Manhattan: End-User Programming for Music

Chris Nash
Department for Computer Science and Creative Technologies,
University of the West of England,
Frenchay Campus, Coldharbour Lane,
Bristol, BS16 1AH
chris.nash@uwe.ac.uk

ABSTRACT

This paper explores the concept of end-user programming languages in music composition, and introduces the Manhattan system, which integrates formulas with a grid-based style of music sequencer. Following the paradigm of spreadsheets, an established model of end-user programming, Manhattan is designed to bridge the gap between traditional music editing methods (such as MIDI sequencing and typesetting) and generative and algorithmic music – seeking both to reduce the learning threshold of programming and support flexible integration of static and dynamic musical elements in a single work.

Interaction draws on rudimentary knowledge of mathematics and spreadsheets to augment the sequencer notation with programming concepts such as expressions, built-in functions, variables, pointers and arrays, iteration (*for* loops), branching (*goto*), and conditional statements (*if-then-else*). In contrast to other programming tools, formulas emphasise the visibility of musical data (e.g. notes), rather than code, but also allow composers to interact with notated music from a more abstract perspective of musical processes.

To illustrate the function and use cases of the system, several examples of traditional and generative music are provided, the latter drawing on minimalism (process-based music) as an accessible introduction to algorithmic composition. Throughout, the system and approach are evaluated using the cognitive dimensions of notations framework, together with early feedback for use by artists.

Keywords

end-user programming, algorithmic composition, generative music, minimalism, sequencers, trackers, digital music notations

1. INTRODUCTION

Several disjunctions exist between mainstream music editing packages, based on arranging and transcribing notes or musical events (e.g. sequencers, score editors), and music programming tools, based on defining abstract processes that dynamically generate music (e.g. SuperCollider, Max). The divide spans not only working methods and interaction styles (manual vs. generative, direct manipulation vs. programming, usability vs. virtuosity), but also artistic aesthetics, practices and communities (popular vs. avant-garde, traditional vs. experimental).

Traditional low-level music editing (sequencing, transcribing, arranging) focuses on a static, concrete and detailed specification of music, where individual notes are addressable and manipulated manually. Based on the recording studio, sequencers focus on capturing a live performance and freezing it on a linear timeline, offering limited opportunities or tools to articulate abstract musical processes or concepts [2]. By contrast, music programming tools enable composers to define abstract processes, and programmatically generate music using dynamic composition techniques, as used in generative, procedural, aleatoric, or process-based music. However, by raising the abstraction level, composers concede low-level control of musical elements (e.g. individual notes or events); it becomes harder to deviate from the formal algorithm [16].

The Manhattan project seeks to develop a unified environment supporting both contemporary and traditional music creativity, by flexibly combining low-level (*concrete*) and high-level (*abstract*) composition practices, in 'mixed-mode' works and workflows, supporting fusion and crossover music that bridges mainstream and avant-garde aesthetics (cf. [8]). Example scenarios include: sequenced music using dynamic elements (e.g. random values, harmonisation, context-sensitive phrases, non-linear musical forms); music using formulas to simplify or automate editing; generative processes outputting music that can be directly edited or manipulated; pieces with parallel or synchronised, but otherwise separate parts, each entered manually or generated programmatically.

Manhattan[1] proposes a synthesis of sequencing and scripting elements, based on the concept of musical grids and formulas. This paper draws on research into digital creativity (see [14]), end-user programming (see [13]), and the *cognitive dimensions of notations* [4] to explain how the spreadsheet paradigm successfully integrates end-user editing with programming functionality, and illustrate how the concept of formulas is adapted and extended for use in computer music, to create a system combining sequencer interaction and the core features of an imperative programming language, with support for conditional statements, loops, branching, variables, pointers, and arrays. The sequencing element draws on tracker notation, a general-purpose, text-based approach for specifying detailed patterns of music, with an appearance and interaction style similar to spreadsheets, and musically similar to an advanced step sequencer or data list. A working prototype is detailed, using analogies to programming languages (such as *BASIC*) and music notation, with practical examples illustrating applications ranging from traditional music editing to generative music, revisiting two pieces of process-based music, Steve Reich's *Piano Phase* and Arvo Pärt's *Fratres*.

2. BACKGROUND

Most systems can be categorised as "low threshold and low ceiling" (built for usability) or "high threshold and high ceiling" (built for experts) [9], as seen in the design of music editors and programming languages [2]. In music, end-user tools such as sequencers are based on low-level data structures (notes, MIDI), manipulated through preset abstractions, processes or metaphors from traditional practices such as music performance and the studio [3], confining the creative process to established paths and working styles [11]. By contrast, programming enables the formal definition of more abstract musical processes, but at the expense of access to low-level data and the freedom to deviate from algorithms [16]. The goal of this project is to find a flexible structure that permits both direct editing of low-level data and the definition of dynamic elements and musical processes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
NIME'14, June 30 – July 03, 2014, Goldsmiths, University of London, UK.
Copyright remains with the author(s).

[1] Named for the grid-like street layout of New York, used in computing to describe rectilinear distances in grid-based geometry, and the original setting for early minimal (process-based) music.

2.1 Spreadsheets and End-User Programming

Spreadsheets are widely recognised as a successful paradigm in end-user programming [7][13]; a programming language designed for non-programmers [10] - it is "possible that more people program with spreadsheets than any other programming environment." [5]

The spreadsheet UI is designed for usability, based on the simple, flexible, and familiar structure of the table, extended by formulas that allow users to define basic relationships between cells, through mastery of only two concepts: cells as variables, and functions as relations between cells [13]. Formulas are automatically recalculated when their dependent values change, providing instant feedback that enables users to "tinker" and experiment with "what-if" scenarios. Users can also selectively learn and use formulas, providing a low threshold for novices, who can plug values into cells manually, and a scalable approach to more complex functionality that allows them to appropriate and extend the notation for their own uses [5][10].

Ko *et al.* [6] characterise the spreadsheet metaphor as a concrete, human-centric approach to end-user programming, in contrast to more abstract or computer-centric approaches (see Figure 1), but one that does not extend well to general-purpose computation [13].

Using the *cognitive dimensions of notations* [4],[2] Hendry and Green [5] discuss the spreadsheet's concreteness as *closeness of mapping*, allowing users to transfer knowledge and formulate problems within the task domain (see [10]); using a familiar, visual representation that allows users to feel as though they are working directly on the task [7]. Additions and changes to the notation can be made quickly and in any order, facilitating rapid editing (*low viscosity*), reducing the need to plan (*premature commitment*), enabling "idea sketching", and offsetting *error-proneness* in exploratory design contexts [5]. Formulas similarly support *progressive evaluation*, allowing the testing of partial and incomplete solutions [4], in contrast to tools where programmers must recompile, re-execute, or re-enter data in order to test a change [7][10]. This engenders a high "level of liveness", facilitating tinkering, exploratory creativity, and learning by experimentation [12].

Spreadsheets offer a *declarative, constraint-based* model of programming that avoids control flow (loops, branching), in favour of data flow – similar to *visual programming languages*, a popular approach to end-user and novice programming, also seen in music (e.g. *Max*). However, cell dependencies and execution order can be unclear, making it difficult to define events and processes over time, which is needed in music [17]. In the this paper, the spreadsheet model is extended to support the control and abstraction of time, in an *imperative* programming style, based on familiar concepts of musical time.

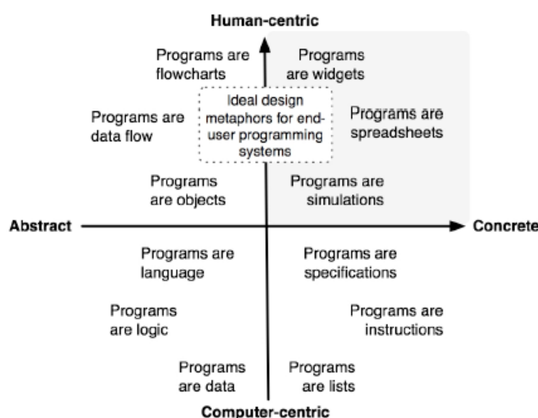

Figure 1. Programming Metaphors (from [6])

Figure 2. The reViSiT tracker [11]

3. MANHATTAN

Manhattan system is designed as an integrated composition environment combining direct music editing (i.e. sequencing) and end-user music programming. Section 3.1 introduces tracker notation, a textual, grid-based style of sequencer software, similar in form and function to the spreadsheet, which is extended to support formula expressions in Section 3.2. A variety of use cases are then detailed in subsequent sections.

3.1 Trackers as grid-based sequencers

Trackers ([11], e.g. Figure 2) are a class of sequencer based on a concise text notation, edited using the computer keyboard. Music is represented in fixed grids (*patterns*), visually similar to a spreadsheet, where columns represent tracks and rows represent fixed time slices like a step sequencer. Each cell has spaces for: pitch, instrument, volume (or panning) and one of an predefined set of musical effects, for example: **C#5 01 64 D01** triggers [C#5], using voice [01], volume [64], with a slow [01] *diminuendo* [D].[3]

As in spreadsheets [5][10], the rapid editing interaction (*low viscosity*) and fast feedback cycle (*progressive evaluation*) supports a high degree of liveness, enabling sketching and flow [12]. In both cases, formulas offer a visual and interactive mode that easily integrated with the existing editing UI, emphasising the visibility of data, rather than code. Unlike spreadsheets, however, there is a linear time axis and sequential execution order, spatially illustrating the event timing, similar to a score or graphical views in sequencers. As detailed in Section 4, this engenders a shift from the *declarative*, *data-flow* programming style of spreadsheets, to an *imperative* programming style, with an explicit representation of time, while retaining the 2D grid layout that allows music relationships to be shown spatially (see [13]). Moreover, there is a high degree of parallelism implicit in the pattern, as in music generally; musicians are familiar with many complex concepts of concurrency, e.g. synchronisation of parts, counter-point, polyrhythms, polytempi; the interplay of parallel processes is visible in adjacent tracks of the pattern, where a sequential (left-to-right) execution order further simplifies concurrency, a major hurdle for novice coders [13].

3.2 Integrating Formulas

Like spreadsheets, scripting is introduced at the cell-level, where properties of individual notes or events are manually entered or defined by formula expressions written using mathematical operators (e.g. +, -, /, *), built-in functions (e.g. *abs*, *rnd*, *mod*), conditional statements (e.g. *if-then-else*), and references to data elsewhere in the piece. Unlike spreadsheets, each cell defines multiple values and formulas corresponding to different musical properties (pitch, instrument, volume, panning, effect).

[2] The framework was developed to describe usability factors in programming languages, but has subsequently been adapted for wider use in HCI, including computer music [2][11].

[3] Manhattan is based on the *reViSiT* tracker (Figure 1), originally developed as a VST plugin for sequencers, extending the tracker paradigm to improve usability and musical expressivity [11].

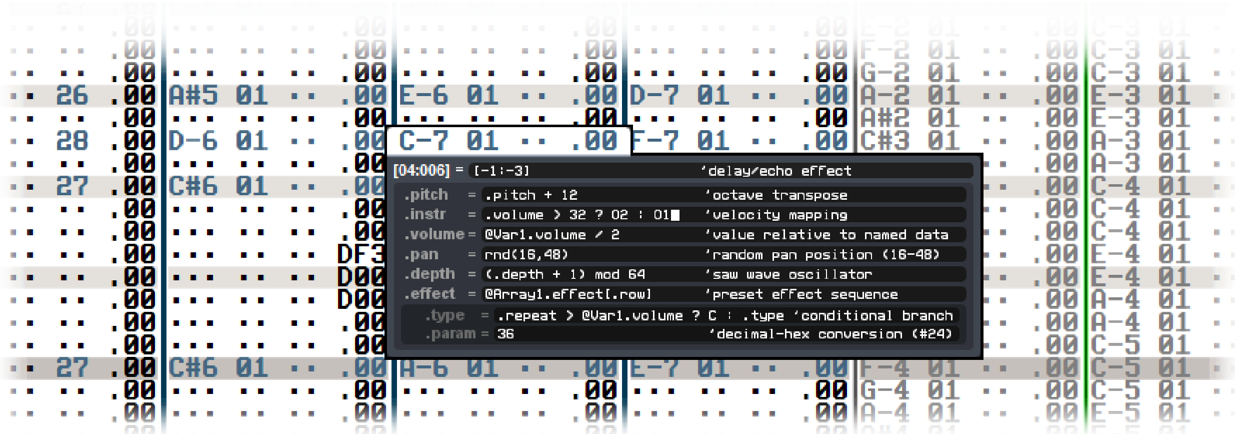

Figure 3. Formula Editing in Manhattan, with examples.

Formula evaluation occurs when cells are played, updating existing data in the cell. Resulting notes and values are thus both audible and visible, and also become editable, just like manually sequenced music. Outside of playback, formulas are evaluated upon editing, offering instant feedback and enabling experimentation. Partial or incomplete processes can thus be incrementally tested, supporting progressive evaluation and liveness (see Section 4.2.1 for example).

While relationships and constraints are declared as in spreadsheets, the serialised execution (playback) order of cells engenders an imperative style of programming, supporting control flow structures like loops (*for*, *while*) and both conditional (*if-then-jump*) and unconditional (*goto*) branching that can be used to control playback (e.g. musical form). While this increases the number of concepts to learn and cognitive effort in coding [10][15], music notations include analogous concepts – e.g. musicians know iteration (loops) as repeats and branching (*goto*) as jumps such as **D.S.** (*dal segno*).[4]

Like spreadsheets, all data is contained in the visible cells of the pattern and referenced using Cartesian-style x/y coordinates (*[3:4]*), but can be given a custom label to provide mnemonic handles in formulas (*@Foo*) that act like named variables (or pointers). Syntactic sugar is also provided to abbreviate references, such as defaulting to current column (*[3]*; row 3, current channel) or relative coordinates (*[-1]*; previous row). This exploits the locality of formulas [10], keeping expressions concise and portable, and allowing a formula and its referenced cells to be moved or copied as a block without breaking their function. The label mechanism also provides a form of *secondary notation* that can be used to annotate or comment sections of the music or formulas, which also support end-of-line commenting (`). Like spreadsheets, unused cells in the grid provide space for informal descriptions of musical elements, processes, or other information such as section headers, lyrics, etc.

Data is 'initialised' by a conventional edit to the pattern, avoiding the confusion novice coders might have with variable declaration and initialisation, or memory allocation [13]. Labels and coordinates, however, support *pointer arithmetic* to provide array functionality (e.g. *@Foo[2]* adds two addresses to point to the second row below "Foo"). Users can thus enter a series of values in the pattern, label the series, and reference it as an array in code. The data is always visible, but can be placed in muted tracks to avoid playback, for use as formula constants or parameters. Muted cells are not evaluated, also allowing users to selectively execute and test formulas that are parts of a larger system. In unmuted channels, unused cell digits (e.g. volume, effect parameters) may also be repurposed to store intermediate values or formulas without being themselves audible.

Since tracker notation is alphanumeric, it can be directly referenced in formula expressions, preserving *closeness of mapping* and *consistency*. For example, pitches can be entered as shown in the pattern (e.g. *C#5 + 4*). Cell properties are accessed like class members in object-oriented programming (e.g. *[2].volume*, *@Foo.effect.type*). Most properties are represented as integers or enumerated values (pitches, effect types), but processed using floating-point numbers, extended to also support complex types using *NaN-tagging*.[5] This, for example, enables whole cells to be referenced, compared, and copied, e.g. *[3:1] = [3:0]*.

Formulas are visible only during editing, which can impact code readability, hide dependencies between elements, and make the flow of complex pieces hard to follow. Hendry and Green [5] suggest using alternative visual modes and *secondary notation* to address such issues. Thus, in addition to the label mechanism, an additional visual mode can be toggled to visualise dependencies between formulas and referenced data (see Figures 3 and 4). During editing, the current cell's dependencies are shown; during playback, all currently playing cells' dependencies are shown. While the 'spaghetti' associated with non-structured programming (and visual programming, e.g. *Max*) is apparent for more complex processes, channels can be selectively muted and disabled to restrict formula evaluation and dependency visualisation, making it easier to inspect, learn, and debug expressions iteratively. Other visualisations are also supported: displaying all formulas in the pattern grid at once, or serialising them into an imperative-style code listing – which could later be extended to detect control flow structures (*for*, *if-then-else*, *while*) and automatically translate the listing to a *BASIC*-style pseudocode (e.g. Figure 4; cf. *CogMap* in [5]).

As an imperative programming environment, Manhattan is broadly equivalent to *non-structured programming languages* (e.g. *BASIC*), where code structure is largely inherent in the underlying machine – here, the grid of the pattern. Although non-structured programming is unsuited to major software projects, musical algorithms are less complex. The simplicity of the syntax and low-level structure make such languages easy for novices to learn, drawing on rudimentary maths skills while exposing the basic building blocks of coding [15]. Similarly, issues of maintainability, readability and correctness are arguably less critical in artistic expression; so long as artists can understand and read their own work, the removal of structural constraints grants them greater editing freedom (though become significant in collaborative scenarios and knowledge sharing).

The current lack of support for function definition means recursion is not directly supported, though its omission reduces the chance of stability issues or coding errors arising (e.g. infinite recursion).

---

[4] In trackers, repeats and jumps are supported by effects (*Cxx*, jump to row *xx*; *SB0/SBx*, repeat *SB0* to *SBx*, *x* times), which can be controlled via formulas and used with an *if-then-else* clause (*x ? y : z*) to enable conditional loops and branching, as seen in Section 4.2.2.

[5] The process of setting a floating-point variable's *Not-a-Number* bit, using remaining bits to encode custom binary data, thus supporting both fast math operations and extended data types.

However, Manhattan is *Turing complete* (see Video 1) and, in many cases, iteration offers a practical alternative to recursion, while also representing a concept more easily assimilated by novice coders [13][15] and familiar to musicians (see earlier).

4. FORMULAS IN PRACTICE

Formulas can be used for a variety of purposes, from isolated expressions that simplify or automate editing to more generative examples that use algorithms to define the content or structure of a piece. This section discusses several use cases for the system, illustrating how formulas can not only support and extend conventional music editing, but also enable generative processes; using two examples from process-based, minimalist music: Arvo Pärt's *Fratres* and Steve Reich's *Piano Phase*.

4.1 Traditional Music Editing

Formulas can be used to support, simplify, or automate traditional music editing tasks. Such uses target productivity, rather than creativity, but also represent intermediate and practical applications that introduce end-users to basic functions, syntax, and coding concepts, acting as a stepping stone to more complex uses (e.g. generative music).

Formulas can benefit usability by reducing *hard mental operations*, offering users simpler ways to work with tracker notation, such as entering values as fractions (*64/7*) or hex values (used in effect settings) as decimals (*#24=36*). Defining simple relationships or constraints (*[3].volume/2*, *[-1].pitch+4*) can reduce the *knock-on viscosity* of editing, so that subsequent changes to referenced data are automatically propagated to dependent cells. Basic formulas can thus be used to define common musical devices and techniques, such as arpeggios, echoes, transposition, or automatic harmonisation. In this way, composition becomes a constraint-satisfaction problem [1], where composers develop 'solutions by finding and solving constraints that gradually restrict the set of possible solutions; formulas allow them to fix known constraints and focus experimentation on unresolved artistic decisions.

Formulas can also be used to simulate elements of musical prosody in live performances, by adding small, random variations to the notated music (e.g. timing, volume, pitch). While this approach is unlikely to rival the emotion or virtuosity of human rendition, it can make textures sound richer and rhythms more natural, masking the digital precision that can make digital music feel rigid or mechanical.

By abstracting time, control flow statements can reduce the *diffuseness* of music. Repeats and jumps are common in music notation, allowing composers to abstract patterns and more concisely represent music. Scores include simple conditional statements, such as the first and second repeat endings or codas, which can be encapsulated in formulas (e.g. *if-then-else*). Formulas can similarly be used to abstract dynamic forms and progressions in many styles of folk and popular music, from cumulative songs based on extending and repeating short phrases (e.g. *12 Days of Christmas*) to the iterative layering and switching of elements in progressive music (e.g. house, trance).

Figure 4. Steve Reich's *Piano Phase* in Manhattan, with formula listing.

Labels and Initial values

```
@Phase  = [1:0]     @Phase.Volume  = 0
@Notes  = [2:0]
```

Formulas (and manual entry)

```
    @Phase.Volume  = (.Repeat == 15) ? .Volume + 1 : .Volume
  ( [4:0].effect   = "SB0" )
            [4:*]  = @Notes[(24 - @Phase.Volume + .Row) Mod 24]
      [4:*].Instr  = (.Pitch != None) ? 3 : None
[4:*].Effect.Param = #D + @Phase.Repeat
 ( [4:23].effect   = "SBF" )
```

In mathematics and programming, conciseness represents an artistic aesthetic; solutions that are shorter or simpler are often seen as more elegant or beautiful. A similar aesthetic exists in digital music communities, such as tracking [11], which explicitly celebrate the virtuosity involved in creating complex art using limited or minimal resources (e.g. explicitly restricting file size or limiting polyphony). These 'minimal' trends are not associated with experimental music, but more mainstream genres, and practitioners may thus see formulas as a new outlet for demonstrating technical virtuosity.

4.2 Generative Music

Formulas can be used to explore and develop techniques in contemporary art music, such as algorithmic composition, aleatoric music (e.g. using random function, *rnd*), atonal and serial music based on alternative pitch systems, or otherwise derived from formal or mathematical processes.

While the examples here demonstrate how the system can be used to define formal processes in generative music; the underlying sequencer architecture enables integration with manually-edited music, supporting crossovers between artistic styles. The minimalist music of Steve Reich, himself a notable proponent of (re)integrating disparate art and popular music cultures, has inspired many popular artists and reversions of his process-based music (e.g. the Orb, ColdCut, Brian Eno, David Bowie), and it is hoped that end-user music programming tools can be used to bring generative processes into mainstream music. Minimalism, specifically, is a school of experimental music based on simple generative and process-based works often routed in consonant tonality, offering a musical aesthetic more accessible to composers and listeners accustomed to popular and traditional styles. The style shifts the artistic focus from individual notes and phrases to processes and more abstract concepts in a way that can be supported and explored in a formula-based editing environment like Manhattan.

4.2.1 Steve Reich's "Piano Phase" Revisited

Piano Phase is a 1967 minimalist composition by Steve Reich, based on a repeated phrase of twelve notes played by two pianists at different tempi. Over several minutes, cycles of each phrase gradually move out of phase, before returning to synchronisation. During the piece, melodic interaction between the two parts creates interesting counterpoints that dramatically change the character of the music; the listener unconsciously merging the parts, chunking the evolving stream of notes into distinct melodies or harmonic textures.

In Manhattan, the piece is replicated by encoding the twelve-note phrase as an array, then offsetting playback to simulate the phase change. For the first part, the original phrase (*@Notes* array) is played back looped. Each cycle, a counter increments to shift the phase of the second part. To refine the phase shift, each increment plays 16 times, each time delaying the second part by a sixteenth of a row, using the note delay effect (*SDx*, where *x* is delay), slowly edging to the next row and array offset. The pattern and formulas are shown in Figure 3, and execution in Video 2.

01:Repeats | 02:Baseline | 03:Bottom | 04:Middle | 05:Upper | 06:Scale | 07:Tintinnabuli | 08:Tintinnabuli

BASIC-Style Pseudocode (with row numbers)

```
        LET Start = 27
        LET Phase = 0
        DIM DMinor AS ARRAY
        DIM 2ndPosSup AS ARRAY
        DIM 2ndPosInf AS ARRAY

00:     Phase = Phase + 1
        PLAY DMinor[Start]
        PLAY 2ndPosInf[Start]
        PLAY DMinor[Start + 9]

04:     FOR Repeat = 0 to Phase
            LET x = Start - Repeat - 1
            PLAY DMinor[x]
            PLAY 2ndPosInf[x]
            PLAY Dminor[x + 9]
05:     NEXT Repeat

06:     FOR Repeat = 0 to Phase
            LET x = Start + Phase - Repeat
            PLAY DMinor[x]
            PLAY 2ndPosSup[x]
            PLAY Dminor[x + 9]
07:     NEXT Repeat

08:
        PLAY DMinor[Start]
        PLAY 2ndPosInf[Start]
        PLAY DMinor[Start + 9]

13:     IF Phase < 3 GOTO 00
```

Labels and Initial Values

```
   @Phase = [1:0]
   @Start = [2:0]
  @DMinor = [6:0]
@2ndPosSup = [7:0]    @Phase.Volume  = 0
@2ndPosInf = [8:0]    @Start.Volume  = 27
```

Formulas (and manual entry)

```
  @Phase.Volume = .Volume + 1
          [3:0] = @DMinor[[2:+0].Volume]
          [4:0] = @2ndPosSup[[2:+0].Volume]
          [5:0] = @DMinor[[2:+0].Volume + 9]

  ( [1:4].Effect = "SB0" )
    [2:4].Volume = @Start.Volume - [1:+0].Repeat - 1
           [3:4] = @DMinor[[2:+0].Volume]
           [4:4] = @2ndPosSup[[2:+0].Volume]
           [5:4] = @DMinor[[2:+0].Volume + 9]
[1:5].Effect.Param = #B0 + @Phase.Volume

  ( [1:6].Effect = "SB0" )
    [2:6].Volume = @Start.Volume + @Phase.Volume - [1:+0].Repeat
           [3:6] = @DMinor[[2:+0].Volume]
           [4:6] = @2ndPosInf[[2:+0].Volume]
           [5:6] = @DMinor[[2:+0].Volume + 9]
[1:7].Effect.Param = #B0 + @Phase.Volume

    [2:8].Volume = @Start.Volume
           [3:8] = @DMinor[[2:+0].Volume]
           [4:8] = @2ndPosInf[[2:+0].Volume]
           [5:8] = @DMinor[[2:+0].Volume + 9]

[1:13].Effect.Type = @Phase.Volume < 3 ? C : none
```

Figure 5. Arvo Pärt's Fratres in Manhattan with formulas (right) and BASIC-style pseudocode (left).

In an imperative model of programming, code manipulates the state of the program (e.g. memory). In Manhattan, this state is preserved in the visible pattern data, and remains after playback ends, making it easy to stop, edit, and continue – enabling the user to interactively test and tinker with variable and formulas without restarting from the beginning, thus improving the liveness of coding and debugging. For example, in a long and slowly-evolving progression like *Piano Phase*, the user can edit the pattern data (i.e. *@Phase.volume*) to jump to an arbitrary stage of the piece.

The example also highlights limitations of the grid-based formula system. In Figure 3, phase is modelled explicitly, in order to simulate polytempi, which is hard to visualise in a grid that imposes a unified tempo. This limitation is an inherent trade-off imposed by the linear timeline of sequencers, which surrenders some of the abstractive power with respect to musical time, in order to preserve a more concrete, traditional paradigm for manual editing.

4.2.2 Arvo Pärt's "Fratres" Revisited

Fratres is a series of pieces written between 1977 and 1992 by Arvo Pärt, based on a formal process that generates a melodic progression, which is used to produce a three-part harmony. The melody is built of four phases: (1) beginning on a given pitch, (2) stepping along a diatonic scale *away* from the pitch, then (3) stepping in the same direction along the scale *towards* the pitch, until (4) returning to the pitch. The sequence repeats three times, each time increasing the steps taken along the scale (see Figure 5). The entire process is repeated several times in the piece, alternating between stepping down and up the scale.

The harmony comprises a lower voice using the melodic progression, a parallel upper voice transposed 9 scale steps up (diatonically) and a middle voice determined by the *tintinnabuli* method, based on the other parts. This method selects a pitch by stepping away from the given note a set number of times, exclusively using pitches of a chosen chord. In *Fratres*, Pärt uses the A minor triad, restricting steps to A, C, E. In the first half of each sequence, where the progression begins and steps away (1, 2), Pärt takes two steps up from the lower voice (*second position superior*). In the second half of the sequence, where the melodic progression steps toward and ends (3, 4), he takes two steps down from the upper voice (*second position inferior*).

In Manhattan, the melodic progression is modelled using a single, looped pattern, divided into two halves that alternate between falling and rising versions of the sequence. Each half is divided into four row sections corresponding to the phases of the sequence, repeated three times. On each iteration, the second and third sections are extended by incrementing their repeats *(SBx)*, allowing their formulas to step further along the scale. Within the these sections, the repeat iteration is found by querying an internal counter (*.repeat*) for the repeat effect (*SBx*). At the third iteration, the instruction to jump back to the beginnging (*Cxx*) is cleared, allowing playback to proceed to a rising version of sequence. The current iteration of the whole sequence is tracked using a counter that increments when played (*@Phase.volume*). This musical structure and melodic progression is encapsulated by the first two tracks of the pattern shown in Figure 4, used for repeats/jumps and the progression in numeric values, respectively.

Figure 6. Melodic and harmonic progression in Fratres.

The next three tracks (03-05) use these values to index arrays stored in muted channels, containing the notes for playback. The *@DMinor* array contains the scale of D Minor, used for lower and upper voices: the lower directly indexes the array; the upper adds 9 to the original offset, transposing the part. Lastly, depending on the phase (2 or 3), the middle voice uses either the lower or upper voice's offset to index one of two arrays of pre-calculated *tintinnabuli* values, for notes in the adjacent scale: second position superior (*@2ndPosSup*) and inferior (*@2ndPosInf*) respectively.

Video 3 shows the script in action, and Figure 4 shows the pattern and formulas with equivalent *BASIC*-style pseudocode. The comparison highlights the close mapping between music (repeats, jumps) and programming concepts (conditional clauses, iteration, branching), though broader musical processes are notably clearer in the pseudocode, and partially-obfuscated by the dispersal and selective visibility of formulas in the pattern. While this can again be seen as a trade-off against the increased visibility of music data (notes), it would be feasible to automatically-translate pattern formulas into code-style listings, as in Figure 4, within the editing environment itself (cf. [5]).

5. DISCUSSION & FUTURE WORK

This paper has presented end-user programming as an approach to unifying computer-based composition practices based on more concrete, low-level direct editing interfaces and more abstract and formal programming languages, and described a system integrating formulas in a grid-based sequencer. As in spreadsheets, formulas lower the threshold for, and enable graduated levels of, programming by extending rather than replacing conventional workflows, supporting applications in both mainstream and experimental music, plus fusions of the two. The visibility of data keeps the results of code processes apparent, while the mapping of code elements to a linear timeline and execution order engenders a flexible imperative-style programming language with control flow structures based on familiar concepts from music, mathematics, and spreadsheet use.

Early feedback from artists using the system has been both positive and informative, and the design of the system has already been shaped by responses from users. However, a more in-depth study of user experiences is planned, to collect feedback from non-musicians, traditional sequencer users, and composers in experimental music.

Formulas could enable end-user programming in other types of music software, though it remains to be seen if a suitable reference system can be developed when the UI is not based around a unified notation (e.g. sequencer) or grid (e.g. score editor). Sequencers may pose a challenge, where the prosody in live performance already raises issues for translating data to notation [3][12].

Research in end-user programming (and spreadsheets) offers directions for extending Manhattan, including: graphical debugging, meta-programming (code macros), self-modifying code (formulas as cell properties), function calling and recursion [15], modularity (custom abstractions or groupings) [7], and external bindings to other tools (e.g. DAWs, Max/MSP/Jitter), protocols (MIDI, SysEx, *OSC*, IP) and both general-purpose and live coding programming languages (C/C++, Java, SuperCollider, Max).

As users gain experience, extensions can offer more flexible coding styles, supporting advanced programming. Ultimately, scripts could be situated in the music (in cells) with capabilities, syntax, and interaction styles similar to general-purpose programming languages, with the ability affect data in any part of the pattern, significantly increasing the expressivity of the notation and spreadsheet model. While any extension adding complexity risks raising the threshold for novices, so long as the music remains visible and editable, scripting functionality optional, and formulas exist to provide stepping-stones for beginners, the provision of multiple end-user programming approaches may illustrate a mechanism for scaling the challenge and offering a graduated learning experience, raising the creative ceiling of the notation as the user develops ability.

6. ACKNOWLEDGEMENTS

Many thanks to Sam Aaron, Darren Edge, TOPLAP, and the [livecode] mailing list for valuable insights and feedback on the project. Thanks also to composers Esa Ruoho (aka Lackluster), Maarten van Strien, and Phill Phelps for ongoing user feedback.

7. SUPPORTING MATERIALS / VIDEOS

The following videos available from: http://video.revisit.info:

Video 1. *Manhattan: Conway's Games of Life.*
A video of the system running Conway's *Game of Life*, a cellular automaton known to be Turing complete.

Video 2. *Manhattan: Steve Reich's Piano Phase.*
A video of the system simulating a piece of process-based, minimalist music based on repeated playback of the same phrase, gradually moving in and out of phase.

Video 3. *Manhattan: Arvo Pärt's Fratres.*
A video of the system simulating a piece of process-based, minimalist music based on a mathematical melodic prog-ression, harmonised using Pärt's *tintinnabuli* method.

8. REFERENCES

[1] J. Alty. Navigating though Compositional Space: The Creativity Corridor. *Leonardo,* 28, 3 (1995), 215-9.

[2] A. Blackwell and N. Collins. The programming language as a musical instrument. *Proc. of PPIG 2005*, 120-130.

[3] M. Duignan. Computer mediated music production: A study of abstraction and activity. *PhD thesis*, Victoria University of Wellington, 2007.

[4] T. Green and M. Petre. Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework. *Journ. of Vis. Lang. & Comp.* 7:131-74, 1996.

[5] D.G. Hendry and T. Green. Creating, comprehending and explaining spreadsheets: a cognitive interpretation of what discretionary users think of the spreadsheet model. *Int. Journal of Human-Comp. Studies,* 40, 1994, 1033-1065.

[6] A.J. Ko, B.A. Myers, and H.H. Aung. Six Learning Barriers in End-User Programming Systems. *Proceedings of VL/HCC 2004*, 2004, 199-206.

[7] C. Lewis and G.M. Olson. Can Principles of Cognition Lower the Barriers to Programming? *Empirical Studies of Programmers: Second Workshop*, 1987, 248-263.

[8] A. McLean. Artist-Programmers and Programming Languages for the Arts. *PhD Thesis*, Goldsmiths, 2011.

[9] B. Myers, S.E. Hudson, and R. Pausch. Past, Present, and Future of User Interface Software Tools. *ACM Trans. on Computer-Human Interaction*, 7(1): 3–28, 2000.

[10] B.A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. 1993.

[11] C. Nash and A. Blackwell. Tracking Virtuosity and Flow in Computer Music. *Proceedings of ICMC 2011*, 575–82.

[12] C. Nash and A. Blackwell. Liveness and Flow in Notation Use. *Proceedings of NIME 2012*, 28–33.

[13] J.F. Pane and B.A. Myers. Usability Issues in the Design of Novice Programming Systems. *Carnegie Mellon University, Technical Report CMU-CS-96-132*, 1995.

[14] M. Resnick *et al.* Design Principles for Tools to Support Creative Thinking. *Creativity Support Tools* (ed. Shneiderman *et al.*), 2005, 25-36.

[15] J. Rogalski and R. Samurçay. Acquisition of Programming Knowledge and Skills. *Psychology of Programming (ed.* J.-M. Hoc *et al.*), 1990, 157-174.

[16] R. Rowe *et al.* Putting Max in Perspective. *Computer Music Journal* 17, 2, 1993, 3-11

[17] G. Wang. A History of Programming and Music. *Cambridge Companion to Electronic Music (ed.* N. Collins and J. d'Escriván), 2007