtInt>
    <harmony:midi rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
        24
    </harmony:midi>
    <harmony:octave
     rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
        2
    </harmony:octave>
    <timeline:beginsAtInt
     rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
        51333
    </timeline:beginsAtInt>
    <timeline:onTimeLine>
        mozartbhqrtetsk08003mTimeline143
    </timeline:onTimeLine>
    <harmony:noteType rdf:resource="harmony:Cn"/>
    <rdf:type rdf:resource="harmony:Note"/>
</rdf:Description>
```

6.9 The Triple Database

An alternative approach is to use a triple database schema implemented within a Derby relational database to persist the model as a set of triples; the triple database.

This approach was to take advantage of existing tools for the visual querying of triples. This will be described further in the next chapter.

Each of the published harmony events have two *insert* methods each with different signatures. The first adds the event to the Jena model whilst the second adds the event as a set of triples to the triple database. Taking the *NoteStartEvent* as an example, the event is added to the Jena model using the following method.

```
public void insert(Model model) {
      Literal l;

      // Create the Note
      model.createResource(noteURIStr, HARMONY.Note);
      model.createResource(noteURIStr).addProperty(HARMONY.noteType,
          NoteTable.getIndividual(noteName));
      model.createResource(noteURIStr).addProperty(TIMELINE.onTimeLine
          , timelineURIStr);
      l = model.createTypedLiteral(new Integer((int) startTime));
      model.createResource(noteURIStr).addProperty(TIMELINE.beginsAtIn
                                                    t, l);
      l = model.createTypedLiteral(new Integer(octave));
      model.createResource(noteURIStr).addProperty(HARMONY.octave, l);
}
```

In this code, the static variables indicated by HARMONY and TIMELINE are created by the vocabulary exported from Protege. The Note is first created with the URI contained within the *NoteStartEvent*. The NoteType individual is looked up in a table given the event *noteName* and added as a Note property. The Timeline is linked via its URI and the Note start time is added using the *beginsAtInt* property. Finally the *octave* property is added. The *beginsAtInt* and *octave* properties are encoded as typed literals via the Jena model API.

In contrast, the following Java method is used to insert the event into the triple database.

```
public void insert(Connection conn, String tcName) {
      insertTriple(conn, reducedStr(noteURIStr), "Note", -1, -1, null,
                   noteName, "string", -1, -1, null, "noteType",
                   tcName);
      insertTriple(conn, reducedStr(noteURIStr), "Note", -1, -1, null,
                   Integer.toString(octave), "int", -1, -1, null,
                   "octave", tcName);
      insertTriple(conn, reducedStr(noteURIStr), "Note", -1, -1, null,
                   Integer.toString((int)startTime), "int", -1, -1
                   null, "timeline:beginsAtInt", tcName);
}
```

The *insertTriple* method assembles a *java.sql.PreparedStatement* from the method arguments and executes it to insert the triple as a row into the database. The *tcName* argument is the testcase name which corresponds to the Jena model. Storing this name in each row of the triple database allows the serialised OWL file to be linked to the triples for subsequent display as part of any subsequent query and retrieval. The relational data model for the triple database is shown in Appendix D and discussed in more detail in the next chapter.

6.10 Summary

In this chapter I have reviewed current semantic web languages and technologies and their use in documenting metadata for music. I have discussed how time can be represented and illustrated this in the context of temporally related objects in music; in particular Notes, Chords and Keys. I have illustrated how OWL descriptions and associated inferencing can derive facts not asserted by applications. Finally I have defined an ontology for describing harmonic structure and shown how this ontology (or model) can be assembled as OWL files and triples in a triple database using the events generated by the applications described in earlier chapters.

Chapter 7 Query by Patterns

7.1 Introduction

The previous chapter described how musical objects, in particular keys, chords and notes, could be represented as a triple in the form of a subject, object and predicate. This representation could be serialised in the form of an OWL description or alternatively as rows within a relational database table. This chapter discusses how these representations may be queried by taking into account the inherent patterns that exist within harmonic structures. It describes how these patterns may be visualised and transformed into SQL and SPARQL queries appropriate to different database technologies.

7.2 Patterns as knowledge

In chapter 2 I described the Data, Information, Knowledge and Wisdom (DIKW) hierarchy and how it has motivated this research. In that chapter, I cited the work of Bellinger et al [Bellinger07] who suggest that understanding is not a separate layer in the hierarchy as proposed by Ackoff but is the process of transformation between layers of the DIKW hierarchy. They suggest that transforming data to information involves an understanding of the relationships that exist in the data. Similarly, knowledge is an understanding of the patterns that exist in information and wisdom is an understanding of the principles that exist in knowledge. It is this hypothesis that patterns that exist in information guide our knowledge that has motivated the database search and retrieval approach described here.

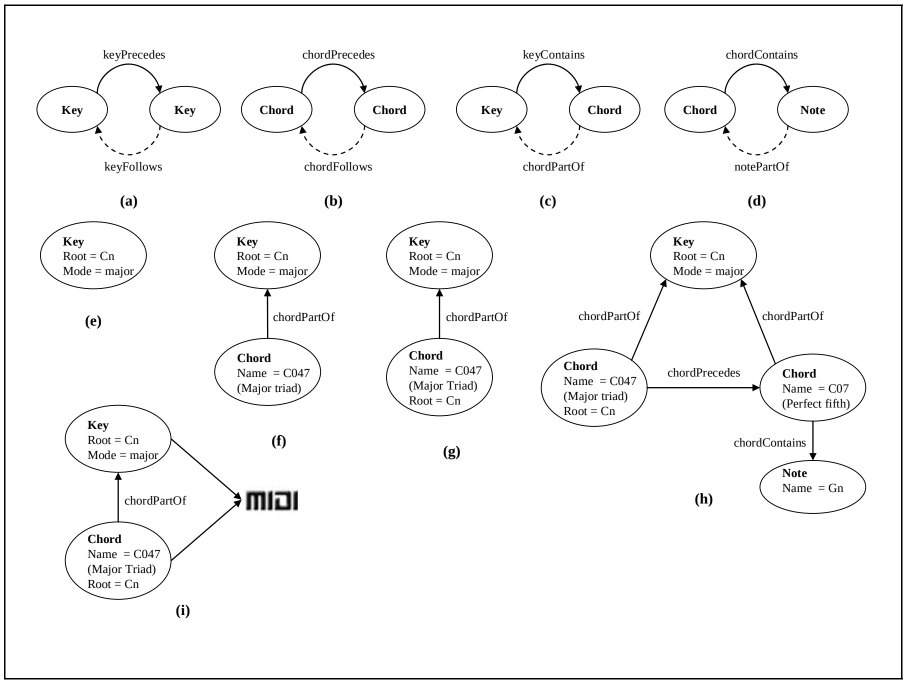

keyPrecedes
Key
Key
keyFollows
(a)
chordPrecedes
Chord
Chord
chordFollows
(b)
keyContains
Key
Chord
chordPartOf
(c)
chordContains
Chord
Note
notePartOf
(d)
Key Root = Cn Mode = major
(e)
Key Root = Cn Mode = major
chordPartOf
Chord Name = C047 (Major triad)
(f)
Key Root = Cn Mode = major
chordPartOf
Chord Name = C047 (Major Triad) Root = Cn
(g)
Key Root = Cn Mode = major
chordPartOf
chordPartOf
Chord Name = C047 (Major triad) Root = Cn
chordPrecedes
Chord Name = C07 (Perfect fifth)
chordContains
Note Name = Gn
(h)
Key Root = Cn Mode = major
chordPartOf
MIDI
Chord Name = C047 (Major Triad) Root = Cn
(i)

Figure 24: Patterns within the Harmony ontology

The subject, object, and predicate of a triple is an inherent pattern with the predicate defining the relationship or association between a subject and an object. This can be thought of as two nodes of a directed graph; one representing the subject and one the object. The edge connecting the two nodes represents the predicate. In the Harmony ontology, two generic forms of pattern between entities can be identified; temporal and containment. Temporal relationships exist between entities of the same type and are identified by the *precedes* and *follows* predicates whose names are augmented depending on whether the relationship is between keys or chords. Therefore the *keyPrecedes* and its inverse *keyFollows* define temporal relationships between Keys and *chordPrecedes* and its inverse *chordFollows* define the relationships between Chords. It is assumed that only a single Key or Chord is present at a given point in time therefore a 1:1 relationship exits between the two entities. A similar pair of relationships have not been defined for Notes since the temporal relationships between notes are more complex. Clarification of these inter-note relationships remains a future research task.

Containment relationships exist between entities of different types. Chords sound within the context of a Key and individual Notes make up sounding Chords. These are defined by the *partOf* and *contains* relationships. Therefore the Chords that sound in a particular Key are identified by the “Key *contains* Chord” relationship and its inverse, the “Chord *partOf* Key” relationship. Once again, these generic relationships are augmented with Chord, Key and Note to give the *chordPartOf* and *notePartOf* relationships together with their inverse *keyContains* and *chordContains*. These relationships are illustrated as (a) to (d) in Figure 24.

Extending this concept further means that we can represent more complex structures within music as directed graphs with the nodes representing entities (Key, Chord and Note) and the graph edges as relationships between them. If we can represent these strucures by means of a graph, then by extension the graph itself can become the representation of a query against a database containing the harmonic structure of testcases represented using the same set of entities and their relationships. This is illustrated in (e) to (i) in Figure 24. Note that for ease of notation, some properties (for example chord and key root, key mode and chord name) are shown as annotations of a graph node. In the harmony ontology, these are also triples with predicates such as *chordRoot*, *chordName* and *keyMode* etc.

The first four examples can be interpreted as follows:

(e) A query for all C major keys returning the URI of all C major keys in the database

(f) A query for all major triads (C047) in the key of C major returning the URIs of all major triads together with the URIs of the C major keys they are contained within.

(g) A query for all major triads in the key of C major with a root of C natural returning the URIs of all major keys in C major with a root of C natural. The URI of the C major key containing the triads would also be returned.

(h) A query for all major triads with a root of C natural followed by a perfect fifth in the key of C major. The perfect fifth must contain a G natural. This would return the URIs of the major triad and its following perfect fifth. The URI of the C major key containing the triad and perfect fifth would also be returned.

The fifth example (i) introduces an additional node to represent MIDI files. This may be interpreted as the same query as example (g) but in addition, identify all MIDI files where this pattern is present. Alternative nodes can be introduced for other media sources such as the OWL file containing the harmonic description where the pattern is present.

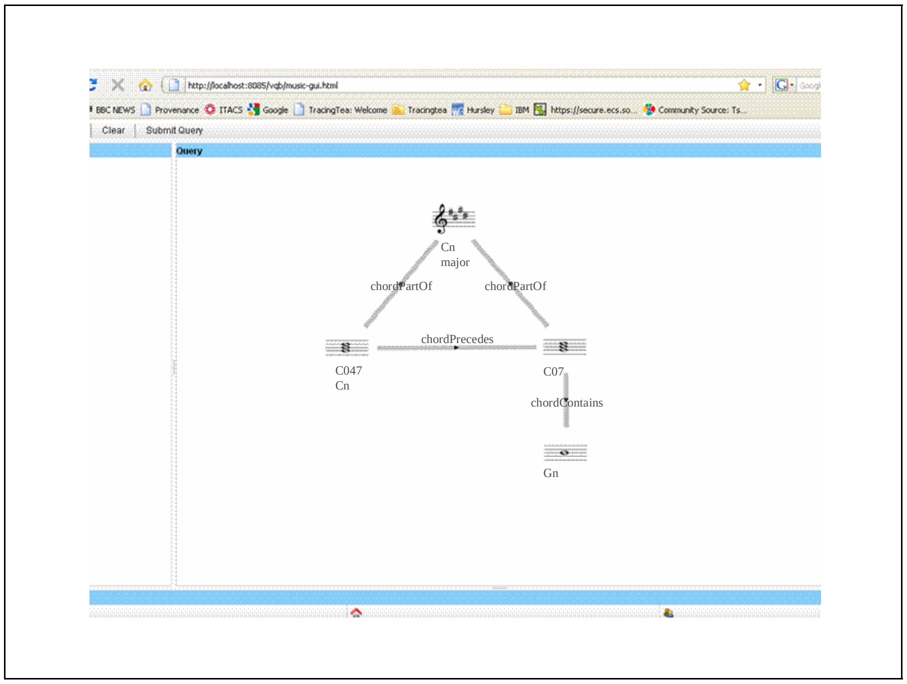


Figure 25: Mock-up of a Visual Query Builder

Figure 25 shows a mock-up of a browser based visual query builder being developed using Web 2.0 techniques that allows a user to draw and annotate a query in the form of a directed graph. The query is then submitted to a triple database or semantic model of the kind described in the previous chapter.

7.3 Encoding the Pattern

The Graph Exchange Language [GXL02] is an XML vocabulary designed to be a standard exchange format for graphs. Structurally, GXL represents a typed, attributed directed graph which can be used to represent schemas in addition to instances of graphs. Its flexibility means that it has wide use in applications that need to exchange data in the form of graphs; particularly between software engineering tools.

The graphs illustrated in Figure 24 (e) to (h) can be represented in GXL with three node types of Key, Chord and Note. Each node type has its own specific set of attributes including a position attribute that denotes its location corrdinates on the drawing canvas. In the initial version of the query builder, the Key node type has attributes for the key root and mode. The Chord and Note nodes types each have a single attribute called name. Graph edges have a type sub-element which documents the relationship between the edge "from" and "to" nodes. Note that a full implementation of the query builder should include all entity properties from the Harmony ontology. These would include entity start and end times which would allow a rich querying environment to navigate the database of harmonic structures. The GXL for the five examples in Figure 24 are shown in Appendix B. Given that GXL is a vocabulary of XML, alternative representations (in particular SQL and SPARQL) can be generated from GXL using appropriate XSL [XSL08] transformations.

7.4 The Query as SQL

The Structured Query Language (SQL) is the language designed for the retrieval and management of data in relational database systems. Using SQL, the subject, object and predicate of a triple can be represented as columns in a database table called *TRIPLES*. The schema for this table is:

```
CREATE TABLE TRIPLES (
      TRANS INTEGER,
      HEAD VARCHAR(100),
      HEAD_TYPE VARCHAR (40),
      BODY VARCHAR (100),
      BODY_TYPE VARCHAR (40),
      SOURCE_DOC INTEGER,
      TIMESTAMP TIMESTAMP,
      ASSOCIATION VARCHAR (200)
);
```

With HEAD containing the subject URL, HEAD_TYPE its type, BODY containing the object, BODY_TYPE its type and ASSOCIATION the predicate. A transaction id is also stored together with a timestamp and a link to a source document which identifies where the triple originated. This could also be the testcase that generated the triple.

For Note and Chord entities, a single row in the table can can be used to identify their types. Therefore the SQL:

```
SELECT * FROM TRIPLES
     WHERE ASSOCIATION = ‘noteType’
     AND BODY = ‘Cn’;
```

would identify all C natural notes. The HEAD column contains the URL of such notes. The SQL:

```
SELECT * FROM TRIPLES
      WHERE ASSOCIATION = ‘chordName’
      AND BODY = ‘C047’;
```

would identify all major triads. Keys however, cannot be uniquely identified from a single row since they are identified by the combination of their root and mode. Therefore an SQL view is created. In SQL, a view consists of a stored query accessible as a virtual table created whenever a select from that table is processed. The Key view is created:

```
CREATE VIEW KEY_VIEW (K_URL, K_ROOT, K_MODE) AS
      SELECT T1.HEAD AS K_URL, T1.BODY AS K_ROOT,
             T2.BODY AS K_MODE
      FROM
            TRIPLES T1,
            TRIPLES T2
      WHERE T1.ASSOCIATION = 'keyRoot'
      AND T2.ASSOCIATION = 'keyMode'
      AND T1.HEAD = T2.HEAD;
```

Queries can now select from the created view which conceptually joins two instances of the TRIPLES based on the HEAD (Key URL) being the same for both root and mode associations. The following query can then be performed:

```
SELECT * FROM KEY_VIEW
      WHERE K_ROOT = ‘Cn’
      AND K_MODE = ‘major’;
```

This would identify all keys in C major.

Views can also be used to model the subject, object, predicate triple in SQL where the subject and object are both entities; in particular the forward and backward associations between entities illustrated in Figure 24 (a) to (d). By modeling these relationships as views, we can combine the views in order to search for the more complex patterns illustrated in Figure 24 (e) to (i). In this case, the views model the edges of the graph between two nodes. A set of these views are joined in order to query the complete graph.

Consider the temporal relationships between two keys. In the forward direction key K1 precedes key K2 and for the inverse K2 follows K1 (assuming the start of K2 occurs after the end of K1). These relationships can be modeled by the two VIEWS K2K_FORWARD_VIEW and K2K_BACKWARD_VIEW.

```
-- K2 follows K1
CREATE VIEW K2K_FORWARD_VIEW (K1_URL, K1_ROOT, K1_MODE, K2_URL,
                              K2_ROOT, K2_MODE) AS
     SELECT K1.K_URL AS K1_URL, K1.K_ROOT AS K1_ROOT,
            K1.K_MODE AS K1_MODE,
            K2.K_URL AS K2_URL, K2.K_ROOT AS K2_ROOT,
            K2.K_MODE AS K2_MODE
     FROM
            TRIPLES T1,
            KEY_VIEW K1,
            KEY_VIEW K2
     WHERE T1.ASSOCIATION = 'keyPrecedes'
            AND T1.HEAD = K1.K_URL
            AND T1.BODY = K2.K_URL;

-- K2 precedes K1
CREATE VIEW K2K_BACKWARD_VIEW (K1_URL, K1_ROOT, K1_MODE, K2_URL,
                               K2_ROOT, K2_MODE) AS
     SELECT K1.K_URL AS K1_URL, K1.K_ROOT AS K1_ROOT,
            K1.K_MODE AS K1_MODE,
            K2.K_URL AS K2_URL, K2.K_ROOT AS K2_ROOT,
            K2.K_MODE AS K2_MODE
     FROM
            TRIPLES T1,
            KEY_VIEW K1,
            KEY_VIEW K2
     WHERE T1.ASSOCIATION = 'keyFollows'
     AND T1.HEAD = K1.K_URL
     AND T1.BODY = K2.K_URL;
```

Both views provide a virtual table with the URL, root and mode of both Key entities. Similar views can be generated for other pairings of entities such as Chords and Notes using CHORD_VIEW which provides a simpler view of a Chord,.

```
-- Chord to Note
CREATE VIEW C2N_VIEW (C_URL, C_NAME, C_ROOT, N_URL, N_TYPE) AS
     SELECT C1.C_URL AS C_URL, C1.C_NAME AS C_NAME,
            C1.C_ROOT AS C_ROOT,
            T2.HEAD AS N_URL, T2.BODY AS N_TYPE
     FROM
            TRIPLES T1,
            TRIPLES T2,
            CHORD_VIEW C1
     WHERE T1.ASSOCIATION = 'chordContains'
     AND T1.HEAD = C1.C_URL
     AND T2.ASSOCIATION = 'noteType'
     AND T2.HEAD = T1.BODY;

-- Note to Chord
CREATE VIEW N2C_VIEW (N_URL, N_TYPE, C_URL, C_NAME, C_ROOT) AS
```

```
SELECT C1.C_URL AS C_URL, C1.C_NAME AS C_NAME,
       C1.C_ROOT AS C_ROOT,
       T2.HEAD AS N_URL, T2.BODY AS N_TYPE
FROM
       TRIPLES T1,
       TRIPLES T2,
       CHORD_VIEW C1
WHERE T1.ASSOCIATION = 'notePartOf'
AND T1.BODY = C1.C_URL
AND T2.ASSOCIATION = 'noteType'
AND T2.HEAD = T1.HEAD;
```

A complete set of views is provided in the Harmony relational model in Appendix D. The efficient use of SQL views depends upon the sophistication of the SQL parser optimiser provided with the database software. Logically, the KEY_VIEW is performing a Cartesian product of joins between two instances of the TRIPLES table. In cases where the number of rows in the table is large, this can take a very long time. Optimisers implement algorithms that operate on table indexes which radically improve performance but their efficiency (such as the number of joins they can support) varies between different database implementations. Apache Derby as an open source relational database implementation does not have as efficient optimisation as for example, a commercial relational database such as IBM’s DB2. Therefore there may need to be an alternative approach to overcome inefficient optimisations of SQL views and joins.

One approach is the use of temporary tables instead of views. Creating a temporary table for each edge in the graph may rely on some of the simpler views but it obviates the need for complex joins and the associated reduction in query performance. This approach is illustrated in the early example of GXL to SQL transformation shown in Appendix C.

7.5 The Query as SPARQL

SPARQL is emerging as the equivalent to SQL for semantic databases where the data is expressed as RDF graphs whose schema is defined by an OWL ontology. It queries the data within the semantic model but does not in itself perform any inferencing or trigger any rules associated with the model. Rule interpretation and inferencing are roles of the underlying database or triple store which are reflected in the result set returned from the query.

The previous section discussed the querying of a triple database where the harmonic model is stored as a set of triples within a relational database. A visual paradigm for describing the query was presented in which the query graph was represented by a model serialised as GXL. The representation was transformed to SQL in order to query the triple datastore using an XSL transformation. In addition to the triple datastore, figure 23 in chapter 6 also shows the harmonic structure optionally being captured as a semantic model using the Jena semantic framework. Once in this model form, the structure can be queried using SPARQL generated by an alternative XSL transform which converts the GXL representation to SPARQL.

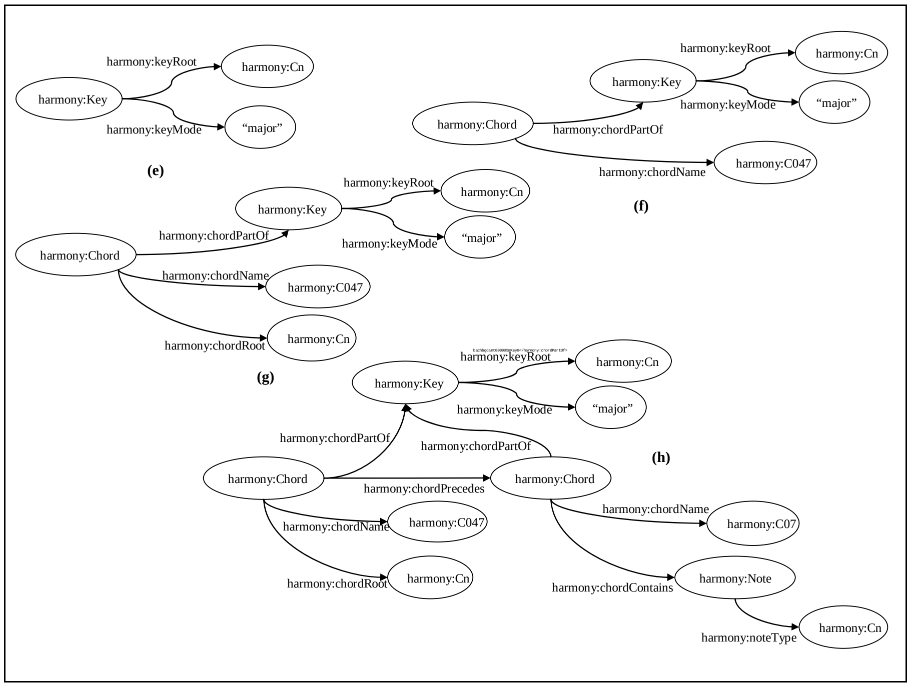


Figure 26: Examples from Figure 24 as complete graphs

If the examples (e) to (h) in Figure 24 are fully expanded as RDF graphs, their structures can be used to generate equivalent query patterns in SPARQL. These expanded graphs are shown in Figure 25. Note that in this expansion, Note and Chords are typed as objects such as harmony:Cn and harmony:C047 which reflects the Protege defined individuals in the Harmony ontology. There exists in the ontology an individual for each Note type (*NoteType*) and Chord type (*ChordType*). A *NoteType* defines the Note's natural (A..G), its modifier (n, s, f, ss, ff), pitchClass and tonalPitchClass; there are 35 *NoteType* Individuals. The 352 *ChordType* individuals include information such as the forteNumber, pitchClassSet and IntervalVector together with a name and description. These type properties have not been used in the example queries shown in this section but could be used in a richer retrieval application which traverses the stored harmonic structure.

The SPARQL for example (e) retrieves the URLs for instances of the Key of C major:

```
PREFIX harmony: <http://localhost:8080/ontology/Harmony/Harmony.owl#>
SELECT ?kurl
WHERE {
      ?kurl harmony:keyRoot harmony:Cn.
      ?kurl harmony:keyMode "major".
}
```

The SPARQL for example (f) retrieves the URLs for the Keys of C major and the URLs of Chords that are part of the Key and a major triad (type C047):

```
PREFIX harmony: <http://localhost:8080/ontology/Harmony/Harmony.owl#>
SELECT ?kurl ?curl
WHERE {
      ?kurl harmony:keyRoot harmony:Cn.
      ?kurl harmony:keyMode "major".
      ?curl harmony:chordPartOf ?kurl.
      ?curl harmony:chordType harmony:C047.
}
```

The SPARQL for example (g) is similar to that for example (f), but the Chord also has a root of C natural:

```
PREFIX harmony: <http://localhost:8080/ontology/Harmony/Harmony.owl#>
SELECT ?kurl ?curl
WHERE {
      ?kurl harmony:keyRoot harmony:Cn.
      ?kurl harmony:keyMode "major".
      ?curl harmony:chordPartOf ?kurl.
      ?curl harmony:chordType harmony:C047.
      ?curl harmony:chordRoot harmony:Cn.
}
```

Finally, the SPARQL for example (h) retrieves the URLs for the Key, the two Chords and the Note contained within the second Chord:

```
PREFIX harmony: <http://localhost:8080/ontology/Harmony/Harmony.owl#>
SELECT ?kurl ?c1url ?c2url ?nurl
WHERE {
      ?kurl harmony:keyRoot harmony:Cn.
      ?kurl harmony:keyMode "major".
      ?c1url harmony:chordPartOf ?kurl.
      ?c1url harmony:chordType harmony:C047.
      ?c1url harmony:chordRoot harmony:Cn.
      ?c1url harmony:chordPrecedes ?c2url.
      ?c2url harmony:chordPartOf ?kurl.
      ?c2url harmony:chordType harmony:C07.
      ?c2url harmony:chordContains ?nurl.
      ?nurl harmony:noteType harmony:Cn.
}
```

7.6 Summary

In this chapter I have discussed the retrieval of harmonic patterns from a stored representation of the music harmonic structure in line with the work of Bellinger et al who view knowledge as patterns of information. I have identified directed graphs as a structural model for querying patterns held within either a relational or semantic database. A visual query interface may be constructed which uses a graphical representation of the query to access either types of database using SQL or SPARQL. The query is generated from a serialisation in GXL of the directed graph using appropriate XSL transformations.

Chapter 8 Conclusions

This thesis has investigated the use of publish/subscribe messaging and how it can provide an interconnection framework for distributed, independent applications that collaborate to create a description of musical harmonic structure. Such a framework may take a data stream of MIDI commands and generate information by the naming of notes and the extraction of key and chord structure. By use of semantic web techniques, this information may be stored within databases and retrieved using pattern based mechanisms which may be mapped into the query syntax appropriate for the database. These processes mirror the Data, Information, Knowledge and Wisdom (DIKW) hierarchy proposed by Ackoff and elaborated by Bellinger and others.

It has demonstrated how this framework can operate by presenting a novel event based implementation of Chew’s spiral array note naming algorithm. The results of this implementation have been favourably compared with other published implementations. A key and chord extraction application has been described which uses metrical analysis to provide sampling points within the musical stream where segments for key finding can be identified and notes can be grouped into chords using a technique based on Forte’s pitch class set theory.

An ontology for musical harmonic structure has been defined which uses the standardised Web Ontology Language OWL. In this ontology, musical note, chord and key objects together with their relationships are described. Instances of the ontology have been created from messaging events generated by the note naming and extraction applications and stored in either a semantic or relational database using appropriate schemas.

Finally, a user model of database interaction based on visual graph patterns has been presented. Given the graph serialisation, queries in SQL and SPARQL may be generated to access the harmonic structure model stored in different database technologies.

8.1 Discussion

The publish/subscribe messaging model allows applications to be distributed between processors within a network. This provides benefits in that analysis and other applications are not as resource constrained as if they were all running within a single processor. The downside is that the applications are distributed with inherent network latency (or delay) as events are distributed between publishers, the broker and subscribers. Latency within a network may be thought of as having two parts: propagation and application latency. Propagation latency is the latency in moving network packets from one node to another and is a function of the capacity of the network link. Application latency is the latency induced by general network traffic and is subject to the type of applications connected to the network and the quantity of traffic they generate; this is more random. MIDI connections between devices are via a serial connection which preserves the real time nature of the interface with predictable and acceptable latency. For the analysis framework to perform with minimal degradation due to latency, the applications should be deployed on a dedicated sub-net to minimise both propagation and application latency. The investigation of latency and the requirements for the real time analysis framework is however, outside the scope of this thesis.

Recently, there have been more music structure descriptions published by groups active in computer musicological research. In addition to MusicXML [MXMLDef], ontologies have been published for symbolic music [SMusic], key [KOnt] and chord [ChOnt] ontologies. Whilst not providing a complete set of descriptive formats for harmonic structure, these are starting to address some aspects of semantic musical description. The Harmony ontology presented in this thesis is intended to illustrate how fundamental musical objects (Key, Chord and Note) are inter-related and provides a mechanism that illustrates how a high level semantic model can be created from lower level algorithmic applications. Further work is required to take the contribution of the Harmony ontology and assess how it complements other published musical descriptions. It may be that the Harmony ontology is subsumed by this published work. In that case, the Harmony events published by the analysis applications presented in this thesis may be re-structured to support the creation of descriptions in one or more of the published formats.

The chord identification application presented in chapter 5 of this thesis identifies chords from notes that are co-sounding at the same beat. There are, however, many different combinations of notes that constitute the same chord as illustrated in the following figure.

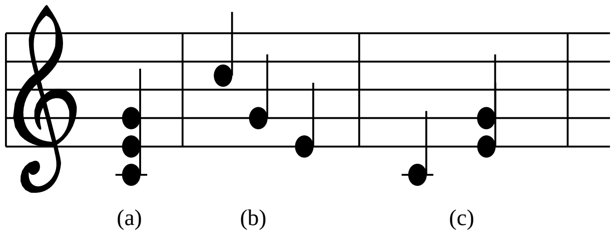


Figure 27: Alternative representations of the same chord

For example, a C major triad consists of three notes (C, E and G) sounding simultaneously (a), separately (b) or as a single note followed by a minor third (c). The chord identification described in this thesis would correctly identify (a) as a C major triad but would identify (b) as three unison notes and (c) as a unison followed by a minor third. A further layer of semantic inferencing is needed to correctly identify all three possibilities as forms of the same chord. The definition of temporal rules needed to infer higher level chord structures from sequences of notes and chords has not been addressed in this thesis. This problem of inferring higher level chord structured from sequences within music maps to a more generic problem of applying temporal rules to streams of events and is an ongoing research activity within the semantic web research community.

In music, a suspension is one or more notes temporarily held before the harmony is resolved to a particular chord. Suspensions are important indicators of particular musical styles. Correct chord extraction will identify suspensions and ignore them before identifying the resolved chord type. The chord extraction algorithm developed in this thesis does not support the identification of suspensions.

In addition to the inference of higher level chord structures, the techniques described in this thesis do not address the ongoing research topic of genre classification. A harmonic structural description alone will not differentiate between musical styles. Other characteristics of the audio signal are better discriminators. For example, much 1972s progressive rock was heavily influenced by classical music and used many of its motifs and harmonic structures. Analysis of its harmonic structure would not clearly differentiate it from classical forms; however, the instruments used in the performance would clearly identify the genre.

8.2 Future Work

Whilst this thesis has concentrated on a particular set of contributions, the work has identified some interesting directions for future research.

8.2.1 Coordination with Music Ontology Research

Raimond’s Music Ontology (MO) is gaining widespread acceptance within the Music Information Retrieval (MIR) research community. The Harmony ontology developed as part of this thesis includes some of Raimond’s work (the Timeline and Event ontologies) but has not been designed as an ontology derived from the broader MO. Further work should investigate how linkages can be made between the work of this thesis and the MO and other published musical structure descriptions..

8.2.2 Developing Temporal Rules of Music

The chord extraction technique presented in this thesis can identify a chord from the notes that are concurrently sounding at the sampling point. This is a restricted view of chord identification since separately sounding notes over a period of time also contribute to chords which may last for several measures in the music. Inferencing rules within OWL are limited, but the development of temporal rules based on Allen's work and musicological theory is feasible. A temporal rule inferencing engine, possibly based on Prolog, would allow the current ontology to be enriched further by identifying these longer lasting chord structures.

8.2.3 Developing an engineered Framework

The framework described in this thesis has been demonstrated by a small number of separate applications interconnected by a publish/subscribe messaging infrastructure. For an engineered implementation of the framework, a full design and test process needs to be completed. This implementation would include designed programming interfaces to allow new applications to be developed, management processes to allow components to be distributed throughout a network, and the addition of further topics and events to the framework.

8.2.4 Developing a complete Visual Query Application

The visual query paradigm presented in this thesis has not been hardened into a complete browser based application. Web 2.0 techniques allow a complex browser/server based application to be developed which involves query drawing, submission and results management to be created.

For such an application, a full design process should be adopted which identifies use cases from the computer musicology and information retrieval research community to motivate the implementation of the application.

List of References

[Ackoff89] Ackoff, R. L. (1989). "From Data to Wisdom." Journal of Applied Systems Analysis **16**: 3 - 9.

[Allen83] Allen, J. F. (1983). "Maintaining knowledge about temporal intervals." Communications of the ACM **26**(11): 832 - 843.

[Altherr99] Altherr, M., M. Erzberger, et al. (1999). iBus - A software bus middleware for the Java platform. Proceedings of the International Workshop on Reliable Middleware Systems.

[Antoniou08] Antoniou, G. and F. van Harmelen (2008). A Semantic Web Primer. Cambridge, Massachusetts, The MIT Press.

[Awad04] Awad, E. M. and H. M. Ghaziri (2004). Knowledge Management. Upper Saddle City, NJ, Pearson Educational International.

[Banavar99] Banavar, G., T. Chandrra, et al. (1999). An efficient multicast protocol for content-based publish-subscribe systems. Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99).

[Bellinger07] Bellinger, G., D. Castro, et al. "Data, Information, Knowledge and Wisdom."   Retrieved 26/07/2007, from http://www.systems-thinking.org/dikw/dikw.htm.

[Bergamaschi07] Bergamaschi, F., D. Conway-Jones, et al. (2007). A Distributed Test Framework for the Validation of Experimental Algorithms Using Real and Simulated Sensors. Annual Conference of ITA, University of Maryland, USA.

[Berners-Lee01] Berners-Lee, T., J. Hendler, et al. (2001). "The Semantic Web." Scientific American **284**: 34 - 43.

[Birrell83] Birrell, A. D. and B. J. Nelson (1983). Implementing remote procedure calls. Proceedings of the ACM Symposium on Operating System Principles, Bretton Woods, NH, ACM Press, New York.

[Brinner95] Brinner, B. (1995). Knowing music, making music: Javanese gamelan and the theory of musical competence and interaction, The University of Chicago Press.

[Chafe82] Chafe, C., B. Mont-Reynaud, et al. (1982). "Toward an intelligent editor of digital audio: Recognition of musical constructs." Computer Music Journal **6**(1): 30 - 41.

[Chew00] Chew, E. (2000). Towards a Mathematical Model of Tonality. Operations Research Center, MIT. Cambridge, MA, MIT. **PhD**.

[Chew04] Chew, E. and Y.-C. Chen (2004). "Real Time Pitch Spelling Using the Spiral Array." Computer Music Journal **20**(2).

[ChOnt]. "The Chord Ontology." from http://www.omras2.org/ChordOntology.

[Cohn97] Cohn, R. (1997). "Neo-riemann operations, parsimonious trichords, and their tonnetz representation." Journal of Music Theory **41**(1): 1 - 66.

[Cooper97] Cooper, D., K.-C. Ng, et al. (1997). MIDI extensions for musical notation (2): Expressive MIDI. Beyond MIDI: The Handbook of Musical Codes. E. Selfridge-Field, MIT Press**:** 80-98.

[DEC94] DEC (1994). DECMessageQ: Introduction to Message Queuing, DEC; Hewlett Packard, Palo Alto, CA.

[Derby]. "Apache Derby." from http://db.apache.org/derby/.

[Desain92] Desain, P. and H. Honing (1992). Music, Mind, and Machine: Studies in Computer Music, Music Cognition, and Artificial Intelligence (Kennistechnologie), Thesis Pub.

[Dowling86] Dowling, W. J. and D. L. Harwood (1986). Music Cognition.

[FaCT] "FaCT (Fast Classification of Terminologies)."

[FOAF05] Brickley, D. and L. Miller. (2005). "FOAF vocabulary specification." 2008, from http://xmlns.com/foaf/0.1.

[Forte73] Forte, A. (1973). The Structure of Atonal Music, Yale University Press.

[Gelernter85] Gelernter, D. (1985). "Generative communication in Linda." ACM Transactions in Programming Language Systems **7**: 80 - 112.

[Good01] Good, M. (2001). "MusicXML: An Internet-Friendly Format for Sheet Music." Proceedings of XML 2001.

[Good02] Good, M. (2002). MusicXML in Practice: Issues in Translation and Analysis. MAX 2002 Conference on Musical Applications of XML, Milan.

[GXL02]. "Graph Exchange Language (GXL)." from http://www.gupro.de/GXL/.

[Hapner02] Hapner, M., R. Burridge, et al. (2002). Java Messaging Service, Sun Microsystems Inc, Santa Clara CA.

[Hargreaves86] Hargreaves, D. J. (1986). Developmental Psychology of Music, Cambridge University Press.

[Hewlett97] Hewlett, W. B. (1997). MuseData: multipurpose representation. Beyond MIDI: The Handbook of Musical Codes. E. Selfridge-Field, MIT Press**:** 402 - 447.

[Hewlett97b] Hewlett, W. B. (1997). MIDI extensions for musical notation (3): MIDIPlus. Beyond MIDI: The Handbook of Musical Codes. E. Selfridge-Field, MIT Press**:** 99-104.

[Hewlett97c] Hewlett, W. B. (1997). MuseData: multipurpose representation. Beyond MIDI: The Handbook of Musical Codes. E. Selfridge-Field, MIT Press**:** 402-447.

[Holtzmann77] Holtzmann, S. R. (1977). "A program for key determination." Interface **6**: 29 - 56.

[Horstmann97] Horstmann, M. and M. Kirtland. (1997). "DCOM Architecture." from http://www.microsoft.com/com/tech/DCOM.asp.

[Howard97] Howard, J. (1997). Plaine and Easie Code: a code for music bibliography. Beyond MIDI: The Handbook of Musical Codes. E. Selfridge-Field, MIT Press**:** 362-372.

[Huang01] Huang, Y. and H. Garcia-Molina (2001). Publish/subscribe in a mobile environment. Proceedings of MobiDE.

[Huron97] Huron, D. (1997). Humdrum and Kern: selective feature coding. Beyond MIDI: The Handbook of Musical Codes. E. Selfridge-Field, MIT Press**:** 375-401.

[IBM95] IBM (1995). MQSeries: An introduction to messaging and queuing, Technical Report GC33-0805-01, IBM Corporation, Yorktown Heights, NY.

[Jena]. "Jena – A Semantic Web Framework for Java." from http://jena.sourceforge.net/.

[JMS02] JMS. (2002). "Java Message Service Specifcation version 1.1." from http://java.sun.com/products/jms/docs.html.

[KOnt]. "The Key Ontology." from http://motools.sourceforge.net/keys/keys.owl.

[Kostka95] Kostka and D. Payne (1995). Tonal Harmony. New York, McGraw-Hill.

[Krumhansl04] Krumhansl, C. L. (2004). "The cognition of tonality as we know it today." Journal of New Music Research **33**(3): 253 - 268.

[Krumhansl78] Krumhansl, C. L. (1978). The Psychological Representation of Musical Pitch in a Tonal Context. Stanford University. Stanford, CA, Stanford University. **PhD**.

[Krumhansl82] Krumhansl, C. L. and E. F. Kessler (1982). "Tracing the dynamic changes iin perceived tonal organization in a spatial representation of musical keys." Psychological Review **89**: 334 - 68.

[Krumhansl90] Krumhansl, C. L. (1990). Cognitive Foundations of Musical Pitch. New York, OUP.

[Laden89] Laden, B. and D. H. Keefe (1989). "The Representation of Pitch in a Neural Net Model of Chord Classification." Computer Music Journal **13**(4): 12 - 26.

[Large94] Large, E. W. and J. F. Kolen (1994). "Resonance and the perception of musical meter." Connection Science **6**: 177 - 208.

[Lee91] Lee, C. (1991). The perception of musical structure:Experimental evidence and a model. Representing Musical Structure. P. Howell, R. West and I. Cross. London, Academic Press**:** 59 - 127.

[Lehman99] Lehman, T., S. M. Laughry, et al. (1999). TSpaces: The next wave. Proceedings of the Hawaii International Conference on System Sciences.

[Lerdahl83] Lerdahl, F. and R. Jackendoff (1983). A Generative Theory of Tonal Music, MIT Press.

[Li89] Li, K. and P. Hudak (1989). "Memory coherence in shared memory systems." ACM Transactions of Computing Systems **7**(4): 321 - 359.

[Longuet-Higgins71] Longuet-Higgins, H. C. and M. J. Steedman (1971). "On interpreting Bach." Machine Intelligence **6**: 221 - 241.

[MacMillan02] MacMillan, K. (2002). Common Music Notation as a Source for Music Information Retrieval. Workshop on the Creation of Standardized Test Collections, Tasks, and Metrics for Music Information Retrieval (MIR) and Music Digital Library (MDL) Evaluation, Portland, Oregon.

[Mathews97a] Mathews, M. V. (1997). MIDI extensions for sound control: Augmented MIDI. Beyond MIDI: The Handbook of Musical Codes. E. Selfridge-Field, MIT Press**:** 105-108.

[Maxwell92] Maxwell, J. H. (1992). An Expert System for Harmonizing Analysis of Tonal Music. Understanding Music with AI: Perspectives on Music Cognition. M. Balaban, K. Ebcioglu and O. Laske. Cambridge, Massachusetts, MIT Press.

[Meredith07] Meredith, D. (2007). Computing Pitch Names in Tonal Music: A Comparative Analysis of Pitch Spelling Algorithms (Early Draft). Faculty of Music. Oxford, University of Oxford. **DPhil**.

[MMA01] MMA. (2001). "The Complete MIDI 1.0 Detailed Specification." from http://www.midi.org/about-midi/specinfo.shtml.

[MXMLDef]. "MusicXML Definition." from http://www.musicxml.org/xml.html.

[Nelson04] Nelson, P. (2004). "Pitch Class Sets." Retrieved 2008, from http://composertools.com/Theory/PCSets/.

[Nordli97] Nordli, K. E. (1997). MIDI extensions for musical notation (1): NoTAMIDI meta-events. Beyond MIDI: The Handbook of Musical Codes. E. Selfridge-Field, MIT Press: 73 - 79.

[O'Connell07] O'Connell, B., A. Stanford-Clark, et al. (2007). "Using the IBM Lotus Expeditor micro broker MQTT client to publish messages." from http://www.ibm.com/developerworks/lotus/library/expeditor-mqtt/index.html.

[OMG02] OMG (2002). The Common Object Request Broker: Core Specification, Object Management Group, Needham, MA.

[Oracle02] Oracle (2002). Oracle9i Application Developer's Guide - Advanced Queuing, Oracle, Redwood Shores, CA.

[OWL04]. "Web Ontology Language (OWL)." from http://www.w3.org/2004/OWL/.

[Pardo00] Pardo, B. and W. Birmingham (2000). Automated Partitioning of Tonal Music. Proceedings of the 13th International FLAIRS Conference,, Orlando, Florida.

[Pardo02] Pardo, B. and W. Birmingham (2002). "Algorithms for Chordal Analysis." Computer Music Journal, **26**(2): 27 - 49.

[Parncutt94] Parncutt, R. (1994). "A perceptual model of pulse salience and metrical accent in musical rhythms." Music Perception **11**: 409 - 464.

[Pellet]. "Pellet: The Open Source OWL DL Reasoner." from http://clarkparsia.com/pellet.

[Povel85] Povel, D.-J. and P. Essens (1985). "Perception of temporal patterns." Music Perception **2**: 411 - 440.

[Protege]. "Protege Ontology Editor." from http://protege.stanford.edu/.

[Racer]. "Racer (Renamed Abox and Concept Expression Reasoner)." from http://www.sts.tu-harburg.de/~r.f.moeller/racer/.

[Raimond07c] Raimond, Y. and S. Abdallah. (2007). "The Timeline Ontology." Retrieved 17th November, 2008, from http://motools.sourceforge.net/timeline/timeline.html.

[Raimond07d] Raimond, Y. and S. Abdallah. (2007). "The Event Ontology." Retrieved 17th November, 2008, from http://motools.sourceforge.net/event/event.html.

[Raimond2006] Raimond, Y., S. Abdallah, et al. (2006). An ontology-based approach to information management for music analysis systems. 120th AES Convention.

[Raimond2007a] Raimond, Y., S. Abdallah, et al. (2007). The Music Ontology. Proc. 8th International Conference on Music Information Retrieval, ISMIR 2007.

[Raimond2007b] Raimond, Y., C. Sutton, et al. (2007). A distributed data-space for music related information. WMS ’07, ACM Multimedia, International Workshop on Multimedia Semantics, Augsberg, Germany.

[Raphael03] Raphael, C. and J. Stoddard (2003). Harmonic analysis with probabilistic graphical models. ISMIR 2003.

[RDF04]. "Resource Description Framework (RDF)." from http://www.w3.org/RDF/.

[Robinson06] Robinson, J. M., J. G. Frey, et al. (2006). "The Combechem MQTT LEGO Microscope." from http://www.allhands.org.uk/2006/proceedings/papers/670.pdf.

[Rosenblum97] Rosenblum, D. and A. Wolf (1997). A design framework for Internet-scale event observation and notification. Proceedings of the 6th European Software Engineering Conference / ACM SIG-SOFT 5th Symposium on the Foundations of Software Engineering, ACM Press, New York, NY.

[Rosenthal92] Rosenthal, D. (1992). "Emulation of human rhythm perception." Computer Music Journal **16**(1): 64 - 76.

[Rowley07] Rowley, J. (2007). "The wisdom hierarchy: representations of the DIKW hierarchy." Journal of Information Science **33**(2): 163-180.

[Schaffrath97] Schaffrath, H. (1997). The Essen Associative Code: a code for folksong analysis. Beyond MIDI: The Handbook of Musical Codes. E. Selfridge-Field, MIT Press**:** 343-361.

[Shepherd82] Shepherd, R. N. (1982). Structural representation of musical pitch. The Psychology of Music. D. D, Swets and Zeitlinger.

[Sloboda85] Sloboda, J. A. (1985). The Musical Mind: The Cognitive Psychology of Music, Oxford University Press.

[Smaill93] Smaill, A., G. Wiggins, et al. (1993). "Hierarchical music representation for composition and analysis." Computers and the Humanties **93**: 7 - 17.

[SMusic]. "Symbolic Music Ontology." from http://purl.org/ontology/symbolic-music/.

[Smythe03] Smythe, T. (2003). "Player Piano Rebirth." 2007, from http://members.shaw.ca/smythe/rebirth.htm.

[Solomon]. "Table of Pitch Class Sets." from http://solomonsmusic.net/pcsets.htm
[Solomon82] Solomon, L. (1982). "The List of Chords, Their Properties and Use in Analysis." Interface **11**: 61-107.

[SPARQL07]. "The SPARQL Query Language for RDF." from http://www.w3.org/2001/sw/DataAccess/.

[Sun00] Sun (2000). Java Remote Method Invocation Specification, Sun Microsystems, Santa Clara, CA

[Sun02] Sun (2002). JavaSpaces Service Specification, Sun Microsystems, Santa Clara CA.

[Swartz02] Swartz, A. (2002). "Musicbrainz: A semantic web service." IEEE Intelligent Systems **17**(1): 76 - 77.

[SWRL04]. "The Semantic Web Rule Language." from http://www.w3.org/Submission/SWRL/.

[Temperley01] Temperley, D. (2001). The Cognition of Basic Musical Structures, MIT Press.

[Temperley04] Temperley, D. (2004). "An Evaluation System for Metrical Models." Computer Music Journal **28**(3): 28 - 44.

[Temperley07] Temperley, D. (2007). Music and Probability, MIT Press.

[Temperley99] Temperley, D. and D. Sleator (1999). "Modeling Meter and Harmony: A Peference Rule Approach." Computer Music Journal **23**(1): 19 - 27.

[TIBCO99] TIBCO (1999). TIB/Rendezvous, TIBCO, Palo Alto, CA.

[Tsui02] Tsui, W. S. V. (2002). Harmonic Analysis Using Neural Networks. Graduate Department of Electrical and Computer Engineering. Toronto, University of Toronto. **Master of Applied Science**.

[Turing50] Turing, A. (1950). "Computing Machinery and Intelligence." MIND: A Quarterly Review of Psychology and Philosophy **LIX**(236): 433 - 460.

[Vos96] Vos, P. G. and E. W. Van Geenen (1996). "A parallel-processing key-finding model." Music Perception **14**: 185 - 224.

[Winograd68] Winograd, T. (1968). "Linguistics and the computer analysis of tonal harmony." Journal of Music Theory **12**(1).

[XSL08]. "The Extensible Stylesheet Language (XSL)." from http://www.w3.org/Style/XSL/.

Appendix A – Framework Topic Spaces

MIDI topic space and event contents

Topic	Event Contents	Description
Midi/Short/NoteOn	Timestamp, Channel, Note, Velocity	Note depressed
Midi/Short/NoteOff	Timestamp, Channel, Note, Velocity	Note released
Midi/Short/PolyKeyPressure	Timestamp, Channel, Note, Pressure	Note aftertouch – pressing a note after "bottoming out"
Midi/Short/ControlChange	Timestamp, Channel, Controller, Value	Change in a controller value
Midi/Short/ProgramChange	Timestamp, Channel, Program (Patch)	Change to a program patch number
Midi/Short/KeyPressure	Timestamp, Channel, Pressure	Note aftertouch – pressing a note after "bottoming out"
Midi/Short/PitchWheelChange	Timestamp, Channel, Value	Change in pitch wheel setting
Midi/Short/SystemMessage	Timestamp, Channel, Message Text	System message
Midi/Meta/SeqNumber	Timestamp, Sequence Number	Change in a Midi sequence number
Midi/Meta/Text	Timestamp, Text String	Arbitrary text event
Midi/Meta/Copyright	Timestamp, Copyright Text String	Copyright text
Midi/Meta/TrackName	Timestamp, Track Name String	Track name
Midi/Meta/InstrumentName	Timestamp, Instrument Name String	Instrument name
Midi/Meta/Lyric	Timestamp, Lyric String	Lyric
Midi/Meta/Marker	Timestamp, Marker String	Marker
Midi/Meta/CuePoint	Timestamp, CuePoint String	Cue point
Midi/Meta/ChannelPrefix	Timestamp, Value	Channel prefix
Midi/Meta/EndOfTrack	Timestamp, Empty	The end of a Midi track
Midi/Meta/Tempo	Timestamp, Value	Time in microseconds per beat
Midi/Meta/SMPTE	Timestamp, Value[0..4]	SMPTE time information
Midi/Meta/TimeSig	Timestamp, Value[0..3]	Time signature and other timing information
Midi/Meta/KeySig	Timestamp, Value	Key signature including Major/Minor
Midi/Meta/Vendor	Timestamp, String	Vendor specific information
Midi/Meta/UnknownMeta	Timestamp, String	Unknown Meta Event
Midi/Sysex/SysExcl	Timestamp, Value	Extension and manufacturer specific info
Midi/Sysex/SysSpExcl	Timestamp, Value	Extension and manufacturer specific info

Harmony topic space and event contents

Topic	Event Contents	Description
Harmony/Note	CCNote	Event contains note named using Chew Chen
Harmony/OPNDNote	OPNDNote	Event contains note defined by its offset, pitch class, name and duration (OPND)
Harmony/Beat	Timestamp, Strength	Beat strength and time
Harmony/CentreOfEffect	Point3f	X, Y, Z value of Chew Chen Centre of Effect in 3D space

Harmony Ontology topic space and event contents

Topic	Event Contents	Description
Harmony/Ontology/TimelineStart	URI, Timestamp	Start of Timeline
Harmony/Ontology/TimelineEnd	URI, Timestamp	End of Timeline
Harmony/Ontology/NoteStart	URI, Timestamp, Name, Octave	Start of Note
Harmony/Ontology/NoteEnd	URI, Timestamp	End of Note
Harmony/Ontology/ChordStart	URI, Chord, URI	Start of Chord and link to previous Chord
Harmony/Ontology/ChordEnd	Timestamp	End of Chord
Harmony.Ontology/ChordContains	URI	Chord contains Note with given URI
Harmony/Ontology/KeyStarts	Timestamp, Root, Mode	Start of Key with given Root and Mode
Harmony/Ontology/KeyEnds	Timestamp	End of Key

Framework control topics

Topic	Event Contents	Description
Control/Closing	URI, Timestamp	An application is terminating. Timestamp is the final event time generated by the publishing application e.g. a final MIDI Off command.

Control/PitchSpellingFinished	URI	Signals that the Pitch Spelling application has finished
Control/DatabaseFinished	URI	Signals that the database results capture application has finished
Control/TriplestoreFinished	URI	Signals that the triplestore capture application has finished
Control/Filename	URI, String	Identifies the testcase that is about to be published

Appendix B – Example Graphs in GXL

GXL for Figure 24 (e):

```
<?xml version="1.0" encoding="UTF-8"?>
<gxl xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd"
     xmlns:xlink="http://www.w3.org/1999/xlink">
      <node xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" id="Z1"
            pos="1170,611">
            <type xlink:href="Key"/>
            <attr name="root"><string>Cn</string></attr>
            <attr name="mode"><string>major</string</attr>
      </node>
</gxl>
```

GXL for Figure 24 (f):

```
<?xml version="1.0" encoding="UTF-8"?>
<gxl xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd"
     xmlns:xlink="http://www.w3.org/1999/xlink">
      <node xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" id="Z1"
            pos="1170,611">
            <type xlink:href="Key"/>
            <attr name="root"><string>Cn</string></attr>
            <attr name="mode"><string>major</string></attr>
      </node>
      <node xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" id="Z3"
            pos="1195,882">
            <type xlink:href="Chord"/>
            <attr name="name"><string>C047</string></attr>
      </node>
      <edge xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" from="Z3"
            to="Z1" id="E4">
            <type xlink:href="chordPartOf"/>
      </edge>
</gxl>
```

GXL for Figure 24 (g):

```
<?xml version="1.0" encoding="UTF-8"?>
<gxl xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd"
     xmlns:xlink="http://www.w3.org/1999/xlink">
      <node xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" id="Z1"
            pos="1170,611">
            <type xlink:href="Key"/>
            <attr name="root"><string>Cn</string></attr>
            <attr name="mode"><string>major</string></attr>
      </node>
      <node xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" id="Z3"
            pos="1195,882">
            <type xlink:href="Chord"/>
            <attr name="root"><string>Cn</string></attr>
            <attr name="name"><string>C047</string></attr>
      </node>
      <edge xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" from="Z3"
```

```
            to="Z1" id="E4">
            <type xlink:href="chordPartOf"/>
      </edge>
</gxl>
```

GXL for Figure 24 (h):

```
<?xml version="1.0" encoding="UTF-8"?>
<gxl xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd"
     xmlns:xlink="http://www.w3.org/1999/xlink">
      <node xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" id="Z3"
            pos="1195,882">
            <type xlink:href="Chord"/>
            <attr name="name"><string>C047</string></attr>
            <attr name="root"><string>Cn</string></attr>
      </node>
      <node xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" id="Z9"
            pos="1475,868">
            <type xlink:href="Chord"/>
            <attr name="name"><string>C07</string></attr>
      </node>
      <node xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" id="Z6"
            pos="1575,932">
            <type xlink:href="Note"/>
            <attr name="name"><string>Gn</string></attr>
      </node>
      <node xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" id="Z1"
            pos="1315,594">
            <type xlink:href="Key"/>
            <attr name="root"><string>Cn</string></attr>
            <attr name="mode"><string>major</string></attr>
      </node>
      <edge xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" from="Z3"
            to="Z1" id="E4">
            <type xlink:href="chordPartOf"/>
      </edge>
      <edge xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" from="Z3"
            to="Z9" id="E10">
            <type xlink:href="chordPrecedes"/>
      </edge>
      <edge xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" from="Z9"
            to="Z1" id="E11">
            <type xlink:href="chordPartOf"/>
      </edge>
      <edge xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" from="Z9"
            to="Z6" id="E7">
            <type xlink:href="chordContains"/>
      </edge>
</gxl>
```

GXL for Figure 24 (i):

```
<?xml version="1.0" encoding="UTF-8"?>
<gxl xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd"
     xmlns:xlink="http://www.w3.org/1999/xlink">
      <node xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" id="Z1"
            pos="1170,611">
            <type xlink:href="Key"/>
            <attr name="root"><string>Cn</string></attr>
            <attr name="mode"><string>major</string></attr>
      </node>
      <node xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" id="Z3"
            pos="1195,882">
            <type xlink:href="Chord"/>
            <attr name="root"><string>Cn</string></attr>
            <attr name="name"><string>C047</string></attr>
      </node>
      <node xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" id="Z6"
            pos="1250,950">
            <type xlink:href="MIDI"/>
      </node>
      <edge xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" from="Z3"
            to="Z1" id="E4">
            <type xlink:href="chordPartOf"/>
      </edge>
      <edge xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" from="Z3"
            to="Z6" id="E5">
            <type xlink:href="unknown"/>
      </edge>
      <edge xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" from="Z1"
            to="Z6" id="E6">
            <type xlink:href="unknown"/>
      </edge>
</gxl>
```

Appendix C – GXL to SQL Transformation

This appendix illustrates an early version of the GXL2SQL transform which converts a GXL graph description into SQL. The GXL structure does not include separate attributes for nodes and edges which were included in later versions.

The input GXL description:

```
<?xml version="1.0" ?>
<!DOCTYPE gxl [<!ELEMENT gxl (node | edge)*>
<!ELEMENT edge ANY>
<!ELEMENT node ANY>
<!ATTLIST node id ID #REQUIRED>
<!ATTLIST edge id ID #REQUIRED>]>
<gxl xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd"
     xmlns:xlink="http://www.w3.org/1999/xlink">
  <node xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" id="Z1"
        pos="871,623">
      <type xlink:href="#KEY_TYPE"/>
      <attr name="name">
          <string>GMajor</string>
      </attr>
  </node>
  <node xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" id="Z3"
        pos="744,816">
      <type xlink:href="#CHORD_TYPE"/>
      <attr name="name">
          <string>C</string>
      </attr>
  </node>
  <node xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd" id="Z5"
        pos="1118,898">
      <type xlink:href="#CHORD_TYPE"/>
      <attr name="name">
          <string>D</string>
      </attr>
  </node>
  <edge xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd"
        from="Z3" to="Z5" id="E6">
      <type xlink:href="unknown"/>
  </edge>
  <edge xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd"
        from="Z3" to="Z1" id="E7">
      <type xlink:href="unknown"/>
  </edge>
  <edge xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd"
        from="Z5" to="Z1" id="E8">
      <type xlink:href="unknown"/>
  </edge>
</gxl>
```

The generated SQL:

```
DECLARE GLOBAL TEMPORARY TABLE E6(HEAD VARCHAR(100), BODY
VARCHAR(100),     SOURCE_DOC INTEGER)
            NOT LOGGED ON COMMIT PRESERVE ROWS;

insert into SESSION.E6(HEAD, BODY) select distinct HEAD, BODY from
            new com.ibm.db2j.GaianTable('TRIPLES' ) GT
            where HEAD_TYPE = 'harmony:Chord'
            and HEAD like '%C%'
            and   BODY_TYPE = 'harmony:Chord'
            and BODY like '%D%';

DECLARE GLOBAL TEMPORARY TABLE E7(HEAD VARCHAR(100), BODY
VARCHAR(100),     SOURCE_DOC INTEGER)
            NOT LOGGED ON COMMIT PRESERVE ROWS;

insert into SESSION.E7(HEAD, BODY) select distinct HEAD, BODY from
            new com.ibm.db2j.GaianTable( 'TRIPLES' ) GT
            where HEAD_TYPE = 'harmony:Chord'
            and HEAD like '%C%'
            and   BODY_TYPE = 'harmony:Key'
            and BODY like '%GMajor%';

DECLARE GLOBAL TEMPORARY TABLE E8(HEAD VARCHAR(100), BODY
VARCHAR(100),     SOURCE_DOC INTEGER)
            NOT LOGGED ON COMMIT PRESERVE ROWS;

insert into SESSION.E8(HEAD, BODY) select distinct HEAD, BODY from
            new com.ibm.db2j.GaianTable( 'TRIPLES' ) GT
            where HEAD_TYPE = 'harmony:Chord'
            and HEAD like '%D%'
            and   BODY_TYPE = 'harmony:Key'
            and BODY like '%GMajor%';

SELECT DISTINCT
SESSION.E7.BODY as Z1, SESSION.E6.HEAD as Z3, SESSION.E6.BODY as Z5
FROM
 SESSION.E6,
 SESSION.E7,
 SESSION.E8
WHERE
SESSION.E7.BODY != SESSION.E6.HEAD and SESSION.E7.BODY !=
SESSION.E6.BODY and SESSION.E6.HEAD != SESSION.E6.BODY
 and SESSION.E6.HEAD = SESSION.E7.HEAD
 and SESSION.E6.BODY = SESSION.E8.HEAD
 and SESSION.E7.BODY = SESSION.E8.BODY
;
```

The GXL2SQL transform:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:gxl="http://www.gupro.de/GXL/gxl-1.0.dtd"
                xmlns:xlink="http://www.w3.org/1999/xlink"
                version="1.0">
```

```
    <xsl:output method="text" encoding="utf-8" />
    <xsl:variable name="lookup" select="document('lookup.xml')" />

    <xsl:template match="/">
        <xsl:apply-templates select="//gxl:edge" mode="table" />
SELECT DISTINCT
        <xsl:apply-templates select="//gxl:node" mode="select" />
FROM
        <xsl:apply-templates select="//gxl:edge" mode="from" />
WHERE
        <xsl:apply-templates select="//gxl:node" />
        <xsl:apply-templates select="//gxl:edge" />
;
    </xsl:template>

    <xsl:template match="gxl:edge" mode="table">
        <xsl:variable name="fmNode" select="id(@from)" />
        <xsl:variable name="toNode" select="id(@to)" />

        <xsl:variable name="typeFm" select="$lookup//entry[@nodeType =
substring-after($fmNode/gxl:type/@xlink:href,'#')]/@entityType" />
        <xsl:variable name="typeTo" select="$lookup//entry[@nodeType =
substring-after($toNode/gxl:type/@xlink:href,'#')]/@entityType" />

        <xsl:value-of select="$typeFm" /></xsl:message>
        <xsl:value-of select="$typeTo" /></xsl:message>

DECLARE GLOBAL TEMPORARY TABLE <xsl:value-of select="@id" />(HEAD
VARCHAR(100), BODY VARCHAR(100),    SOURCE_DOC INTEGER)
            NOT LOGGED ON COMMIT PRESERVE ROWS;

insert into SESSION.<xsl:value-of select="@id" />(HEAD, BODY) select
distinct HEAD, BODY from
            new com.ibm.db2j.GaianTable( 'TRIPLES' ) GT
            where HEAD_TYPE = '<xsl:value-of select="$typeFm" />'
            and HEAD like '%<xsl:value-of select="$fmNode"/>%'
            and   BODY_TYPE = '<xsl:value-of select="$typeTo"/>'
            and BODY like '%<xsl:value-of select="$toNode"/>%';

    </xsl:template>

    <xsl:template match="gxl:node" mode="select">
        <xsl:variable name="colName">
            <xsl:call-template name="getTableAndColumnForNode"/>
        </xsl:variable>
        <xsl:variable name="entityType"
                  select="substring-after(gxl:type/@xlink:href,'#')"/>
        <xsl:variable name="entityName">
            <xsl:choose>
                <xsl:when test= "$entityType = 'DOCUMENT_TYPE'">
                    <xsl:value-of select="concat('DOCUMENT_',@id)"/>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:value-of select="@id"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:variable>

        <xsl:value-of select="$colName"/> as
        <xsl:value-of select="$entityName"/>

        <xsl:if test="not(position() = last())">
            <xsl:text>, </xsl:text>
        </xsl:if>
    </xsl:template>

    <xsl:template match="gxl:edge" mode="from">
 SESSION.<xsl:value-of select="@id"/>
        <xsl:if test="not(position() = last())">
            <xsl:text>, </xsl:text>
        </xsl:if>
    </xsl:template>

    <xsl:template match="gxl:node">
        <xsl:if test="not(position() = 1 or position() = last())">
            <xsl:text> and </xsl:text>
        </xsl:if>
        <xsl:variable name="colNameA">
            <xsl:call-template name="getTableAndColumnForNode"/>
        </xsl:variable>
        <xsl:for-each select="following-sibling::gxl:node">
            <xsl:variable name="colNameB">
                <xsl:call-template name="getTableAndColumnForNode"/>
            </xsl:variable>
            <xsl:value-of select="$colNameA"/> !=
            <xsl:value-of select="$colNameB"/>
            <xsl:if test="not(position() = last())">
                <xsl:text> and </xsl:text>
            </xsl:if>
        </xsl:for-each>
    </xsl:template>

    <xsl:template match="gxl:edge" mode="from">
 SESSION.<xsl:value-of select="@id"/>
        <xsl:if test="not(position() = last())">
            <xsl:text>, </xsl:text>
        </xsl:if>
    </xsl:template>

    <xsl:template match="gxl:edge">
        <xsl:variable name="thisEdge" select="@id"/>
        <xsl:variable name="fmNode" select="@from"/>
        <xsl:variable name="toNode" select="@to"/>

        <xsl:for-each select="following-sibling::gxl:edge[($fmNode =
./@from)]">
 and SESSION.
            <xsl:value-of select="$thisEdge"/>.HEAD = SESSION.
            <xsl:value-of select="@id"/>.HEAD
        </xsl:for-each>
        <xsl:for-each select="following-sibling::gxl:edge[($fmNode =
./@to)]">
 and SESSION.
            <xsl:value-of select="$thisEdge"/>.HEAD = SESSION.
            <xsl:value-of select="@id"/>.BODY
        </xsl:for-each>
        <xsl:for-each select="following-sibling::gxl:edge[($toNode =
./@from)]">
 and SESSION.
            <xsl:value-of select="$thisEdge"/>.BODY = SESSION.
            <xsl:value-of select="@id"/>.HEAD
        </xsl:for-each>
        <xsl:for-each select="following-sibling::gxl:edge[($toNode =
./@to)]">
 and SESSION.
            <xsl:value-of select="$thisEdge"/>.BODY = SESSION.
            <xsl:value-of select="@id"/>.BODY
        </xsl:for-each>
    </xsl:template>

    <xsl:template name="getTableAndColumnForNode">
        <xsl:variable name="anEdge" select="//gxl:edge[(./@from =
current()/@id) or (./@to = current()/@id)][1]"/>
        <xsl:variable name="whichEnd">

        <xsl:choose>
            <xsl:when test="$anEdge/@from = @id">
                <xsl:value-of select="'HEAD'"/>
            </xsl:when>
            <xsl:otherwise>
                <xsl:value-of select="'BODY'"/>
            </xsl:otherwise>
        </xsl:choose>
        </xsl:variable>
        <xsl:value-of
select="concat('SESSION.',$anEdge/@id,'.',$whichEnd)" />
    </xsl:template>
</xsl:stylesheet>
```

Appendix D – The Harmony Relational Model

```
-- ============================================================
--   database name    :  tripleDB
--   database system  :  DERBY
--   creation date    :  26-03-2008
-- ============================================================
drop table TRIPLES;
drop table DOC_TABLE;

-- ============================================================
--   Table : DOC_TABLE
-- ============================================================
create table DOC_TABLE (
      DOC_HASH integer,
      DOC_NAME varchar(200)
);

-- ============================================================
--   Table : TRIPLES
-- ============================================================
create table TRIPLES (
      TRANS integer,
      HEAD varchar(100),
      HEAD_TYPE varchar(40),
      BODY varchar(100),
      BODY_TYPE varchar(40),
      SOURCE_DOC integer,
      TIMESTAMP timestamp,
      ASSOCIATION varchar(200)
      );
create unique index i1 on TRIPLES(TRANS);
create index i2 on TRIPLES(head);
create index i3 on TRIPLES(association);
create index head on TRIPLES(head asc);
create index body on TRIPLES(body asc);
create index head_type on TRIPLES(head_type asc);
create index body_type on TRIPLES(body_type asc);
create index hh on TRIPLES(head asc, head_type asc);
create index bb on TRIPLES(body asc, body_type asc);
create index source on TRIPLES(source_doc asc);

-- ============================================================
-- Drop Views
-- ============================================================
DROP VIEW K2K_FORWARD_VIEW;
DROP VIEW K2K_BACKWARD_VIEW;
DROP VIEW C2C_FORWARD_VIEW;
DROP VIEW C2C_BACKWARD_VIEW;
DROP VIEW K2C_VIEW;
DROP VIEW C2K_VIEW;
DROP VIEW C2N_VIEW;
DROP VIEW N2C_VIEW;
DROP VIEW KEY_VIEW;
DROP VIEW CHORD_VIEW;

-- ============================================================
-- Create Key View
-- ============================================================
CREATE VIEW KEY_VIEW (K_URL, K_ROOT, K_MODE) AS
      SELECT T1.HEAD AS K_URL, T1.BODY AS K_ROOT,T2.BODY AS K_MODE
      FROM
            TRIPLES T1,
            TRIPLES T2
      WHERE T1.ASSOCIATION = 'keyRoot'
      AND T2.ASSOCIATION = 'keyMode'
      AND T1.HEAD = T2.HEAD;

-- K2 follows K1
CREATE VIEW K2K_FORWARD_VIEW (K1_URL, K1_ROOT, K1_MODE, K2_URL,
                              K2_ROOT, K2_MODE) AS
      SELECT K1.K_URL AS K1_URL, K1.K_ROOT AS K1_ROOT, K1.K_MODE AS
             K1_MODE, K2.K_URL AS K2_URL, K2.K_ROOT AS K2_ROOT,
             K2.K_MODE AS K2_MODE
      FROM
            TRIPLES T1,
            KEY_VIEW K1,
            KEY_VIEW K2
      WHERE T1.ASSOCIATION = 'keyPrecedes'
      AND T1.HEAD = K1.K_URL
      AND T1.BODY = K2.K_URL;

-- K2 precedes K1
CREATE VIEW K2K_BACKWARD_VIEW (K1_URL, K1_ROOT, K1_MODE, K2_URL,
                               K2_ROOT, K2_MODE) AS
      SELECT K1.K_URL AS K1_URL, K1.K_ROOT AS K1_ROOT, K1.K_MODE AS
             K1_MODE, K2.K_URL AS K2_URL, K2.K_ROOT AS K2_ROOT,
             K2.K_MODE AS K2_MODE
      FROM
            TRIPLES T1,
            KEY_VIEW K1,
            KEY_VIEW K2
      WHERE T1.ASSOCIATION = 'keyFollows'
      AND T1.HEAD = K1.K_URL
      AND T1.BODY = K2.K_URL;

-- ============================================================
-- Create Chord Views
-- ============================================================
CREATE VIEW CHORD_VIEW (C_URL, C_NAME, C_ROOT) AS
      SELECT T1.HEAD AS C_URL, T1.BODY AS C_NAME, T2.BODY AS C_ROOT
      FROM
            TRIPLES T1,
            TRIPLES T2
      WHERE T1.ASSOCIATION = 'chordName'
      AND T2.ASSOCIATION = 'chordRoot'
      AND T1.HEAD = T2.HEAD;

      -- C2 follows C1
CREATE VIEW C2C_FORWARD_VIEW (C1_URL, C1_NAME, C1_ROOT, C2_URL,
                              C2_NAME, C2_ROOT) AS
      SELECT C1.C_URL AS C1_URL, C1.C_NAME AS C1_NAME, C1.C_ROOT AS
             C1_ROOT, C2.C_URL AS C2_URL, C2.C_NAME AS C2_NAME,
             C2.C_ROOT AS C2_ROOT
      FROM
            TRIPLES T1,
            CHORD_VIEW C1,
            CHORD_VIEW C2
      WHERE T1.ASSOCIATION = 'chordPrecedes'
      AND T1.HEAD = C1.C_URL
      AND T1.BODY = C2.C_URL;

-- C2 precedes C1
CREATE VIEW C2C_BACKWARD_VIEW (C1_URL, C1_NAME, C1_ROOT, C2_URL,
                              C2_NAME, C2_ROOT) AS
      SELECT C1.C_URL AS C1_URL, C1.C_NAME AS C1_NAME, C1.C_ROOT AS
             C1_ROOT, C2.C_URL AS C2_URL, C2.C_NAME AS C2_NAME,
             C2.C_ROOT AS C2_ROOT
      FROM
            TRIPLES T1,
            CHORD_VIEW C1,
            CHORD_VIEW C2
      WHERE T1.ASSOCIATION = 'chordFollows'
      AND T1.HEAD = C1.C_URL
      AND T1.BODY = C2.C_URL;

-- ============================================================
-- Create Key-Chord Views
-- ============================================================
CREATE VIEW K2C_VIEW (K_URL, K_ROOT, K_MODE, C_URL, C_NAME, C_ROOT) AS
      SELECT K1.K_URL AS K_URL, K1.K_ROOT AS K_ROOT, K1.K_MODE AS
             K_MODE, C1.C_URl AS C_URL, C1.C_NAME AS C_NAME, C1.C_ROOT
             AS C_ROOT
      FROM
            TRIPLES T1,
            CHORD_VIEW C1,
            KEY_VIEW K1
      WHERE T1.ASSOCIATION = 'keyContains'
      AND T1.HEAD = K1.K_URL
      AND T1.BODY = C1.C_URL;

CREATE VIEW C2K_VIEW (C_URL, C_NAME, C_ROOT, K_URL, K_ROOT, K_MODE) AS
      SELECT C1.C_URL AS C_URL, C1.C_NAME AS C_NAME, C1.C_ROOT AS
             C_ROOT, K1.K_URL AS K_URL, K1.K_ROOT AS K_ROOT, K1.K_MODE
             AS K_MODE
      FROM
            TRIPLES T1,
            CHORD_VIEW C1,
            KEY_VIEW K1
      WHERE T1.ASSOCIATION = 'chordPartOf'
      AND T1.HEAD = C1.C_URL
      AND T1.BODY = K1.K_URL;

-- ============================================================
-- Create Chord-Note Views
-- ============================================================
CREATE VIEW C2N_VIEW(C_URL, C_NAME, C_ROOT, N_URL, N_TYPE) AS
      SELECT C1.C_URL AS C_URL, C1.C_NAME AS C_NAME, C1.C_ROOT AS
             C_ROOT, T2.HEAD AS N_URL, T2.BODY AS N_TYPE
      FROM
            TRIPLES T1,
            TRIPLES T2,
            CHORD_VIEW C1
      WHERE T1.ASSOCIATION = 'chordContains'
      AND T1.HEAD = C1.C_URL
      AND T2.ASSOCIATION = 'noteType'
      AND T2.HEAD = T1.BODY;

CREATE VIEW N2C_VIEW(N_URL, N_TYPE, C_URL, C_NAME, C_ROOT) AS
      SELECT C1.C_URL AS C_URL, C1.C_NAME AS C_NAME, C1.C_ROOT AS
```

```
       C_ROOT, T2.HEAD AS N_URL, T2.BODY AS N_TYPE
FROM
      TRIPLES T1,
      TRIPLES T2,
      CHORD_VIEW C1
WHERE T1.ASSOCIATION = 'notePartOf'
AND T1.BODY = C1.C_URL
AND T2.ASSOCIATION = 'noteType'
AND T2.HEAD = T1.HEAD;
```

Appendix E - Musical Codes and Representation

This appendix formed part of the MPhil transfer thesis and is provided here as background information for the choice of MIDI as the low level representation of music used in this thesis.

Musical information has traditionally been communicated by the musical score through a process known as common musical notation (CMN). This is a visual representation of music composed of graphical symbols arranged in a two-dimensional space; the printed or hand written page. Interpreting the symbols to form music involves the analysis of the graphical and logical structure of the document. This involves both a descriptive and conceptual interpretation of the music through interpretation of the musical events to be performed together with abstract musical concepts that contribute towards an artist's performance of the work [MacMillan02].

CMN has evolved over the years as a system of symbols for the purpose of describing and disseminating consistent musical practice. It is a system that is relatively, but not completely, self-consistent and stable. Music itself is evolving as new styles and genres emerge and CMN's inherent flexibility and extensibility has allowed it to adapt to new demands. The adaptability of CMN means that it is not a perfect guide for the reproduction of sounds either computationally of by human performers. The apparent continuity of graphical representation does not guarantee the same continuity in interpretation and practice. However, CMN is the basis of all attempts to preserve and publish the corpus of music we know today. This corpus owes its existence to a generally understood system of graphical communication.

The desire to encode music for mechanical or electronic reproduction has given rise to alternative representations of music. For example, the rolls used by player pianos provided a means to mechanically capture a performance using a graphical code using a bar and line representation that is not dissimilar to the notation provided in modern music sequencer software. This process has developed further in that the piano rolls produced in the early 20th century are now being scanned and archived on the Web as MIDI files [Smythe03].

The purpose of this section is to review and comment on the music encoding techniques used to represent CMN within a computer system. The discussion at the end of this section addresses the applicability of the encoding techniques for describing music in a real-time analysis framework.

MIDI and derivatives

The Musical Instrument Digital Interface (MIDI) specification was first published in 1982 and has been regularly updated [MMA01]. It was the first musical encoding scheme to describe a transfer protocol by which electronic musical instruments could communicate with each other. Its major use is still for the control of synthesisers and other instruments but it has also been adopted as a file format for the distribution of compositions. The specification has generally remained unchanged since its first introduction but there have been many extensions proposed and some implemented over the years.

MIDI is a serial protocol running at 31.25 Kbaud to form a daisy chain through all connected instruments. The protocol is optimised using a binary representation that ensures the events specified by the protocol arrive at the instruments in a timely manner. MIDI allows a total of 16 logical channels and instruments may be set up to receive information on a particular channel.

A MIDI message is called an event and contains commands such as *NoteOn*, which specifies both the pitch and amplitude of the note being played. In MIDI, a note's pitch is defined by a number between 0 and 127 where 0 represents C0 (5 octaves below middle C) and 127 represents G10 (5 octaves above middle C). Other MIDI events include *NoteOff* and after touch (the modification of a note currently being played), specifying a different sound and system messages.

Each track in a MIDI file is stored as a binary representation of the event messages together with a time stamp for each event. The time stamp ensures events are delivered sequentially to the attached instruments causing trigger events on their selected channel. This temporal ordering of the events means that the MIDI protocol may be considered a data stream for the purposes of this thesis.

A problem with the MIDI protocol is that it was designed for the interconnection and control of electronic musical instruments and not as a musical notation standard to represent CMN. Other encoding schemes described in this section have been designed to represent musical scores and therefore include richer descriptions of musical notation and performance than the MIDI specifications. Proposals have been made to extend the MIDI specification to make it suitable for printing scores. NoTAMIDI [Nordli97] suggests meta-events within MIDI to facilitate a more complete representation of attributes for musical printing using data captures by an electronic keyboard. Expressive MIDI [Cooper97] was designed to make data capture by OCR from a printed score more practical for generating MIDI output. Hewlett [Hewlett97b] proposes extensions to allow conversion of MIDI note numbers for generating accurate enharmonic notation for harmonic analysis which is otherwise missing from MIDI file information. Augmented MIDI [Mathews97a] includes extensions for more articulate control of MIDI devices in a real-time controller environment. This could include allowing a user to vary the degree of staccato as well as the degree of accentuation and other nuances from note to note.

Monophonic Encoding

Monophonic music is easier to handle and represent than polyphonic music and therefore its use in research applications has been much greater. The codes cited in the following section have generally been held within a single field within a relational database allowing combined access to the music and associated text that is valuable in the management of information about musical sources.

The Essen Associative Code (EsAC) [Schaffrath97] was developed in the early 1980s for representing monophonic music – in particular folk music. EsAC consists entirely of ASCII characters and was designed to run on DOS machines. It was designed to occupy one field of a relational database to allow studies of musical and associated contextual attributes. As of 1994, more than 14,000 folk songs have been encoded with EsAC. The “associative” aspect of EsAC is its simplicity, allowing the association between sight-reading and sight-singing. Its pitch encoding uses scale-degree numbers corresponding to the moveable symbols of the tonic sol-fah scale.

Chromatic alterations are also included for sharps and flats and phrasing is included in the encoding. A suite of analysis tools are included to support the encoded music allowing analysis and comparison of melodies contained within the archives.

The Plaine and Easie Code [Howard97] was developed as a means of representing musical notation with ordinary typewriter symbols for use in bibliographic applications such as card catalogues and indexes for encoding musical incipits; this means the first few words of a book – in this case means a short phrase of music.

Polyphonic Encoding

Since the 1980s and the widespread availability of computing memory and processing speed attention has moved to the encoding of polyphonic notation and the subsequent processing of musical repertoire. This processing is intended primarily for sound production, notation and analysis.

The purpose of the Humdrum Toolkit [Huron97] and its associated encoding, Kern, is to allow the posing and answering of musicological questions. These questions are typically placed against collections of encoded music. The toolkit is a set of utilities written in AWK, C and YACC. These utilities are applied to a set of files using shell scripts in a UNIX environment. Humdrum files are standard ASCII files that use the Kern notation. The Kern representation allows musical pitch, duration, articulation, ornamentation and timbre to be documented in addition to editorial and other notational marks such as barlines, bowing direction, beams and stem direction. Kern may be combined with other symbol-schemes within Humdrum to permit other encoding schemes such as musical dynamics, visual layout of scores and sound synthesis. Each stave of the score is represented by a separate column of text in Kern that gives the impression of a musical score (normally read from left to right) being read from top to bottom of the page. The following example illustrates the first four bars of the Mozart Clarinet Quintet encoded in the Kern format. The example shows the five instrumental staves together with a column (or spine) used to capture dynamics for all the instruments.

```
!! Mozart: Trio II from Clarinet Quintet
**kern         **kern         **kern         **kern         **kern         **dyn
!violon-       !viola         !violino       !violino       !clarinet
!cello         !              !II            !I             !in A
!*Icello       *Iviola        *Iviolin       *Iviolin       *Iclarinet     *
*ICstr         *ICstr         *ICstr         *ICstr         *ICww          *
*sys1          *sys1          *sys1          *sys1          *sys1          *sys1
*staff5        *staff4        *staff3        *staff2        *staff1        *staff*
*clefF4        *clefC3        *clefG2        *clefG2        *clefG2        *
*M3/4          *M3/4          *M3/4          *M3/4          *M3/4          *
*k[f#c#g#]     *k[f#c#g#]     *k[f#c#g#]     *k[f#c#g#]     *k[f#c#g#]     *
*A:            *A:            *A:            *A:            *A:            *
*              *              *              *              *Tr+2d+3c      *
4r             4r             4r             4r             (8a\           p
.              .              .              .              8cc#\          .
=1             =1             =1             =1             =1             =1
4A\            4r             4r             4r             8ee\           .
.              .              .              .              8cc#\          .
4r             4c#\           4e/            4a/            4aa\)          .
4r             4c#\           4e/            4a/            (8ee\          .
.              .              .              .              8cc#\          .
=2             =2             =2             =2             =2             =2
4D\            4r             4r             4r             8b\            .
.              .              .              .              8dd\           .
4r             4B/            4f#/           4a/            4ff#\)         .
4r             4B/            4f#/           4a/            (8dd\          .
.              .              .              .              8b\            .
=3             =3             =3             =3             =3             =3
4E\            4r             4r             4r             8a\            .
.              .              .              .              8g#\           .
4r             4B/            4d/            4g#            8cc#\          .
.              .              .              .              8b\            .
4r             4B/            4d/            4g#            8ee\           .
.              .              .              .              8dd\)          .
=4             =4             =4             =4             =4             =4
4F#\           4r             4r             4r             (4b#\          .
4r             4A/            4c#/           4a/            4cc#\)         .
4r             4a/            4c#/           4a/            (8a\           .
.              .              .              .              8cc#\          .
*-             *-             *-             *-             *-             *-
```

Figure 28: Kern encoding

MuseData [Hewlett97c] is intended to encode the logical content of musical scores and captures both notational and sound information. The representation is not intended to be complete since it is expected that MuseData files would serve as source files for generating graphics files and MIDI sound files. The reasoning for this is that when music is encoded, there is often more information contained within the file than a composer intended to convey. Additionally, since MuseData was used with other processing software, other packages will add specific information about how a graphic rendering of the data should look or how a realisation of the data should sound. The organisation of files is an integral part of the MuseData representation. Each file represents the encoding of a single musical part from a movement or piece. The individual files may then be organised into a hierarchical directory structure within a database. For a given musical work, MuseData files are divided into two types. The first type contains data for pitch and duration of notes. The second type includes large amounts of additional information to support printing, interpretive and analytical applications. This information will support sound generation through MIDI, full and short score printing, separate part printing, analysis, MIDI specific data (channel and instrument assignments) and other management data. The format of single files consists of a set of time-ordered, variable length ASCII records organised as header records, musical attributes, note records and an end of file marker. The following example illustrates the first four bars of the clarinet part from the Mozart Clarinet Quintet in MuseData format.

```
04/16/93 E. Correia
WK#:581       MV#:3c
Breitkopf & H„rtel, Vol. 13
Clarinet Quintet
Trio II
Clarinet in A
1 0
Group memberships: sound, score
sound: part 1 of 5
score: part 1 of 5
$  K:0   Q:6   T:3/4   X:-11   C:4
C5     3        e     d  [      (&0p
E5     3        e     d  ]
measure 1
G5     3        e     d  [
E5     3        e     d  ]
C6     6        q     d         )
G5     3        e     d  [      (
E5     3        e     d  ]
measure 2
D5     3        e     d  [
F5     3        e     d  ]
A5     6        q     d         )
F5     3        e     d  [      (
D5     3        e     d  ]
measure 3
C5     3        e     d  [
B4     3        e     d  =
E5     3        e     d  =
D5     3        e     d  =
G5     3        e     d  =
F5     3        e     d  ]      )
measure 4
D#5    6        q #   d         (
E5     6        q     d         )
C5     3        e     d  [      (
E5     3        e     d  ]
```

Figure 29: MuseData encoding

MusicXML [Good01] is becoming a widely used interchange format for representing sheet music and musical notation. It is intended to act as an intermediate encoding to allow translation between the proprietary binary codes used in many popular sheet music editors. Some publishing packages make use of MIDI as an interchange format, but MIDI is unable to represent many of the features needed for successful musical score publication. MusicXML is intended to overcome these deficiencies.

MusicXML builds on the earlier development of the MuseData and Humdrum formats and represents scores either *partwise* (measures within parts) or *timewise* (parts within measures). This dual approach recognises that musical scores are inherently two-dimensional. Since XML is hierarchical in structure, MusicXML provides two DTDs (one for each representation) with XSLT transformations to move between the two representations.

The following example is a trivial piece of MusicXML representing a single note.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE score-partwise PUBLIC
  "-//Recordare//DTD MusicXML 0.6b Partwise//EN"
  "http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise>
  <part-list>
    <score-part id="P1">
      <part-name>Music</part-name>
    </score-part>
  </part-list>
  <part id="P1">
    <measure number="1">
      <attributes>
        <divisions>1</divisions>
        <key>
          <fifths>0</fifths>
        </key>
        <time>
          <beats>4</beats>
          <beat-type>4</beat-type>
        </time>
        <clef>
          <sign>G</sign>
          <line>2</line>
        </clef>
      </attributes>
      <note>
        <pitch>
          <step>C</step>
          <octave>4</octave>
        </pitch>
        <duration>4</duration>
        <type>whole</type>
      </note>
    </measure>
  </part>
</score-partwise>
```

Figure 30: MusicXML

MusicXML supports a superset of the MuseData features and supports a number of translators into and out of popular music publishing packages [Good02]. Since MusicXML is a vocabulary of XML, analysis and manipulation of musical scores encoded with MusicXML can make use of other XML standards. XSLT has already been mentioned for transformation between different representations of MusicXML and demonstrations of using XML Query to formulate queries against MusicXML encoded scores have been reported. Translators between MIDI and MusicXML have also been produced.

Discussion

With the exception of MIDI, the music encoding schemes reviewed in these sections are used as file descriptions for the purposes of score printing, static musicological analysis or musical information retrieval. MIDI may be used for this purpose, but it was designed for real-time control of electronic instruments in both recording and performance environments. Its structure is based on events occurring at known times and so it may be considered a form of data that is suitable for streaming and processing in real-time.

MIDI control streams have limited capability when compared to other polyphonic representations such as MuseData, Kern and MusicXML. However, these representations contain much richer encoding allowing full score printing (in the case of MusicXML) or analysis using the Humdrum toolkit (as in the case of Kern). Kern relies on the expertise of an author in capturing the richness of the musical score using the language syntax.

The question arises whether such a rich musical representation is needed for real-time analysis. The input format to Temperley's [Temperley01] preference rule based analysis is simply a list of note start and end times (in milliseconds). This is equivalent to the MIDI *NoteOn* and *NoteOff* events with an associated timestamp. The purpose of any analysis is to enrich the event stream; therefore requiring a richly encoded input (whilst advantageous) defeats the purpose of this thesis.

The conclusion reached in this section is that whilst the richer encoding techniques (particularly Kern) reviewed might be appropriate for describing information extracted from a real-time musical stream, MIDI remains the best candidate for representing raw musical events. The elements of its compact structure contain the basic data needed as input to analysis components that can extract higher-level information about the harmonic structure of the musical performance with the objective of approximating to the level of harmonic description provided by other representations such as MuseData and Kern. The only additional information required would be an accurate timestamp associated with the MIDI event. This information is readily obtained, but provides an added complication if the analysis and performance framework is deployed on a distributed processing infrastructure where timing information may be skewed due to different processor clocks and network latency.