MTT – A Music Theory Tool

Alan Marsden, School of Music, Queen's University, Belfast, BT7 1NN, UK, (currently visiting researcher at Infomus, DIST, University of Genoa, Italy) A.Marsden@qub.ac.uk

***Abstract**: A system, currently under development, for expressing musical processes in precise terms of formal logic, and for allowing that formulation to be tested on actual data, is described. The system is based on a theory of representation and of processing as the conversion of data from one representation to another. Examples are given of formal definitions of a representation for melodies and keys, and three examples of translations from one to the other are given, two of them based on Krumhansl's and Longuet-Higgins's key-finding algorithms. It is intended that MTT should be used in the development, testing and refinement of theories of musical processes prior to implementing those processes in software.*

1. Introduction

Good music software depends on good music theory: if one is to write software to perform some musical task successfully, one has to have an understanding of the task itself. Music theory is a highly developed science, but it is not generally immediately applicable in computing tasks. It is generally expressed in varying degrees of mixture of "natural" language, specialised terminology, and specialised notation (e.g. Schenkerian graphs). In most cases, even with specialised terminology or notation, the meaning of statements in music theory is highly dependent on complex considerations of context, and is sometimes ambiguous. As is commonly recognised, the first task facing a computer programmer who hopes to use music theory, then, is to re-express the music theory in a formal, unambiguous, and less context-dependent way.

However, this formalisation frequently reveals that the original music theory is not precise enough for the task at hand. Either the theory turns out to be impossible to formalise without taking decisions to resolve ambiguities – decisions which adapt the theory – or the theory turns out not to adequately account for the phenomena under consideration. To take a concrete example, a task which must be performed for a number of musical processes concerned with tonal music is to determine the key of a passage of music. There is a large body of music theory on this topic, but, even after thirty years of research, there is still not a computer implementation of the task which performs adequately in situations other than highly constrained and specialised ones. (For a survey of research in this area and discussion of the issues, see [Marsden in press].) One of the earliest projects was an expression of tonal theory in a systemic grammar [Winograd 1968], but Winograd found that a complex parser, which used a number of ad hoc concepts not found in classical music theory, was required in order to arrive at a satisfactory analysis. Furthermore, the parser worked from the end of the music to the beginning, and so would not be applicable in contexts where the end of the piece was not yet known but the key of the beginning needed to be known. Subsequent projects had similar limited success in limited domains, using a number of different approaches. Longuet-Higgins [1979] devised a set of rules which determined key on the basis of the sequence of pitches used, altering its interpretation with the objective of avoiding sequences of "chromatic" intervals. The rules applied only to melodies. Krumhansl [1990] proposed a statistical approach which had some points of contact with traditional theory (in the concept of the hierarchy of importance of different pitches in a key) but which ignored all concepts of harmony.

Music theory generally was not created in order to write computer programs, but for the purposes of teaching. A theory which is easily understood but not correct at the margins is better for teaching than a theory which is always correct but very difficult to understand. It is therefore no surprise to discover that music theories are not completely precise. Furthermore, there is no reason to believe that in every case a completely precise theory is possible. The musical phenomena with which the theory is to deal are not fixed mathematical or physical phenomena. They are subject to adaptation, change and reinterpretation in a complex social interaction. The computer programmer, therefore, must make compromises and seek the best expression of theory for the task his or her software is to perform.

The objective of MTT is to facilitate this development of a theory which is formally expressed and adequate for the task at hand. It is not uncommon to find that the inadequacies of musical software have arisen because of an incorrect decision in the interpretation of some aspect of music theory. Early rhythmic quantisation software, for example, produced unusable results because of a naive interpretation of the theory of metre, believing that beats would always be of the same duration. Often, by the time the inadequacy is discovered, the software is in a relatively developed state, and the incorrect interpretation is embedded in the software design and very difficult to correct. The principle of MTT is that it allows the underlying theory to be expressed in formal terms at a very high level where the theoretical concepts can still be manipulated as concepts. On the other hand, the system will implement the theory in functioning software, though probably extremely inefficiently, so that the formal theory can be tested by application to actual tasks. MTT is thus a kind of "rapid prototyping" system. Recall that the objective is unlikely to be a theory which is correct in every circumstance but rather one which is adequate for the task at hand. Thus the researcher might well choose to accept a theory which made mistakes at times, but which was simpler in its formation and/or easier to implement in efficient software.

MTT is essentially a kind of automatic theorem-proover, under development in Prolog. Theories are expressed in first-order predicate logic, within a framework described below. "Input data" may be asserted. "Output data" is derived by posing queries to the system, which are treated as statements to be satisfied or refuted. Therefore, given a theory of tonality expressed in a set of axioms and a passage of music asserted as input, one could pose a query such as "key(C, major)", and the system would attempt to determine whether or not this statement was valid in the theory augmented by the input. One could also pose the query "key(X, Y)", in which X and Y were variables, and receive as response from the system the answer that this is valid when "X" is "C" and "Y" is "major", meaning that the key of the passage is C major. The axioms are not restricted to Horn clauses as in Prolog; the objective is for axioms to be a close to general first-order logic as reasonably possible. However, the language of theories cannot have functions, and there is no equality relationship (apart from identity). Resolution is sound, i.e. the system's answer, whether true or false, is always valid, and complete when the domain is finite. When the domain is infinite, the objective is that the system should report its inability to reach a valid conclusion ("floundering") or warn of the possibility of failure to terminate and allow the user to intervene.

2. The theoretical framework

2.1 Representation

All computer software deals with representations of its data. A representation consists of a set of entities, which are intended to correspond to things in the world represented. These entities stand in specified relations to each other, and possibly to constants also. The constants do not correspond to anything in the world represented, but are items specific to the class of representation and are used in relations to specify the properties of an entity or of a relation between entities. Each representation class has a specified set of constants, which might be finite or infinite. It also has a specified set of relations, each defined as ranging over either entities or constants, but not both, in each of its arguments. A representation class also has a set of axioms defining constraints on the relations of the representation. A "representation function" specifies the relationship between the entities and relations of the representation and the world represented. This function is not expressed in the theory. In practice it will be performed by some input or output device.

MTT representations are expressed explicitly as described above, but the description is also intended as an abstraction of representations in general. A representation of music in one of the Humdrum formats [Huron 1994], for example, is a set of data tokens (entities) which can contain certain combinations of characters (constants). The tokens are related by their position in lines and columns in a way constrained by the syntax of the representation. At a lower level, the representation is as an ASCII file, which again consists of entities (bytes) which contain data from a certain set (ASCII codes), and are related by a simple relationship of succession. Even non-computational representations can be regarded in this way. Music notation, for example, consists of symbols (entities) which are of certain types (constants) in certain spatial relationships to each other.

As a simple example of a representation class, here is a class for representing a melody (a sequence of pitches). The symbols and syntax used in MTT are not exactly those of classical logic, but they are intended to be close to classical logic while also close to classical Prolog. For ease of understanding, classical logical symbols are used here: ∀ for "all", ∃ for "there exists", ↔ for "is equivalent to", → for "implies", ∧ for "and", ∨ for "or", and ~ for "not". Comments, which do not form part of the representation class definition, are enclosed between "/*" and "*/".

Melody representation class
<u>Constants</u>
pitch_class: modulo(12)
register: integer
duration: rational
<u>Relation types</u>
precedes(entity, entity) /* one entity can precede another */
first_note(entity)
last_note(entity)
duration(entity, rational) /* entities can be notes or rests and have duration */
pitch_class(entity, pitch_class)
register(entity, integer) /* relations to indicate the properties of a note */
<u>Axioms</u>
precedes(X, Y) ∧ precedes(X, Z) → Y=Z.
precedes(X, Y) ∧ precedes(Z, Y) → X=Z.
/* precedes is constrained to be a one-to-one relation */
(∃X)first_note(X).
first_note(X) ↔ ~(∃Y)precedes(Y, X) ∧ ~(∃Z)(~(∃Y)precedes(Y, Z) ∧ ~Z=X).
/* there exists a first note and it is not preceded by any other entity */
last_note(X) ↔ ~(∃Y)precedes(X, Y).
/* the last note is the note which does not precede any other entity; we do not need to specify that it is unique because this follows from the axioms for the first note and for the "precedes" relation */
(∃D)duration(X, D).
duration(X, D1) ∧ duration(X, D2) → D1=D2.
(∃Pc)(pitch_class(X, Pc).
pitch_class(X, Pc1) ∧ pitch_class(X, Pc2) → Pc1=Pc2.
(∃R)(register(X, R).
register(X, R1) ∧ register(X, R2) → R1=R2.
/* an entity must have a duration, a pitch class and a register, and it can have only one pitch class, one register, and one duration */

2.2 Combining representations

Processing tasks can be defined as taking one representation as input and producing another as output. Effectively, this is mimicking the relationship between a representation and the world represented. Thus the "output" desired is a kind of representation of the "input" in a different manner. Examples of common musical tasks are taking a representation of a "score" for a synthesis system and producing a sound file, or taking MIDI input and producing music notation. Although we would not necessarily use the word "representation" to describe what is happening in the first case, the formal relationship between the input and output is similar.

In contrast to the relationship between the world and a representation, however, the relationship between two representations can (sometimes, at least) be represented in the formal language of the system. In logical terms, a representation class is a theory, with the world it represents as one of its models. To define the relationship between two representation classes, we make a single combined theory – a single combined representation class – which includes the constants, relation types and axioms of both theories, modified if necessary to avoid clashes of names. The combined theory can also include additional constants and relation types, and it certainly includes additional axioms which define the relationship between the two representations.

As an example, here is a very simple representation class for a key, followed by a combined representation class which naively assumes that a piece is in a particular key if it uses only the notes of the scale of that key (including the ascending and descending melodic forms for minor keys).

Key representation class
Constants
pitch_class: modulo(12)
mode: {major, minor}
Relation types
tonic(pitch_class)
mode(mode) /* a key is expressed by its tonic pitch class and its mode */
Axioms
tonic(X) $\wedge$ tonic(Y) $\rightarrow$ X=Y.
mode(X) $\wedge$ mode(Y) $\rightarrow$ X=Y. /* there can be only one tonic and only one mode */

Naive melody/key combined representation class
Uses class **melody**
Uses class **key**
Uses class **modulo(12)**
Axioms
tonic(T) $\wedge$ mode(major) $\rightarrow$ ~($\exists$X)(pitch_class(X, P) $\wedge$ (sub(P, T, 1) $\vee$ sub(P, T, 3) $\vee$ sub(P, T, 6) $\vee$ sub(P, T, 8) $\vee$ sub(P, T, 10)).
/* if the mode is major, there can be no notes 1, 3, 6, 8 or 10 semitones above the tonic */
tonic(T) $\wedge$ mode(minor) $\rightarrow$ ~($\exists$X)(pitch_class(X, P) $\wedge$ (sub(P, T, 1) $\vee$ sub(P, T, 4) $\vee$ sub(P, T, 6)).
/* if the mode is minor, there can be no notes 1, 4, or 6 semitones above the tonic */

The first three lines of the combined representation class definition state the classes which are combined, and effectively include their definitions in the combined theory. This combined class also makes use of a theory not defined here which gives access to arithmetic functions for integers modulo 12. Note that the two axioms are implications from the key to the melody and not the other way around. This means that a particular passage of music does not necessarily have a key. MTT is intended to implement a kind of modal logic, which includes concepts of possibility and necessity, and a "possible worlds" metaphor. "Necessity" means that something is definitely true or false. Something is "possibly" true if it is not necessarily false. In the case of this combined theory, we can ask if a particular key is a possible interpretation of the key of the melody. Since the set of keys is finite, we can also ask if the key is the only possible interpretation. At the present stage of design, possibility and necessity can be handled in top-level queries, but mechanisms for incorporating possibility into axioms have yet to be designed.

3. Examples

3.1 Krumhansl's key-finding algorithm

As a more sophisticated example, here is an example of a combined representation class which presents Krumhansl's key-finding algorithm in formal logical terms. The essence of the algorithm is to determine the pitch-duration-profile for a passage of music, i.e. to find the sum of the durations of all notes of each pitch class, and to find the "ideal pitch profile" which most closely matches this. (The details of the arithmetic of computing the correlation between the observed pitch profile and the ideal profiles for each key are not presented here.) A particular tonic and mode will be defined except in the case that there is more than one pitch profile with exactly the same correlation.

Krumhansl melody/key combined representation class
Uses class **melody**
Uses class **key**

Uses class **sets**

Uses class **real numbers**

Constants

correlation_index: real

pitch_profile: ⟨rational, rational, rational, rational, rational, rational, rational, rational, rational, rational, rational, rational⟩

duration_set: set(duration)

Relation types

set_of_note_durations(pitch_class, duration_set)

/* the set of duples of ⟨note, duration⟩ of notes with a particular pitch class */

total_duration(pitch_class, rational)

/* the sum of all the durations of notes with a particular pitch class */

pitch_profile(pitch_profile)

/* the profile of durations for each pitch class */

C_major_pitch_profile(pitch_profile)

C_minor_pitch_profile(pitch_profile)

/* the ideal pitch profiles for C major and C minor */

profile_rotation(pitch_profile, pitch_class, pitch_profile)

/* the last argument is the pitch profile found in the first argument rotated so that if the first argument began at C, the last argument begins at the pitch class in the second argument */

profile_correlation(pitch_profile, pitch_profile, correlation_index)

/* the last argument is the degree of correlation between the two profiles */

Axioms

C_major_pitch_profile(6.35, 2.23, 3.48, 2.33, 4.38, 4.09, 2.52, 5.19, 2.39, 3.66, 2.29, 2.88).

C_minor_pitch_profile(6.33, 2.68, 3.52, 5.38, 2.60, 3.53, 2.54, 4.75, 3.98, 2.69, 3.34, 3.17).

/* data from [Krumhansl & Kessler 1982] */

(∃S)set_of_note_durations(P, S).

/* there is a set of note/durations for every pitch class */

pitch_class(X, P) ∧ duration(X, D) ∧ set_of_note_durations(P, S) ↔ ⟨X, D⟩∈ S.

/* There is a member of the set of note/durations for pitch class P for every note with that pitch class. The member is a duple of the note entity and its duration. */

set_of_note_durations(P, S) ∧ sum_of_second_elements(S, T) ↔ total_duration(P, T).

/* the total duration for a pitch class is the sum of the second elements of the tuples in the corresponding set of note/durations */

total_duration(0, D0) ∧ total_duration(1, D1) ∧ total_duration(2, D2) ∧ total_duration(3, D3) ∧ total_duration(4, D4) ∧ total_duration(5, D5) ∧ total_duration(6, D6) ∧ total_duration(7, D7) ∧ total_duration(8, D8) ∧ total_duration(9, D9) ∧ total_duration(10, D10) ∧ total_duration(11, D11) ↔ pitch_profile(⟨D0, D1, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11⟩).

pitch_profile(PP) ∧ C_major_pitch_profile(CMPP) ∧ C_minor_pitch_profile(CmPP) ∧ pitch_profile_rotation(PP0, P1, PP1) ∧ pitch_profile_correlation(CMPP, PP1, C1) ∧ ~(∃P2)(pitch_profile_rotation(PP0, P2, PP2) ∧ pitch_profile_correlation(CMPP, PP2, C2) ∧ C2≥C1) ∧ ~(∃P2)(pitch_profile_rotation(PP0, P2, PP2) ∧ pitch_profile_correlation(CmPP, PP2, C2) ∧ C2≥C1) → tonic(P1) ∧ mode(major).

pitch_profile(PP) ∧ C_major_pitch_profile(CMPP) ∧ C_minor_pitch_profile(CmPP) ∧ pitch_profile_rotation(PP0, P1, PP1) ∧ pitch_profile_correlation(CmPP, PP1, C1) ∧ ~(∃P2)(pitch_profile_rotation(PP0, P2, PP2) ∧ pitch_profile_correlation(CmPP, PP2, C2) ∧ C2≥C1) ∧ ~(∃P2)(pitch_profile_rotation(PP0, P2, PP2) ∧ pitch_profile_correlation(CMPP, PP2, C2) ∧ C2≥C1) → tonic(P1) ∧ mode(minor).

/* if the correlation between some rotation of the pitch profile with the ideal profile for C major (or C minor) is greater for this rotation than for any other and greater than the correlation with the ideal profile for C minor (or respectively C major) for any rotation, then the tonic is the pitch which produced this rotation and the mode is major (or respectively minor) */

3.2 Longuet-Higgins's key-finding algorithm

As a second example, a formal logical expression of the algorithm described by Longuet-Higgins [1979] is given. This operates on a principle of starting on the assumption that the first note is the tonic or dominant, and then adjusting the interpretation of key so as to avoid sequences of two or more intervals which are chromatic in the current key interpretation (e.g. C to C sharp in C major). The mechanism for determining whether the mode is major or minor is not included. The algorithm was really intended to track the shifting key of a piece rather than find a particular key, but in order to make possible comparisons with Krumhansl's algorithm, a final axiom not derived from Longuet-Higgins is included which identifies the key of the piece as the interpretation at the last note, provided all the previous notes also meet the rules with respect to chromatic intervals in this interpretation.

Longuet-Higgins melody/key combined representation class
Uses class **melody**
Uses class **key**
Uses class **integers**
Uses class **modulo(12)**
<u>Relation types</u>
pc_sharpness_in_key(pitch_class, pitch_class, integer)
/* the "sharpness" of the a pitch_class in the context of the "extended key" of the second argument */
tonic_at_note(entity, pitch_class)
/* the tentative interpretation of key up to the note of the first argument */
chromatic(modulo(12))
/* identifies intervals which are "chromatic" by Longuet-Higgins definition */
<u>Axioms</u>
pc_sharpness_in_key(P, K, S) ↔ modulo(12):add(K, 1, K1) ∧ modulo(12):sub(P, K1, I) ∧ modulo(12):mult(I, 7, S).
/* The "sharpness" is the number of fifth-steps up from the note a semitone above the tonic. (This gives a range 0–12. Longuet-Higgins actually defined it slightly differently with a range of -5–6, but the effect is the same.) */
chromatic(X) ↔ X>6 ∨ X<-6.
first_note(X) → tonic_at_note(X, X).
precedes(X, Y) ∧ first_note(X) ∧ (pc_sharpness_in_key(Y, X, 0) ∨ pc_sharpness_in_key(Y, X, 1) ∨ pc_sharpness_in_key(Y, X, 4) ∨ pc_sharpness_in_key(Y, X, 8)) ∧ integer:sub(X, 7, T) → tonic_at_note(Y, T).
precedes(X, Y) ∧ first_note(X) ∧ ~pc_sharpness_in_key(Y, X, 0) ∧ ~pc_sharpness_in_key(Y, X, 1) ∧ ~pc_sharpness_in_key(Y, X, 4) ∧ ~pc_sharpness_in_key(Y, X, 8)) → tonic_at_note(Y, X).
/* These two axioms reflect Longuet-Higgins's rule about whether the first note should be interpreted as tonic or dominant. */
precedes(X, Y) ∧ precedes(Y, Z) ∧ tonic_at_note(Y, Ty) ∧ pc_sharpness_in_key(X, Ty, Sx) ∧ pc_sharpness_in_key(Y, Ty, Sy) ∧ pc_sharpness_in_key(Z, Ty, Sz) ∧ integer:sub(Sy, Sx, Ixy) ∧ integer:sub(Sz, Sy, Iyz) ∧ chromatic(Ixy) ∧ chromatic(Iyz) ∧ (Sy<4 ∧ add(Sy, 1, Sy1) ∨ Sy=12 ∧ Sy1=-1) ∧ modulo(12):mult(Sy1, 7, Tadj) ∧ modulo(12):add(Ty, Tadj, Tz) → tonic_at_note(Z, Tz).
precedes(W, X) ∧ precedes(X, Y) ∧ precedes(Y, Z) ∧ tonic_at_note(Y, Ty) ∧ pc_sharpness_in_key(W, Ty, Sw) ∧ pc_sharpness_in_key(X, Ty, Sx) ∧ pc_sharpness_in_key(Y, Ty, Sy) ∧ pc_sharpness_in_key(Z, Ty, Sz) ∧ integer:sub(Sy, Sw, Iwy) ∧ integer:sub(Sy, Sx, Ixy) ∧ integer:sub(Sz, Sx, Ixz) ∧ chromatic(Iwy) ∧ chromatic(Ixy) ∧ chromatic(Ixz) ∧ (Sx<4 ∧ integer:add(Sx, 1, Sx1) ∨ Sx=12 ∧ Sx1=-1) ∧ modulo(12):mult(Sx1, 7, Tadj) ∧ modulo(12):add(Ty, Tadj, Tz) → tonic_at_note(Z, Tz).
/* These two axioms reflect Longuet-Higgins's rules about adjusting the current interpretation of the key in the light of sequences of chromatic intervals. */
precedes(X, Y) ∧ precedes(Y, Z) ∧ first_note(X) ∧ tonic_at_note(Y, Ty) ∧ pc_sharpness_in_key(X, Ty, Sx) ∧ pc_sharpness_in_key(Y, Ty, Sy) ∧ pc_sharpness_in_key(Z, Ty, Sz) ∧ integer:sub(Sy, Sx, Ixy) ∧ integer:sub(Sz, Sy, Iyz) ∧ (~chromatic(Ixy) ∨ ~chromatic(Iyz)) → tonic_at_note(Z, Ty).
precedes(W, X) ∧ precedes(X, Y) ∧ precedes(Y, Z) ∧ tonic_at_note(Y, Ty) ∧ pc_sharpness_in_key(W, Ty, Sw) ∧ pc_sharpness_in_key(X, Ty, Sx) ∧ pc_sharpness_in_key(Y, Ty, Sy) ∧

pc_sharpness_in_key(Z, Ty, Sz) ∧ integer:sub(Sy, Sw, Iwy) ∧ integer:sub(Sy, Sx, Ixy) ∧ integer:sub(Sz, Sy, Iyz) ∧ integer:sub(Sz, Sx, Ixz) ∧ (~chromatic(Ixy) ∨ ~chromatic(Iyz)) ∧ (~chromatic(Iwy) ∨ ~chromatic(Ixy) ∨ ~chromatic(Ixz) → tonic_at_note(Z, Ty).

/* These two axioms reflect the implicit part of Longuet-Higgins's rules about chromatic intervals in that the tonal interpretation remains unchanged if the rules do not apply. */

last_note(L) ∧ tonic_at_note(L, T) ∧ ~(∃W)(∃X)(∃Y)(∃Z)(precedes(W, X) ∧ precedes(X, Y) ∧ precedes(Y, Z) ∧ pc_sharpness_in_key(W, T, Sw) ∧ pc_sharpness_in_key(X, T, Sx) ∧ pc_sharpness_in_key(Y, T, Sy) ∧ pc_sharpness_in_key(Z, T, Sz) ∧ integer:sub(Sx, Sw, Iwx) ∧ integer:sub(Sy, Sw, Iwy) ∧ integer:sub(Sy, Sx, Ixy) ∧ integer:sub(Sz, Sx, Ixz) ∧ (chromatic(Iwx) ∧ chromatic(Ixy) ∨ chromatic(Iwy) ∧ chromatic(Ixy) ∧ chromatic(Ixz)) ↔ tonic(T).

4. Implementation

MTT is being implemented in Prolog, which is a language which implements a kind of theorem-prooving in the restricted domain of Horn clauses. Although efficiency is not the main concern of MTT, in an effort to arrive at a practical tool, normal Prolog resolution (basically SLD resolution with Negation as Failure) will be used wherever logically sound. In other cases, the theorem-proving mechanism employed will be SLI resolution [Das 1992], but without negation as failure. In order to distinguish places in processing where SLDNF resolution can be used from places where it cannot, one or more representation classes must be designated as "input", and certain restrictions must apply to the use of its relations in order to preserve consistency. The "closed-world assumption" will apply to these relations. (This is in keeping with the intention of modelling a computing system; the input devices are considered to give sufficiently complete information about the environment for processing.) This information might enable the closed-world assumption, and SLDNF resolution, to be applied to other relations also. Mechanisms are also planned to recover from loops, and to know when the overheads of this are not required because loops are impossible [Brough & Hogger 1986], and to delay subqueries whose arguments are insufficiently instantiated (e.g. arithmetic queries) [Naish 1985]. Negative information in the case of non-closed relations will be explicitly represented, with an appropriate consistency check.

5. Discussion and future developments

Krumhansl presents a comparison of the performance of her algorithm with other key-finding algorithms [Krumhansl 1990:77–120], including an earlier version of Longuet-Higgins's algorithm, so achieving some of what an implementation like this in MTT would set out to do. However, her results depend on the comparison of her own tests with published results of others. Although she could apply her algorithm to the same music as others had used, she was not able to test others' algorithms on any new data without a reimplementation of their algorithms, which would have been difficult and time-consuming. Furthermore, if another algorithm were found to perform better than her own, it would be difficult to determine what changes were necessary to her algorithm in order for it to perform as the other had done, because the language in which the two were expressed was so different.

This kind of comparison of theories, through the results of their application, is a kind of "model-theoretic" comparison. More "proof-theoretic" comparisons should be possible also, comparing different theories through examination of the structure of their axioms. In this respect these two examples are not particularly enlightening, but it is clear, for example, that Longuet-Higgins's theory is recursive through the definition of axioms for the relation I call 'tonic_at_note', whereas Krumhansl's is not. (As noted, the example is not enlightening, because it was a principle of Longuet-Higgins's procedure that it should base its determination of key on the key of the preceding music.) Formal comparisons directed towards implementation are also possible. The relations of both theories can be demonstrated to correspond to functions in the direction from music to key, for example. Thus an implementation in a funtional manner is indicated to be possible. The long conjunctions in some of the axioms of Krumhansl's theory can be shown to consist of independent terms whose satisfaction does not depend on the satisfaction of the others, suggesting a strategy for parallelisation. This is not the case in the Longuet-Higgins theory.

These kind of comparisons have been little explored, as yet, and are not the main objective of MTT. Instead, MTT is envisaged as being a tool for application in areas where there is only a poorly expressed theoretical model at present, and such a model can only be formed by a combination of formal analysis and application to test cases. Besides tonal theory, other possible areas of application are in formalising the measurement of similarity of melodies, theories of musical segmentation, and music notation (an area less understood than is often assumed). Most realistic applications will be considerably more complex than the cases discussed above. Even a formalisation of more sophisticated tonal theories than those discussed, e.g. Schenkerian theory or simply tonal theory as presented in most harmony text-books for that matter, would be more complex. Since predicate logic allows any relation of the appropriate language to be included in an axiom with any other relation, there is a real problem of potential explosion in the number and complexity of axioms when treating a larger theory. It is intended that the mechanism of representation classes should provide a discipline of segmentation of a theory through modularity. This will probably involve extension of the underlying principles of MTT to incorporate some analogue of the "private" and "public" distinction common in modular and object-oriented approaches, in order to impose the discipline of modularity and to facilitate its exploitation in implementation. Another possible future development is extension of MTT, either in software tools or in software development methodology, to allow a smooth path from formalisation of theory using to MTT to functioning software. In particular, it is intended to allow the integration of externally compiled software with MTT so that parts of a theory can be compiled and still operate with the other parts, or a previously competed implementation of some other theory can be used in an extension or use of that theory.

References

[Brough & Hogger 1986] D.R. Brough & C.J. Hogger, The Treatment of Loops in Logic Programming, Research report DoC 86/16 Dept. of Computing, Imperial College of Science and Technology, London, 1986.

[Das 1992] S.K. Das, *Deductive Databases and Logic Programming*, Addison-Wesley, Wokingham, 1992.

[Huron 1994] D. Huron, *The Humdrum Toolkit; Reference Manual*, Distributed by the *Centre for Computer Assisted Research in the Humanities*, Stanford University, CA, USA, 1994. (Humdrum is available by *ftp* from archive.uwaterloo.ca in directory uw-data/humdrum, or from the Centre for Computer Assisted Research in the Humanities.)

[Krumhansl 1990] C.L. Krumhansl, *Cognitive Foundations of Musical Pitch*, Oxford University Press, New York & Oxford, 1990.

[Krumnhansl & Kessler 1982] C.L. Krumhansl & E.J. Kessler, Tracing the dynamic changes in perceived tonal organization in a spatial representation of musical keys, *Psychological Review*, v.89, 1982, 334–368.

[Longuet-Higgins 1979] H.C. Longuet-Higgins, The perception of music, *Proceedings of the Royal Society, London*, v.205, 1979, 307–322.

[Marsden in press] A. Marsden Computers and the concept of tonality, In J.T. Coppock (ed.) *Information Technology and Scholarly Disciplines*, British Academy, London, in press.

[Naish 1985] L. Naish *Negation and Control in Prolog*, Springer-Verlag, Berlin, 1985.

[Winograd 1968] T. Winograd, Linguistics and the computer analysis of tonal harmony, *Journal of Music Theory*, v.12, 1968, 2-49.