A Flexible Music Composition Engine

Maia Hoeberechts, Ryan J. Demopoulos and Michael Katchabaw

Department of Computer Science
Middlesex College
University of Western Ontario
London, Canada N6A 5B7
hoebere, rdemopo2, katchab@csd.uwo.ca

Abstract. There is increasing interest and demand for adaptive music composition systems, which can change the character of generated music on the fly, for use in diverse areas such as video game music generation, film score composition, and development of interactive composition tools. This paper describes AMEE™ (Algorithmic Music Evolution Engine), a prototype system for dynamic music generation. The features which distinguish AMEE™ from other composition systems are the use of a *pipelined architecture* in the generation process to allow a structured, yet flexible approach to composition, the inclusion of *pattern libraries* for storing and accessing musical information, and the addition of an *emotion mapper* which allows music to be altered according to emotional characteristics.

1. Introduction

Traditionally, the music one finds in video games consists of a static set of compositions packaged with the game. Creating game music is often an expensive proposition, requiring either the licensing of existing pieces from artists, or hiring professionals for custom compositions. Furthermore, having a static collection of music can become repetitive, and fixed selections cannot be altered as part of the user's interactive experience during game play. Considering these limitations, we are working towards an alternative: music generated on demand during game play, which can be influenced by game events and change its character on the fly.

This paper describes AMEE™ (Algorithmic Music Evolution Engine), a prototype system for dynamic music generation. AMEE™ was designed with several principal goals in mind:

(a) Permit maximal flexibility in the composition process. The engine can either generate music without any restrictions, or a human composer can guide musical choices.
(b) Provide an extensible architecture that can be easily integrated with other software.
(c) Incorporate a multi-level application programming interface which makes AMEE™ functionality accessible to users with varying levels of musical and/or programming knowledge.
(d) Reuse elements, such as note sequences and harmonic structure, both from existing human composed pieces or computer generated material.
(e) Allow music to be altered based on emotional characteristics such as happiness, sadness, anxiety, liveliness etc.

AMEE™ is an object-oriented system written in J# and Java. It includes high-level classes such as Musician and Instrument that model real-world entities involved in music composition. The features which distinguish AMEE™ from other composition systems are the use of a *pipelined architecture* in the generation process to allow a structured, yet flexible approach to composition, the inclusion of *pattern libraries* for storing and accessing musical information, and the addition of an *emotion mapper* which allows music to be altered according to emotional characteristics. A more detailed description of these features can be found in Section 3.

There are two main applications areas we envision for AMEE™ at the present time. First, AMEE™ could be embedded within application software to facilitate online, dynamic composition of music for immediate use within that application. For example, using the engine in a video game would allow endless variety in the game music, and since composition is done dynamically, the generated music could be tuned to reflect emotional context during game play. Alterations to the music could be initiated from within the game, by the game player, or both. Second, we foresee using AMEE™ as a basis for stand-alone composition tools. For example, consider a system which would permit collaboration among human composers who could exchange parts of pieces created with the engine, or share virtual musicians, instruments, and musical elements. The AMEE™ architecture can also support the creation of virtual bands and jam sessions.

AMEE™ is currently in a prototype stage. The software is fully functional, but there are many planned extensions and improvements that we are working on. This paper describes the features, design, implementation, and future development of the AMEE™ prototype.

1.1. Related Work

AMEE™ is an example of an automatic music composition system. These systems can be broadly classified into five categories: stochastic (random) methods, genetic/evolutionary approaches, recombination approaches, grammar/automata based methods, and interactive variants. An overview of some of these systems can be found in [3] and an historical account of some automatic music composition systems can be found in [1]. Concerning the above classification, AMEE™ is a flexible system which uses a combination of stochastic methods, recombination and interaction.

Recently, the terms *interactive music* or *adaptive music* have been adopted to describe music which changes its character according to context in a film, video game or real-life scenario. A comprehensive survey of these systems is beyond the scope of this paper, but the following are few examples of interactive music systems. MAgentA (**M**usical **Agent** **A**rchitecture) supports the generation of mood-appropriate background music by dynamically choosing composition algorithms which were previously associated with particular emotional states [2]. In [6], the authors describe a system for altering music to produce emotional variation based on *structural rules* (for example, affecting tempo) and *performance rules* (for example affecting accenting). Informal empirical testing showed some sucess of the system in perceived emotional content of the generated music. Scorebot [7] is a low-level API to manipulate music to be used in a film score based on scene information such as emotional content, timing and events. It enables the storage of musical themes which can be sent to manipulation modules that change the characteristics of the theme. Dynamic Object Music Engine (DOME) appears to be a system with similar goals to AMEE™, although not much information is publically available on its implementation details [9].

2. Overview of Design and Implementation

The prototype was developed in J# and Java. The source code of the version described in this paper was assembled and compiled under Microsoft Visual Studio .NET. We believe that the software can easily be ported to standard Java and with some modifications to J2ME, although we have not attempted these conversions at this time.

The software is divided into several main groups of classes:

- **The pipeline.** These classes control the flow of the music generation process, and are responsible for calling methods on the generating classes.
- **The producers.** These classes produce high level musical elements from which the composed piece is comprised (sections, blocks and musical lines).
- **The generators.** These classes create the lower level musical elements (harmonic patterns, motif patterns, modes, meters) from which the products are assembled. Generators can have library-based or pseudo-random implementations.
- **High-level classes.** Musician, Instrument, Performer, Piece Characteristics, Style, Mode, Meter, and Mood. These classes implement the real-world entities modelled in the engine.

For the prototype of AMEE™ described in this paper, we have provided rudimentary implementations of all the necessary classes in order for the engine to produce music. The classes are named using the convention "StandardClassName," for example the StandardPipeline and StandardMotifGenerator. The purpose of writing these classes was as a proof-of-concept for the method of music generation used by AMEE™.

2.1. Process of Music Generation

In order to use the engine, producers and generators must be created and loaded in the pipeline.

The first step is to use a GeneratorFactory to create the generators. Five types of generator are necessary for the engine: a StructureGenerator which creates the overall sectional structure of the piece (e.g. ABA form), a HarmonicGenerator which creates a sequence of chords for each section (e.g. I-IV-V-I), a MotifGenerator which creates short sequences of notes (e.g. a four note ascending scale of sixteenth notes beginning on the tonic), a ModeGenerator which returns modes for the piece (e.g. start in F+, progress to C+, divert to D- and return to F+), and a MeterGenerator (e.g. 4/4 time). Every generator contains at least one pseudo-random number component which is used for decisions it needs to make.

The generator might also contain a PatternLibrary which provides the generator with musical elements it can use directly, or as starting points for musical element generation. PatternLibraries are created in advance, and are intended to embody musical knowledge. For example, for the purposes of the prototype, we have created a "Bach" MotifPatternLibrary containing motifs from Bach's Invention No. 8 and a HarmonicPatternLibrary based on the same piece. Eventually, more extensive PatternLibraries should be created by musicians for distribution with the engine, and end users of the music generation package will also have the ability to add to the libraries.

Once the generators have been created, then the producers must be created using a ProducerFactory. There are four producers necessary. The SectionProducer uses the StructureGenerator to produce Sections, where a Section is a chunk of the piece with an associated length in seconds. Each Section contains a number of "blocks" which are short segments of the piece (for example, 4 bars) composed of a musical line played by each musician. The BlockProducer is responsible for deciding how to coordinate the musical lines using a HarmonicPattern created by a HarmonicGenerator associated with the BlockProducer. The musical lines themselves are composed by the LineProducer which uses a MotifGenerator to create the actual note sequences in the musical line played by each musician. Finally, an OutputProducer must be initialized which will convert the generated piece to MIDI and output it as audio.

When the producers have been initialized, they are loaded into a Pipeline. The Pipeline oversees the generation process and calls on each producer in turn to assemble the piece. An example of the process is as follows: the SectionProducer is called on to get a new section of the piece, then the BlockProducer returns the first block of that section, which in turn is passed to the LineProducer and filled in with notes, and lastly the completed block is sent to the OutputProducer for sound output. At each step in the pipeline, if desired the products can be sent to the EmotionMapper for Mood dependent adjustments. Music generation continues until the duration for the piece, specified by the user or determined by AMEE™, is completed.

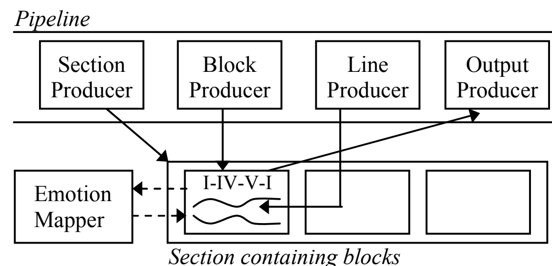

Figure 1: Illustration of pipelined generation

3. High-level Features of AMEE™

This section will describe the unique features of AMEE™, and highlight some of the design choices which were made.

3.1. Realistic Modelling

A key consideration in the design phase was that the entities involved in music creation reflect their real-life counterparts. At the highest level, the engine deals with the following classes: Musician, Instrument, Performer, PieceCharacteristics, Style, Mode, Meter, and Mood.

A Musician plays an Instrument and has a Mood. Also, a Musician has an Ability, and knows a number of Styles. Our intention was to model a real musician, who can play one instrument at a time, but has the ability to play that instrument in different styles with varying ability. Consider a pianist who is classically trained, but also plays some improvisational jazz. We would model this pianist as a Musician who knows (at least) three styles: classical, jazz and her own style – a personal repertoire of improvisational riffs.

The purpose of storing an ability for the musician is that eventually we would like to be able to model "good" and "bad" musicians. Some aspects we have considered are: ability to play an instrument in tune; ability to follow a beat; ability to contain one's mood and play in the manner desired by the conductor. The styles known by the musician will also reflect ability. On the other hand, we might want to model a musician with limited ability – perhaps in a video game role (imagine a bar scene where a bad band is playing), or perhaps eventually we might create an application where one has to "train" musicians, or one might want "bad" musicians simply for the entertainment value of hearing them play. Although our prototype does not currently make use of a Musician's Ability, all the hooks are in place to make this possible.

Our motivation for modelling musicians in this way is that we envisioned applications where users could eventually develop and trade musicians with one another. When considering an end-user of the software, we pictured users having collections of musicians each with their own skill set. The Musicians could be shared, traded, combined into groups, and perhaps even marketed.

3.2. Pattern Libraries

The purpose of the PatternLibraries is to allow new music to be generated by reusing compositional elements: either elements from existing pieces or elements which have been previously generated. Existing pieces used to extract musical elements will be in the public domain to ensure that no copyright issues are encountered. A PatternLibrary can be thought of as a repository of musical ideas. The four libraries which are currently used in AMEE™ are the HarmonicPatternLibrary, the MotifPatternLibrary, the ModePatternLibrary and the MeterPatternLibrary. Our motivation in creating the PatternLibraries was twofold: to give the system the ability to compose in different styles, and to provide a mechanism for exchanging and storing musical ideas. Each of these goals will be discussed in turn.

To answer the question, "What style is this piece of music?" you would listen for clues among the musical elements of the piece to determine its classification. The instruments being played are often an initial hint (the "sound" of the piece). Further to that, you would attend to the rhythmic structure, the harmony, the melody, the mode and the meter, and transitions between elements. Consider the knowledge of a jazz trumpet player, for instance Louis Armstrong. He knows typical harmonic progressions that will be used in a piece, and has many "riffs" in mind that he can use and improvise on. In our system, this knowledge would be captured in a HarmonicLibrary and MotifLibrary respectively.

How the libraries are used is determined by the implementation of the generators. A generator implementation could use the library as its only compositional resource, that is, all the musical elements returned by the generator are those taken from the library, or it could use the library patterns as starting points which are then modified, or it could return a mixture of library patterns and generated patterns (and naturally, it could ignore the library contents entirely and return only generated patterns). Thus the libraries are rich musical resources which can be flexibly used depending on the desired musical outcome.

Regarding the exchange of musical ideas, consider a situation where you meet someone using AMEE™ who has a very dynamic guitar player (let us call him Jesse Cook). The knowledge of the guitar player is contained in the libraries the guitar player is using for music generation. You could allow your Louis Armstrong player to learn how to make his trumpet sound like a flamenco guitar by incorporating the riffs from Jesse Cook's MotifLibrary into Louis Armstrong's existing library. A different possibility is that you could add the Jesse Cook player to your band. What would it sound like if the two musicians jammed together? Or, you could ask Jesse Cook and Louis Armstrong to collaborate in an Ensemble. Could they influence each other while playing? All of these functions are directly supported by the idea of PatternLibraries as embodiments of musical knowledge.

3.3. Pipelined Architecture

The pipelined architecture described in Section 2.1 has several significant advantages over other potential architectures. The generation process can be pictured as an assembly line for constructing musical blocks. Each of the producers along the pipeline fills in the elements its generators create, until the finished block is eventually passed to the OutputProducer for playback. The producers all work independently, which means that there is potential to parallelize the generation process. Furthermore, to dynamically alter a composition, a different generator can be substituted in a producer without affecting the rest of the pipeline.

3.4. Emotion Mapper: Mood Based Variations to Music

A key feature of AMEE™ is the incorporation of Mood as a factor which can affect music generation. Mood is considered in two contexts: individual Musicians have a Mood which can be adjusted independently of other Musicians, and a piece can have a Mood as well. Imagine an orchestra with 27 members. Suppose that the bassoon player is depressed because she just learned that she can no longer afford her car payments on her musician's salary. Suppose that the orchestra conductor is trying to achieve a "happy" sound at the end of the piece currently being played. Depending on how professional the bassoon player is, she will play in a way which reflects her own "sad" mood, and the desired "happy" mood to varying degrees. These are the two contexts in which we have considered musical mood.

The StandardEmotionMapper is a simple implementation of a class which makes adjustments to musical elements based only on the emotions Happy and Sad. The StandardEmotionMapper has methods which adjust the Mode, MotifPattern (pitch) and tempo. The logic behind the emotion-based changes is based on research in music psychology. A summary of some papers and ideas which was done for this project can be found in [5]. The adjective descriptors used in the mood class are those defined by Kate Hevner as her well-known Adjective Circle [4].

Presently, all mood adjustments are made based on the Mood characteristics of the first musician added to the group. All other musicians have Mood as well, but it is ignored at this time. The changes necessary to use Moods from all musicians are minimal, however there are a few questions which need to be resolved regarding interactions between global (piece based) mood and local (musician based) mood. For example, Mood can affect tempo of the piece. Should tempo be constant for all musicians? What would happen if musicians could play at different tempos depending on their moods? Our current implementation shows that interesting results can be achieved by altering mood, but we have envisioned many other possibilities which have yet to be fully explored.

4. Implementation Features

This section provides some details about the implementation of AMEE™. Extensive planning was done in the design phase such that the engine is easily extensible, and in order to allow additional features to be added. Hence, the current capabilities of the prototype only reflect a proof-of-concept of the engine's functionality, and by no means define its limitations.

4.1. The MotifGenerator

The MotifGenerator, although it is only one small component in AMEE™, contains the code which one would normally think of as the main element of a music generation system: it generates sequences of notes and rests with associated timing. One very important design decision differentiates AMEE™ from most other music generation systems: the notes (Motifs) that are generated are *independent of both the mode and the harmony*. This is best illustrated by example.

Consider the opening phrase of Mozart's Sonata in A+, K.331:

Figure 2: Mozart Sonata in A+, K.331

The right hand melody in the first bar begins on C#, the third of the scale, and is played over the tonic chord in root position in the bass (harmony I). In the second bar, in the left hand part, we see the exact same melodic pattern, this time starting on G#, played over the dominant chord in first inversion (harmony V_6).

In a motif, we would encode this musical idea as

Pitches:	2	3	2	4	4
Locations:	0.0	1.5	2.0	3.0	5.0
Durations:	1.5	0.5	1.0	2.0	1.0

where pitches are positions in the mode relative to the root of the harmonic chord (with the root as 0), the locations indicate an offset from the beginning of the bar (location 0.0), and the duration specifies the length of the note. Locations and durations are expressed in terms of number of beats (in 6/8 time, an eighth note gets one beat).

The purpose of encoding motifs in this way is to capture the musical pattern associated with the sequence of notes, without restricting ourselves to a specific mode or harmonic chord. The approach we have chosen for motif storage allows composed pieces and musical lines to be transposed and reinterpreted in any mode. Moreover, as illustrated in the above example, a particular pattern of notes often appears more than once during a piece, but serving a different function depending on the underlying harmony.

The StandardMotifGenerator in the prototype operates in both the library-based and pseudo-random manner. When a motif is requested, it first checks whether a library has been loaded. If it has, it attempts to retrieve a motif of the specified length (number of beats), and the desired type (end pattern, which is one that could be found at the end of a section, or regular pattern, or either). If any suitable patterns are found, they are returned. Otherwise, a new pattern will be generated.

Two pseudo-random generators are used in motif creation: a number generator and a pitch generator. A loop continues generating notes/rests as long as the desired number of bars has not been filled. The motifs are generated according to musically plausible rules and probabilities.

As previously mentioned, the motif is encoded in a mode-independent and harmony-independent manner. Thus, the "pitches" that are generated and stored are actually relative pitches to the tonic note in the mode. Consider the following concrete example: Suppose the motif contains the pitches values 0 – 1 – 0. If that motif is eventually resolved in a position where it appears as part of the I chord in C+, would be resolved to the actual MIDI pitches for C – D – C. However, if that same motif were used in a position where the harmony required the V chord in C+, the motif would now be resolved to G – A – G. Suppose now that the motif contains the pitch values 0 – 0.5 – 1 – 0. The value "0.5" indicates that a note between the first and second tone should be sounded, if it exists (this will produce a dissonance). Thus, in C+ for chord I, 0 – 0.5 – 1 – 0 would be resolved as C – C# – D – C. If an attempt is made to resolve a dissonant note where one does not exist (for example, between E and F in C+), one of the neighbouring notes is selected instead.

4.2. Collaboration Between Musicians

The purpose of generating a harmonic structure for a piece is to permit groups of musicians to perform a piece together which will sound musically coordinated. The LineProducer is the entity in the pipeline which is responsible for resolving motifs into notes that fit into chords within a mode. When multiple Musicians are playing together in an Ensemble, the harmonic structure of the block being generated is determined (a HarmonicPattern is chosen), and then each Musician's line is generated *based on this same HarmonicPattern*. The result is that even though each musician is playing a different line from the others (and possibly different instruments, each with its own range), at any given time each musician will be playing a motif which fits into the current harmonic chord and mode for the piece. Of course, this does not imply that every note each musician plays will be consonant with all other notes sounding

at that time – that would be musically uninteresting. Musicians might be playing passing tones, ornaments, dissonant notes, and so on, but the harmonic analysis of each of their musical lines will be the same.

4.3. Choice of Mode

Music can be generated in AMEE™ in *any* mode which can be supported by the underlying MIDI technology. This is a very flexible implementation which allows music played to be played in any major or minor key, or using a whole-tone scale, chromatic scale, blues scale, Japanese scale etc. The restrictions imposed by MIDI on the scale are that the period of repetition for the scale is an octave, and that the smallest distance between two notes is a semi-tone. Thus, an octave is divided into 12 semi-tones, designated by consecutive whole numbers in the MIDI format (i.e. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11). Note that these are "normal" restrictions for Western music. The pattern of the scale is stored using offsets from the starting note of the scale. These offsets specify the number of semi-tones between steps in the scale. For example, every major scale has the following step pattern: 0, 2, 4, 5, 7, 9, 11. To specify which major scale is desired, we also store the starting note of the scale, which is a whole number between 0 and 11 which corresponds to an offset from MIDI pitch 0, which represents C.

Motifs can be resolved into any mode, regardless of the mode in which they were originally specified. An unavoidable consequence of this flexible implementation is that the motifs might sound strange if the mode into which they are resolved contains less tones than the mode in which they were designed. The resolution method which is used in this case is that the notes are "wrapped around" using a modulus-style computation (i.e., the sixth note in a five note mode becomes the first note one octave higher).

4.4. Flexibility of Implementation

A key consideration in the design and implementation of AMEE™ was to provide maximal flexibility for additions, improvements and extensions. We achieved this by implementing all the key classes as Abstract classes which can be subclassed according to the goals of the programmer. As an example, consider the Abstract class MotifGenerator. We have provided a simple concrete implementation of the MotifGenerator in the class StandardMotifGenerator. Now suppose that a developer wishes to have a MotifGenerator which is designed to create jazz style motifs. This would be accomplished by writing a class, JazzMotifGenerator, which extends MotifGenerator, and provides implementations of all the abstract methods. Once this was done, the rest of AMEE™ can be used without any changes.

All the other principal classes in AMEE™ follow this same pattern, and thus the whole system is easily extensible and enhanced.

A further area in which we have attempted to maximize flexibility is in the data structures for the musical elements. As previously mentioned, the mode can be anything which is MIDI supported. The piece length can be user defined or determined by AMEE™. Any MIDI available instruments can be used. Motifs can be as long or as short as the user desires. Any number of PatternLibraries can be developed and used. Any harmonic chords can be defined, which include any number of notes. Musical styles are completely user defined. And so on...

5. Future Development and Open Questions

5.1. Checkpointing

All the pseudo-random number generators are completely deterministic. This means that if generators are initialized with the same seed values during two different runs, the music produced will be exactly the same. We plan to exploit this characteristic in order to be able to checkpoint the pipeline during music generation so that musical elements can be saved and repeated.

5.2. Dynamic Alterations to Music

Eventually, we would like to be able to alter the music which is playing "on the fly." This would permit us, for example, to allow the music generated in a video game to change depending on the player's interactions within the game. Alterations to the music could occur because of a change in mood, addition or removal of a musician, or direct request from a user.

We designed AMEE™ such that one block (a small number of bars) is generated at a time and then output to MIDI. It is therefore possible to make changes between blocks to the music being produced. The difficulty in doing this is that music generation occurs much more quickly than playback, and thus, all the blocks are generated while the first one is playing. To support dynamic alterations, we need a means of keeping track of which block is currently being played and we need to be able to alter or replace subsequent blocks when a change is made.

Furthermore, we might want alterations to occur gradually rather than abruptly, or a mixture of the two. For a gradual change, we would need to know the parameters for starting and end points of the transition, and generate blocks in between accordingly. This gives rise to an additional checkpointing problem since we need to keep track of present and future parameters.

5.3. Better Generator Implementations

To improve and vary the music produced by AMEE™, it will be necessary to implement better generators. Our current prototype only represents a proof of concept; in the next steps of development we are looking to collaborate with musicians to produce different generators for various styles, and to extend the functionality which currently exists.

A few examples of improvements we have planned are the following. (This is by no means an exhaustive list!)

(a) For harmonic generators, we would like to consider transitions between blocks of music – if the previous block ended with an imperfect cadence to chord V, what should come next?
(b) Develop MotifGenerators for various styles of music
(c) Allow generated motifs to be stored in the pattern libraries to facilitate repetition and variations
(d) Some additional musical aspects which we would like to consider are: staccato vs. legato playing; better motifs for endings; motifs typical on different instruments (piano vs. violin vs. guitar vs. electronic etc.)

5.4. Extensions to Mood Implementation

Presently, although our implementation of the Mood class contains 66 emotional descriptors, the StandardEmotionMapper only alters music based on the emotions Happy and Sad. We

need to do more research to determine which emotions we want to be able to vary, and how those emotions will be translated into changes to musical elements. Over the past 70 years, there have been many publications concerning emotional expression in music which we can draw on, but we might also need to do some empirical research in order to properly implement a full EmotionMapper. There is evidence that people only perceive certain emotions in music, but not others [8]. Maybe we will find that some terms collapse into others (perhaps Merry and Joyous are exactly the same in terms of musical expression?) All of these questions require more theoretical investigation.

5.5. Extensions to PatternLibraries

At present, only one of each type of pattern library can be used by the generators. We would like to make it possible for more than one library to be loaded into a generator, especially in the case of the MotifPatternLibrary. This would facilitate combining resources from more than one style in the same piece. Also, it would allow the user to try library combinations without having to make any changes to the libraries themselves. There would be some decisions to be made concerning how patterns are chosen if there are multiple libraries – are any patterns of the desired length returned, or only from one library? Are all the libraries accessed equally often?

We would also like all musicians to be able to use their own libraries. The main difficulty in using a different library for every musician is that it will necessitate storing and copying large amounts of data every time the musician is changed. On a PC this is not a problem, but in future it might be a concern if we try to implement AMEE™ on cell phones or PDAs.

In addition, we would like to be able to add to pattern libraries during composition. This would be simple to implement, and would facilitate repetition in a piece, and also reuse of motifs in future compositions. The only difficulty would be in deciding which motifs to store – for a 3 minute song with 4 musicians, a typical number of generated motifs would be around 200, some of which we might not want to save. Also, if the same library were used many times, it would start accumulating an unreasonably large number of patterns.

5.6. Jam Sessions and Ensembles

We designed AMEE™ such that musicians could play together in two ways: as a coordinated ensemble, which performs a piece with a harmonic structure followed by all the musicians, and as musicians jamming, in which case there is no decided upon harmonic structure, and all the musicians rely on their own musical knowledge to decide what to play. Currently, only the ensembles are implemented, although it is possible to generate a piece with no harmonic structure.

5.7. User Interface

For testing and demonstration purposes, we will be adding a graphical user interface which will allow the functionality of the software to be easily accessed.

6. Potential Applications

As mentioned in the Introduction, the versatile functionality in AMEE™ allows it to be used as an embedded component in a larger software product such as a video game, or to form the basis for a stand-alone music composition application. In this section we mention two other potential applications.

6.1. Emotional Equalizer

A Mood is a collection of emotional descriptors, each present to a different degree. Imagine an "emotional equalizer" which would allow a listener to alter the music's mood as it is playing. This could be either a hardware or a software device which would operate exactly like a normal stereo equalizer, except that the sliders would be marked with emotional descriptors rather than frequency ranges. So, while listening, rather than turning up the bass in the song, you could turn up the "happy."

6.2. Long Distance Musical Collaboration

An application for AMEE™ which is particularly relevant in Canada where people are spread over long distances, would Internet based musical collaboration. Picture three users, one in Iqaluit, one in Vancouver and one in Halifax, each of whom has a collection of Musicians with different qualities and abilities. Those Musicians could perform together and the results could be heard by all the users.

7. Conclusions

The AMEE™ prototype described in this paper is a promising first step toward a dynamic music composition system. We are continuing development based on the extension ideas presented above, and we anticipate using AMEE™ in several exciting application areas in the near future.

Acknowledgments

This research was funded in part by CITO/OCE and by Condition30 Inc.

References

[1] Charles Ames, *Automated composition in Retrospect*, Leonardo, 20(2), 169-185 (1987)

[2] Pietro Casella and Ana Paiva, *MAgentA: an architecture for real time automatic composition of background music*, Intelligent Virtual Agents, LNCS 2190, 224-232 (2001)

[3] Ryan J. Demopoulos, *Towards an Integrated Automatic Music Composition Framework*, MSc Thesis, Department of Computer Science, University of Western Ontario (2007)

[4] Kate Hevner, *Experimental Studies of the Elements of Expression in Music,* The American Journal of Psychology, 48(2), 246-268 (1936)

[5] Maia Hoeberechts, *API Design: Summary of Progress to Date*, Internal progress report, University of Western Ontario (2005)

[6] Steven R. Livingstone, Ralf Mühlberger, Andrew R. Brown and Andrew Loch, *Controlling musical emotionality: an affective computational architecture for influencing musical emotions*, Digital Creativity, 18(1), 43-53 (2007)

[7] Steven M. Pierce, *Experimental Frameworks for Algorithmic Film Scores*, MA Thesis, Dartmouth College (2004)

[8] Mark Meerum Terwogt and Flora Van Grinsven, *Musical Expressions of Moodstates,* Psychology of Music, 19, 99-109 (1991)

[9] Dynamic Object Music Engine (DOME), http://www.dometechnics.com, accessed August 23, 2007.