Department of Computer Science and Information Systems

5h Annual Research Conference

Tuesday, 15th September 1998

S2-05 Schuman Building

Session 1 Chaired by Ita Richardson

Michael O Neill	Grammatical Evolution
John Shinnick	Building a Beowulf System
Tony Cahill	Tools to Monitor Programmers' Information Seeking Behaviour
Norah Power	The Agreement Dimension in Requirements Specifications

Session 2 Chaired by Norah Power

Seamus O Shea	Running IP over ATM
Peter McCarthy	Decompilation on a RISC Architecture
David Burns	Evaluating a Multi-Media CAL Package
Paddy Healy	Some Tools for On-line Assessment

Session 3 Chaired by Donncha O Maidin

Richard Sutcliffe	A Semi-Automatic Method for the Detection of False Cognates between English and Polish.
Pat O Sullivan	Runtime Translation of Client/Server HCI
Caolan McNamara	Reactive Hybrid-Ecological Audio Enhanced Interfaces
Ian O Keeffe	An Interactive Composition System

Grammatical Evolution

Michael O'Neill

Department of Computer Science and Information Systems
University of Limerick
Limerick, Ireland

+353-61-202745
michael.oneill@ul.ie
WWW: http://shine.csis.ul.ie

Abstract
A Genetic Algorithm is described that can evolve complete programs. Using a variable length, linear, binary genome to govern the mapping of a Backus Naur Form grammar definition to a program, expressions and programs of arbitrary complexity may be evolved. Our system, Grammatical Evolution, has been applied to problems such as Symbolic Regression, Symbolic Integration, finding Trigonometric Identities, the Santa Fe Trail, and Grammar Induction.
The purpose of this paper is to give a brief overview of the system Grammatical Evolution.

Introduction

Evolutionary Algorithms have been used with much success for the automatic generation of programs. In particular, Koza's Genetic Programming [Koza 92] has enjoyed considerable popularity and widespread use. While Koza originally employed Lisp as his target language many experimenters generate a homegrown language, peculiar to their particular problem. Lisp is the most generally used language for a number of reasons, not least of which is the property of Lisp of not having a distinction between programs and data. Hence, the structures being evolved can directly be evaluated. Furthermore, with reasonable care, it is possible to design a system such that Lisp programs may be safely crossed over with each other and still remain syntactically correct.

Evolutionary Algorithms have been used to generate other languages, by using a grammar to describe the target language. Researchers such as Whigham [Whigham 95] and Wong and Leung's LOGENPRO system [Wong 95] used context free languages in conjunction with GP to evolve code. Both systems exploited GP's use of trees to manipulate parse trees, but LOGENPRO did not explicitly maintain parse trees in the population, and so suffered from some ambiguity when trying to generate a tree from a program. Whigham's work did not suffer from this, and had the added advantage of allowing an implementor to bias [Whigham 96] the search towards parts of the grammar.

Another attempt was that of Horner [Horner 96] who introduced a system called Genetic Programming Kernel (GPK), which, similar to standard GP, employs trees to code genes. Each tree is a derivation tree made up from the BNF definition. However, GPK has been criticised [Paterson] for the difficulty associated with generating the first generation - considerable effort must be put into ensuring that all the trees represent valid sequences, and that none grow without bounds. GPK has not received widespread usage.

An approach, which generates C programs directly, was described by Paterson [Paterson 97]. This method was applied to the area of evolving caching algorithms in C with some success, but with a number of problems, most notably the problem of the chromosomes containing vast amounts of introns.

Grammatical Evolution (GE) can be used to generate programs in any language, using a genetic algorithm to control what production rules are used in a Backus Naur Form grammar definition. Whereas Whigham used a grammar to define the initial population and lets Genetic Programming operate on the resulting parse trees, GE simply encodes a binary string that the Genetic Algorithm operates on. The grammar is used then to map the individual bit strings onto a program when evaluating fitness. Also, because our system evolves binary strings and not the actual program or parse tree, we do not have to design new genetic operators in our Genetic Algorithm to ensure syntactically correct programs.

GE has proved successful when applied to a Symbolic Regression problem [Ryan et al. 98a], finding Trigonometric Identities [Ryan et al. 98b], and a Symbolic Integration problem [Ryan, O'Neill 98a][Ryan, O'Neill 98b]. Although unreported to date, GE has also proved successful for a single case of the Santa Fe Ant Trail, and a Grammar Induction problem. In each of these problems we take a subset of C as our target language which is described in Backus Naur Form.

Grammatical Evolution

Grammatical Evolution codes a set of pseudo random numbers on a chromosome that consists of a variable number of 8 bit binary genes. These numbers are used to select an appropriate rule from a Backus Naur Form grammar definition, an example of which is given below. A BNF grammar consists of the tuple {N, T, P, S}, where N is the set of non-terminals, T, the set of terminals, P, a set of production rules that maps the elements of N to T, and S is a start symbol which is a member of N. The non-terminals of the grammar are then mapped onto the terminals of the grammar by continuously applying the rules dictated by the gene values. On completion of the mapping process the final code produced consists of these terminals.

```
N = { expr, op, pre-op }
T = { Sin, Cos, Tan, Log, +, -, /, *, X, (, )  }
S = <expr>
```

And P can be represented as:

```
(1) <expr> ::= <expr> <op> <expr>      (A)
             | ( <expr> <op> <expr> ) (B)
             | <pre-op> ( <expr> )     (C)
             | <var>                   (D)

(2) <op> ::= + (A)
           | -  (B)
```

```
        | /  (C)
        | * (D)

(3) <pre-op> ::= Sin (A)
      | Cos (B)
      | Tan (C)
      | Log (D)

(4) <var> ::= X
```

Consider rule #1 from the above example. In this case, the non-terminal can produce one of four different results, to decide which one to use our system takes the next available random number from the chromosome and, in this case gets the modulus four of the number to decide which production to take. Each time a decision has to be made, another pseudo random number is read from the chromosome, and in this way, the system traverses the chromosome.

In a manner similar to biological systems [Elseth 95] the genes in GE are ultimately operated on to produce a phenotype, or in our sense, a program. The phenotype is produced as a result of a mapping process which we liken to the process of gene transcription, followed by translation to a protein, see Figure 1.

Gene transcription is the process of converting a strand of DNA into a strand of RNA. The RNA then carries the encoded instructions of the genes to the machinery of the cell which can synthesize proteins. Translation is the mapping of the RNA onto amino acids. Put simply, the RNA is comprised of four different types of bases, groups of three bases, known as a codon, specifying a particular amino acid. The amino acids are thus joined together in the order dictated by the RNA to produce a protein. These proteins can act either independently or in conjunction with other proteins to produce a phenotype. To complete the analogy with GE then, the transcription process would be equivalent to the conversion of the binary string data structure into a string of integers which is brought from the Genetic Algorithm into the machinery of GE. The integers are then used in the translation process, that is the selection of rules in the BNF definition that result in the mapping of non-terminals to terminals. The production rules, equivalent to amino acids, combine to produce terminals, or proteins. It is the proteins that make up the final program.

In GE it is possible for individuals to run out of genes during the mapping process, and in this case there are two alternatives. The first is to declare the individual invalid and punish them with a suitably harsh fitness value; the alternative is to wrap the individual, and reuse the genes. This is quite an unusual approach in EAs, as it is entirely possible for certain genes to be used two or more times. Each time the gene is expressed it will always generate the same protein, but depending on the other proteins present, may have a different effect. The latter is the more biologically plausible approach, and often occurs in nature. What is crucial, however, is that each time a particular individual is mapped from its genotype to its phenotype, the same output is generated. This is because the same choices are made each time.

To complete the BNF definition for a C function, we need to include the following rules with the earlier definition:

```
<func> ::= <header>

<header> ::= float symb(float X) { <body> }

<body> ::= <declarations><code><return>

<declarations> ::= float a;

<code> ::= a = <expr>;

<return> ::= return (a);
```

Notice that this function is limited to a single line of code, returns a float, and is passed a single float value X as a parameter. However, this is because of the nature of the problems tackled here, the system could easily generate functions which use several lines of code and a different parameter set by simply modifying the BNF grammar definition.

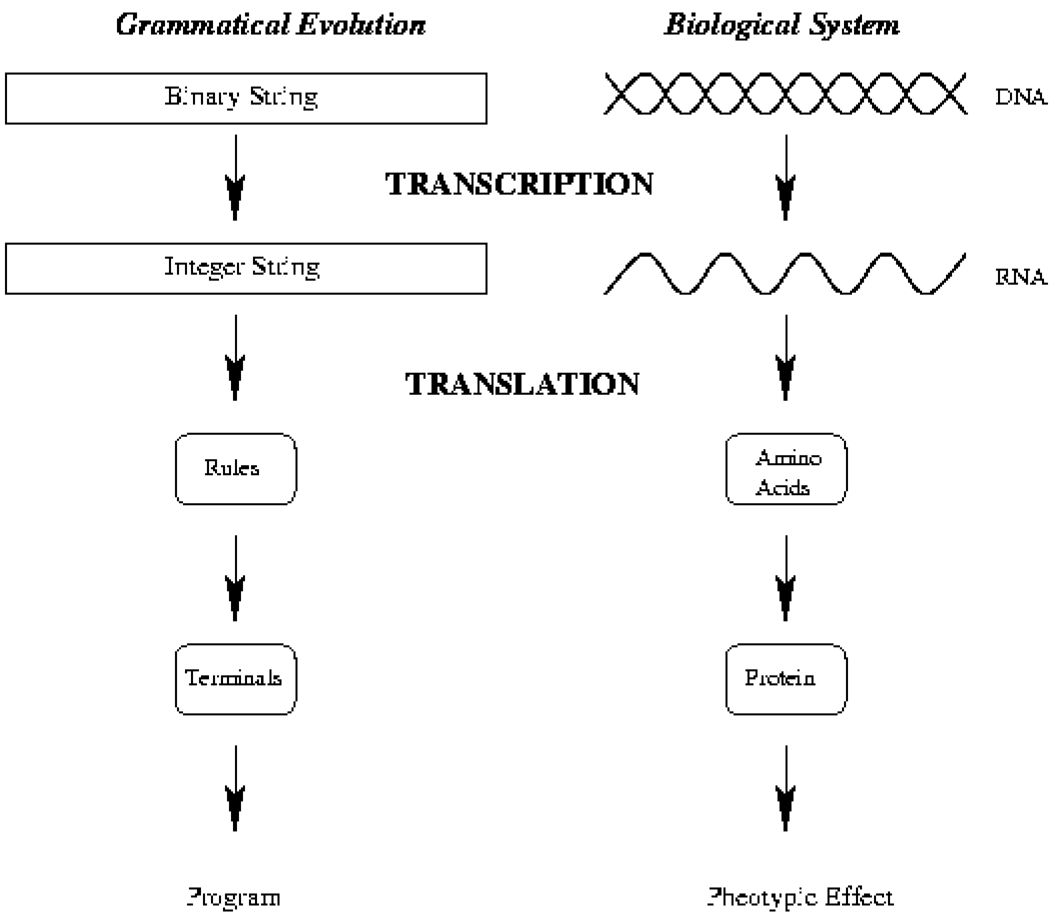

Figure 1

Conclusions

A system, Grammatical Evolution (GE), has been described that can map a binary genotype onto a phenotype which is a high level program. Because our mapping technique employs a BNF definition, the system is language independent, and, can easily generate functions which use several lines of code. GE has been reported successful for three different types of problems, namely, Symbolic Regression [Ryan et al. 98a], finding Trigonometric Identities [Ryan et al. 98b], and a Symbolic Integration problem [Ryan, O'Neill 98a, Ryan, O'Neill 98b].

A more detailed comparison between Genetic Programming and GE is being carried out at present. Other problem domains are being tackled with GE and it is hoped to show that it can perform at least as well if not better than GP across a broad spectrum of problems.

References

[Elseth 95] Elseth Gerald D., Baumgardner Kandy D. 1995. Principles of Modern Genetics. West Publishing Company.

[Horner 96] Horner, H. A C++ class library for GP.Vienna University of Economics.

[Koza 92] Koza, J. 1992. Genetic Programming. MIT Press.

[Paterson 97] Paterson, N Livesey, M. 1997. Evolving caching algorithms in C by GP. In Genetic Programming 1997, pages 262-267. MIT Press.

[Ryan et al. 98a] Ryan C., Collins J.J., O'Neill M. 1998. Grammatical Evolution: Evolving Programs for an Arbitrary Language. Lecture Notes in Computer Science 1391, Proceedings of the First European Workshop on Genetic Programming, pages 83-95. Springer-Verlag.

[Ryan et al. 98b] Ryan C., O'Neill M., Collins J.J. 1998. Grammatical Evolution: Solving Trigonometric Identities. In Proceedings of Mendel '98: 4th International Conference on Genetic Algorithms, Optimization Problems, Fuzzy Logic, Neural Networks and Rough Sets, pages 111-119.

[Ryan, O'Neill 98a] Ryan C.,O'Neill M. Grammatical Evolution: A Steady State Approach. In Late Breaking Papers, Genetic Programming 1998, pages 180-185.

[Ryan, O'Neill 98b] Ryan C.,O'Neill M. Grammatical Evolution: A Steady State Approach. In Proceedings of the Joint Conference on Information Sciences 1998, pages 419-423.

[Whigham 95] Whigham, P. 1995. Grammatically-based Genetic Programming. In Proceeding of the Workshop on Genetic Programming : From Theory to Real-World Applications, pages 33-41. Morgan Kaufmann Pub.

[Whigham 96] Whigham, P. 1996. Search Bias, Language Bias and Genetic Programming. In Genetic Programming 1996, pages 230-237. MIT Press.

[Wong 95] Wong, M. and Leung, K. 1995. Applying logic grammars to induce subfunctions in genetic programming. In Proceedings of the 1995 IEEE conference on Evolutionary Computation, pages 737-740. USA: IEEE Press.

This Ph.D. thesis is being supervised by Dr. Conor Ryan, Department of Computer Science and Information Systems, University of Limerick (conor.ryan@ul.ie).

Tools to Monitor Programmers' Information Seeking Behaviour

Tony Cahill
and
Jim Buckley

Department of Computer Science and Information Systems
University of Limerick
Limerick, Ireland

+353-61-202764
anthony.cahill@ul.ie

+353-61-202772
jim.buckley@ul.ie

Very little empirical evidence had been gathered concerning the "comprehension value" of the representations and facilities which are standard in many re-engineering tool-kits. Likewise, very little work had been undertaken in identifying the programmers_ underlying comprehension behaviour during system understanding.

Current research addresses these areas by proposing a system monitoring instrument to study the behaviour of programmers actively involved in code comprehension. Such an instrument consists of a software browsing component and a monitoring component. The browser component presents various views of the system to the programmer, and provides navigation facilities through these views. The monitoring component _taps_ the programmers_ interactions with the browsing component to capture their behaviour.

Such an instrument has been developed and pilot studies carried out to assess its capabilities. From these studies a number of observation have been made on programmer's comprehension behaviour. It is these observations, along with the associated empirical evidence, which will be discussed in this presentation. In addition, several of the methodological issues which arose during the pilot study are also described.

The Agreement Dimension in Requirements Specifications

Norah Power

Department of Computer Science and Information Systems
University of Limerick
Limerick, Ireland

+353-61-202769
norah.power@ul.ie

My research involves an investigation into the practice of system development focusing on the uses of written requirements documents in different situations, and using qualitative rather than quantitiative research techniques. What I am seeking is a deeper understanding of and insight into the complexities involved in system development than is currently documented in the software engineering literature, in particular, the activities and processes that generate and use requirements specifications. I am attempting to categorise the implications for different kinds of requirements specifications and systematise them into a framework.

One important dimension of this framework is the organisational context. Different organisational and contractual arrangements seem to have clear implications for the role of written requirements documents in practice. I am trying to see if I can use concepts from the literature on organisational behaviour, such as agency exchange, to systematically explain these implications.

It has been claimed that the "requirements-as-contract" model of software development is dead, but this is not generally the case. Certainly, that diagnosis may apply to much of the software that is developed for mass-market "shrink-wrapped" packages; however, a considerable amount of software is still developed under contract. Even within an organisation, in-house developers are often expected to regard their users and user departments as "clients."

The traditional idea of the software requirements document as a contract between the developers and their clients still holds even though the relevance of this notion is now being questioned as it is recognised that software development situations vary widely, and require different approaches.

I have chosen the term "agreement" to characterise this dimension for a number of reasons: it is used extensively by practitioners in discussions, it connotes "understanding" and "obligation" as well as "contract" and also because the concept of agreement applies even in the shrink-wrapped development situation. Agreement is widely recognised as an important aspect of requirements engineering (e.g. many research projects are dedicated to developing techniques to support ways of enhancing understanding and therefore agreement during the RE process). What I am trying to do is map out the "situatedness" of agreement and hence the different varieties of agreement that are possible and the extent to which they are appropriate in different contexts of software development, and therefore how they correlate to the organisational dimension.

Decompilation of Alpha executables

Peter McCarthy

Department of Computer Science and Information Systems
University of Limerick
Limerick, Ireland

+353 61 202710
peter.mccarthy@ul.ie

Decompilation is the inverse of compilation, the reversal of the compilation process. A decompiler starts with an executable image (machine code) and translates the image into some higher level language. Like a compiler, a decompiler generates equivalent output (high level) code based on the input (machine code) file. Where as compilation could be viewed as an encryption process, decompilation could be viewed as a decryption process, and a decompiler as a decryption device.

The syntax of high level languages can be described by means of context free grammars. Compilation is defined as a translation which maps every valid string in a source language (which is describable by a context free language) onto strings of the opcode set of a particular processor. This means that the result of compilation (the machine code) is also a context free language. Therefor the structure of an executable image is describable by means of a context free grammar.

This approach to decompilation has been proven on a CISC architecture. The talk highlighted some of the differences between RISC and CISC architectures, with reference to this methodology.

This M.Sc. thesis is being supervised by Dr. John O'Gorman, Department of Computer Science and Information Systems, University of Limerick (john.ogorman@ul.ie).

A Semi-Automatic Method for the Detection of False Cognates between English and Polish

Gosia Barker[1]
and
Richard F. E. Sutcliffe[2]

Department of Languages and Cultural Studies[1]
and
Department of Computer Science and Information Systems[2]
University of Limerick
Limerick, Ireland

+353 61 202039
gosia.barker@ul.ie

+353 61 202706
richard.sutcliffe@ul.ie

A false cognate may be approximately defined as a word which occurs with similar spelling in two languages L1 and L2 where the most frequent semantic sense in L1 is not the same as the most frequent semantic sense in L2. For example, 'baton' in Polish means a chocolate bar while in English it means a kind of stick. Similarly 'okazja' in Polish means a bargain while 'occasion' in English means an event.

Besides being of interest linguistically, false cognates constitute a hazard for second language learners and translators alike. For this reason, dictionaries of false cognates already exist for a number of language pairs. However, many of these are incomplete or inaccurate. The purpose of the current project therefore is:

- To create an exhaustive list of false cognates between English and Polish,
- To develop semi-automatic methods for the creation of such lists,
- To compose a comprehensive dictionary of false cognates between English and Polish.

So far, an experiment was carried out in which 42 morphological conversion rules (e.g. 'x -> ks') were applied exhaustively to each member of an English word list containing 26,871 entries. Each resulting word was then accepted as a candidate false cognate if it occurred in a list of 109,862 Polish words. For example, by applying 'x -> ks' to 'expert' we obtain 'ekspert' which is a Polish word. A preliminary analysis suggests that around 22% of the resulting 5,745 candidates (i.e. around 1,264 words) are in fact false cognates.

The work is continuing by means of a comprehensive analysis of the above results followed by a further experiment in which an expanded list of conversion rules has been applied to a list of nouns and verbs taken from the British National Corpus with the results be checked against a much expanded list of Polish words.

Two future directions of the work are:

- The development of techniques for the automatic detection of false cognates (at present this must be undertaken by hand following production of the list of candidates),

- The production of a comprehensive dictionary of false cognates between English and Polish.

Runtime Translation of Client/Server HCI

Pat O'Sullivan

Lotus Development Ireland

Software localisation is the process by which a software package and its associated documentation and help files can be transformed to enable it to be used in different language markets. Localisation involves a number of complex processes including:

- Extraction of all prompts and messages from within the software,
- Translation of these prompts and messages into the target language,
- Re-incorporation of the translations into the software,
- Re-Engineering of the software to suit cultural aspects of the target language (e.g. modification to allow double-byte encoded characters and conversion of display modules to allow messages to be shown right-to-left),
- Translation of all on-line help files,
- Translation of all printed documentation.

A product is normally completely tested and fully functional before it is localised. However, the above stages can introduce many new errors into the software. In consequence, a localised product must be fully re-tested before it can be marketed. This testing is complex and time consuming and therefore constitutes a significant proportion of the overall cost of product localisation.

In previous work (O'Sullivan, 1997) an analysis of the different localisation bugs was carried out and in consequence several tools were developed each of which could eliminate certain types of bug. These tools are now in use at Lotus and have significantly decreased localisation costs on a number of product lines. While much progress was made in this work towards the reduction of bugs via the optimisation of current localisation processes, the theme of the current work at University of Limerick is to look at more fundamental changes to the process itself which would eliminate the possibility of those bugs occurring in the first place. In essence, therefore, the primary objective of this research is to investigate the extent to which software localisation can be made a function of text translation only. At present this is being carried out via two strands of investigation:

- The development of a prototype which can carry out translation of prompts etc on-the-fly by intercepting procedure calls at run time. This constitutes a downstream approach applicable to software products which already exist in fully finished form and which need to be localised at low cost;

- The development of a software engineering paradigm which could be adopted into the overall development process and which would make the engineering necessary for localisation an intrinsic part of that process. This constitutes a more upstream process applicable to products which are not yet fully developed.

It is envisaged that in future the Lotus policy on localisation could comprise a combination of these downstream and upstream approaches.

Reference

O'Sullivan, P. (1997). A Software Test Reduction System for use in Localisation Environments. M.Sc. Thesis, Department of Computer Science, University College Dublin, Ireland.

This Ph.D. thesis is being supervised by Dr. Richard Sutcliffe, Department of Computer Science and Information Systems, University of Limerick (richard.sutcliffe@ul.ie).

Reactive Hybrid-Ecological Audio Enhanced Interface

Caolan McNamara

Interaction Design Centre
Department of Computer Science and Information Systems
University Of Limerick
Limerick, Ireland

+353-61-202699
Caolan.McNamara@ul.ie

The Sound of Action Project, is an attempt to enhance the standard graphical desktop interface using audio. Sound is ideally suited to notifying the user of the results of his actions, to express errors and warnings and to monitor background activities. Audio can be used to reinforce the feedback currently given in graphical form, to help minimize confusion over the meanings of the visual cues.

There are two basic branches of audio display, earcons and auditory icons. Earcons are structured and abstract sounds whose meaning must be learnt, earcons consist of an easily generated sequence of tones and chimes and has an internal grammar and structure of its own. Auditory Icons are ecologically tied to the event, the sound they create maps naturally to the event that created them. i.e an auditory icon representation of a button would make a click sound that matches what the real-world counterpoint of that button might make, whereas an earcon would make a musical sound of some kind. The trade off is that earcons are more difficult to recognize and learn, while auditory icons are difficult to parameterize and generate, but are easily understood and recognized by users.

Parameterization is an important part of an audio interface, the audio feedback of the environment should match the action and activities taking place in the environment to create any added value. It is quite easy to parameterize an earcon, as the sound of an earcon can be derived from a set grammar. Auditory icons are more messy to parameterize, it is difficult to artifically generate an appropiate natural-like sound from a given scenario.

What SOAP is attempting to do it find a middle way, a hybrid, between these two approaches. To create a system that uses a simplified framework of sounds that allow us to generate auditory icons that while not perfectly natural sounding, like true auditory icons, have the key invariants that users use to identify the nature of real world sounds. As an example, the key invariant in judging the speed and distance of a car from its sound is the rate of increase of its pitch, the other elements of the sound are non-essential in determining this information. Just so, we hope to create a range of skeleton sounds that we can use in the desktop to harness the natural strengths of the users listening skills to enhance the existing graphical environments.

This M.Sc. thesis is being supervised by Dr. Liam Bannon, Interaction Design Centre, Department of Computer Science and Information Systems, University of Limerick (liam.bannon@ul.ie).

An Interactive Composition System

Ian O’Keeffe.

PKS Ltd.
National Technological Park
Castletroy
Limerick, Ireland

iokeeffe@pks.ie

A compositional/performance system that allows the composer/performer to compose and perform in real-time. A rhythmic structure is set up, and this percussive musical strand can be altered in real-time by the user. Melodic musical strands are played into the system live, within the constraints of performance time units which are synchronised to the rhythm. These time units vary in length from instrument to instrument. Interactive use of the keyboard exercises a gestural influence on the various musical strands, via velocity, rapidity, key, and location, as well as providing an optional melody line. This influence ranges from tempo alteration, modal orientation and velocity to instrumentation, percussive fills, and ’mood’ and ’darkness’ changes."

This M.Sc. (Music Technology) thesis was supervised by Dr. Niall Griffith, Department of Computer Science and Information Systems, University of Limerick (niall.griffith@ul.ie).