Live coding in Western classical music

Álvaro Cáceres Muñoz
Theatre, Film, TV and Interactive Media, the University of York
alvaro.caceres@york.ac.uk

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Abstract

This research project explores how to maximize the usability of live coding tools (which allow improvising music with programming) for classical musicians.

To do so, two goals are set for the project: understanding how live coding can be used in traditional composition, and understanding how do classical musicians feel when live coding. These goals have been achieved using a prototype for a live coding system, which is specifically designed for classical musicians.

Current literature and technologies have been studied to think of potential target users and their skills and needs. Based on this, the prototype has been designed (both from HCI and programming language design perspectives), implemented, and evaluated by classical musicians.

Results show that experienced composers may benefit the most from live coding systems, provided they offer enough expressiveness, responsiveness and feedback, and that their grammar is aligned with musical language.

Keywords: Live coding, Classical music, HCI, Human-computer interaction, Usability, Improvisation, Composition, Algorithmic composition

Research Question

Western-based music creation largely relies on pattern transformation. Therefore, programming algorithmic nature seems to match classical musicians mindset. The hypothesis proposed is that classically-trained musicians use conventional instruments to improvise and compose music, but they could benefit from live coding. However, current live coding programming languages are more oriented towards either experienced programmers or musicians performing electronic music. Therefore, this research tries to address the following problem: How can live coding be used in classical music?

Aims and Objectives

Such problem exposes a series of goals to fulfil.

- Address how live coding can be used in traditional composition: improvisation and programming share things in common (abstraction, structuring, immediateness), but classical musicians may not feel comfortable programming. Live coding systems should reduce these difficulties while helping them to create music more efficiency.

- Understand how classical musicians feel when using live coding: to make sure the usability of these systems is maximized for them.

In order to do this, a prototype for a live coding system has been developed that allows musicians to create music easily. The system notation has been designed to be as similar to solfége notation as possible. Also, it combines code typing with MIDI input, to provide a balance between algorithmic music generation and traditional instrument playing (which classical musicians are more familiar with). This prototype has been evaluated with users using qualitative methods to make sure that it is well suited for their specific needs.

Motivation

Live coding is becoming increasingly popular, with new languages being released every year. Most of these languages are designed for sound synthesis and electronic music. Because of this, this project analyzes how live coding could be used in classical music. The hypothesis for this project has been tested with a high fidelity prototype of a live coding system (BachTracking), based on preliminary user feedback and a study of current live coding programming languages.

Background and related work

Live coding programming languages

Several programming languages have been described in this section, each highlighting different aspects relevant to live coding for classical music. SuperCollider (Wilson et al. 2011) focuses on sound, has an object-oriented and decoupled architecture, and its OSC compatibility makes it the basis of other languages. One of them is SonicPi, designed for school students (Aaron 2016), by providing intuitive UI and documentation. TidalCycles controls SuperCollider via MIDI and OSC and relies on Laurie Spiegel's pattern transformations (Spiegel 1981) to create new music. It is functional like Extempore (Sorensen and Gardner 2017), which grants live low-level audio programming (Sorensen 2014). Other languages like Max and Pure Data (Zimmer 2007) use visual paradigms; similarly, OpenMusic (Bresson et al. 2010) includes graphical score notation. Orca (GitHub 2019) is visual and text-based, and it uses compact base-36 notation. Gibber (Roberts et al. 2015) and Serialist (Github 2016) explore abstractions that feel more familiar to musicians (e.g. a score).

HCI perspective

Magnusson (2019) states that computers are symbolically controlled, but physically actuated; this can pose UI mapping problems. Pane et al. (2002) suggest that "idiomatic" syntax improves programmers learning process, and so special characters should be avoided whenever possible. Wanderley and Orio (2002) propose that usable electronic instruments should be easy to learn, explore and modify, and they should allow precise timing.

Music composition techniques

Like programming, composition exploits the concept of transforming small music fragments. Counterpoint (Encyclopedia Britannica 2019; Jackson 2013) follows this philosophy, using three basic transforma-

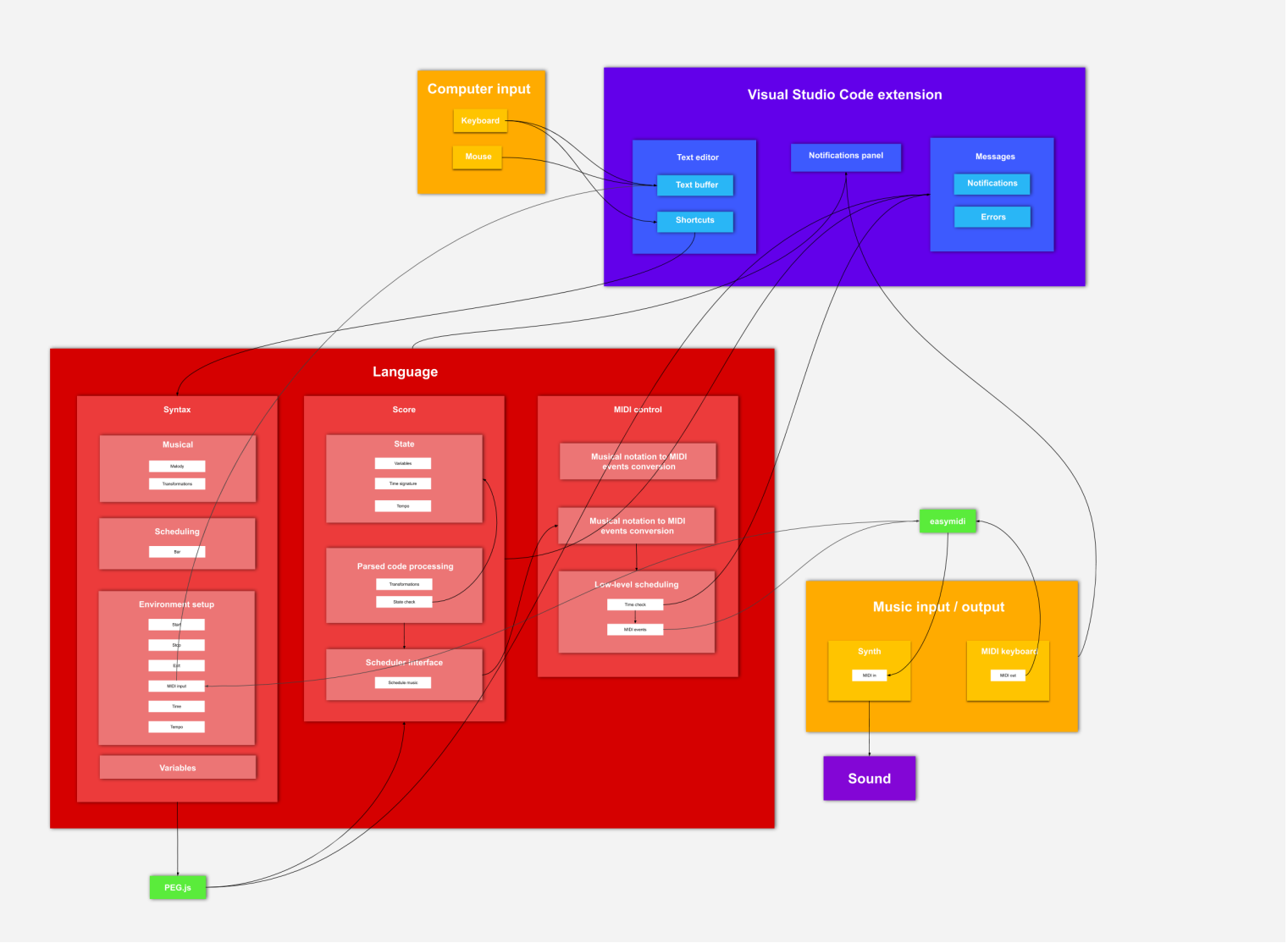


Figure 1: System overview

tions: inversion, augmentation/diminution, and retrograde. Based on counterpoint, serialism (Forte 1973) abstracts notes as numbers, sets and vectors of distances (intervals). Spiegel (1981) proposed similar techniques, specifically focusing on pattern transformation.

Design

Target users

Table 1.1 shows classical musicians can be divided into:

User	Music Expertise	Coding Expertise	Music Abstraction
Composer	High	Low	High
Player	High	Low	Intermediate
Student	Intermediate	Intermediate	Low

Table 1.1

A system that maximizes usability for these users should take their specific skills and needs into account (for instance, it should be possible to write music either note by note or using more abstract structures).

Language

Figure 1 provides an overview of the system. It offers the metaphor of an editor, a score and instruments. The score can be updated and read sequentially. Reading the score sends MIDI to instruments (keyboards, VSTs...)

The grammar (Figure 2) tries to be as similar to natural language as possible. Transformations can process melodies, variables or transformations of any of those. They have been inspired by counterpoint and Spiegel's work: transposition, inversion, retrograde, retrograde inverse, join and mirror. Parenthesis are not required, and shorthand names are available.

User interface

Figure 3 shows the UI proposed for the prototype. The text editor allows users to edit text without modifying the score. Errors are displayed in red, to call the user's attention. Keyboard shortcuts follow consistency whenever possible (Ctrl-S Starts reading, Ctrl-Shift-S stops reading).

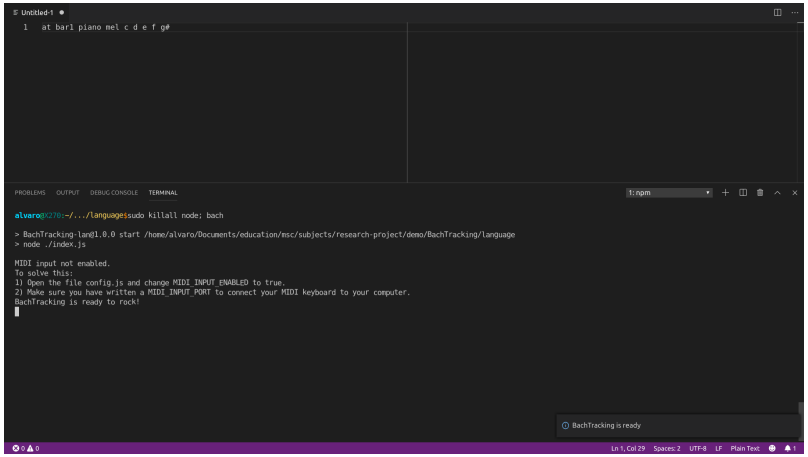
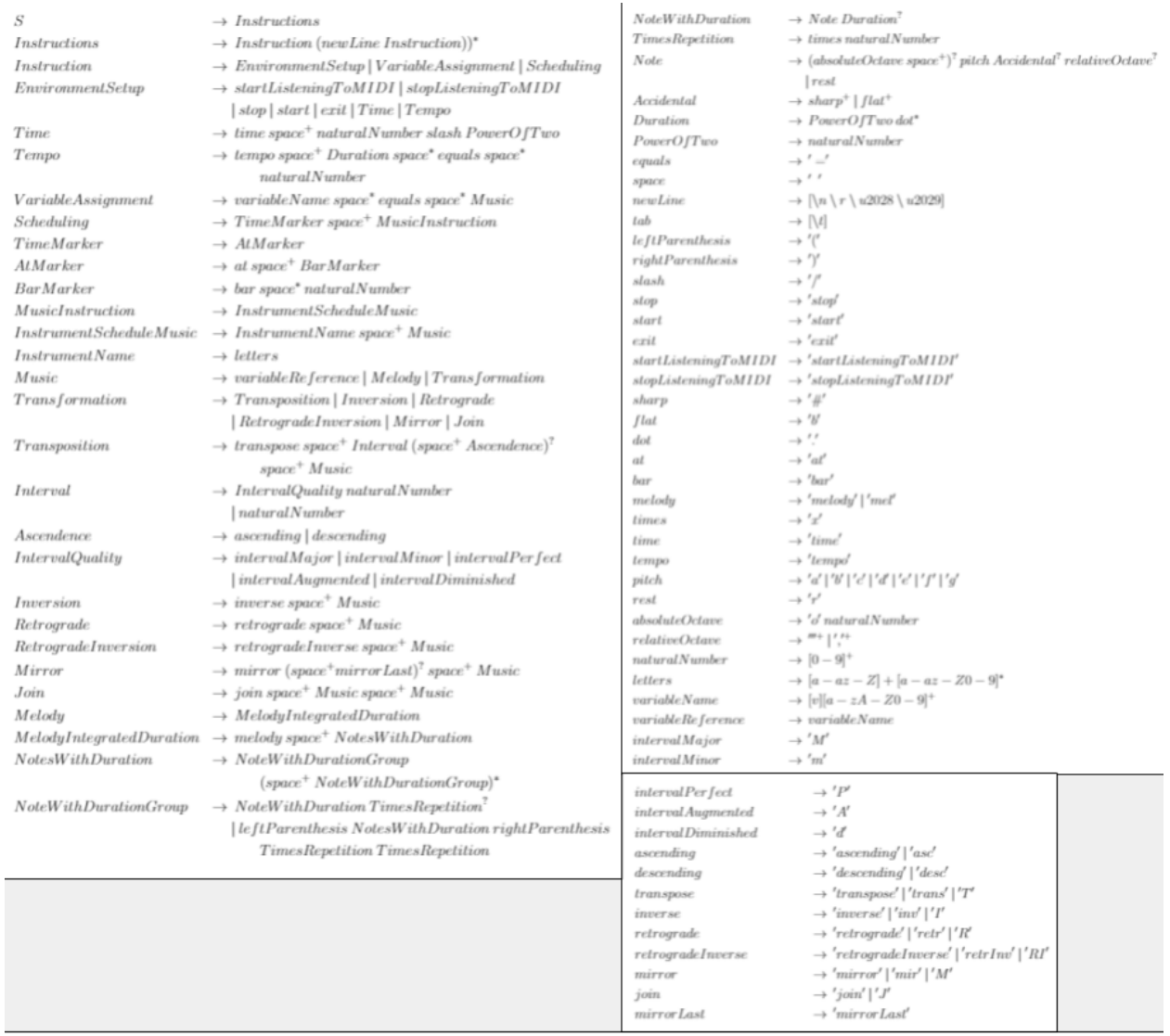


Figure 3: UI Layout

MIDI note input is available as well, inspired by music notation software (Avid 2018). Input is activated with a keyboard shortcut, and it writes notes as plain text (code) in the text editor, so they can be manually edited if needed.



$$
\begin{array}{ll}
S & \rightarrow Instructions \\
Instructions & \rightarrow Instruction\ (newLine\ Instruction))^{*} \\
Instruction & \rightarrow EnvironmentSetup \mid VariableAssignment \mid Scheduling \\
EnvironmentSetup & \rightarrow startListeningToMIDI \mid stopListeningToMIDI \\
 & \quad \mid stop \mid start \mid exit \mid Time \mid Tempo \\
Time & \rightarrow time\ space^{+}\ naturalNumber\ slash\ PowerOfTwo \\
Tempo & \rightarrow tempo\ space^{+}\ Duration\ space^{*}\ equals\ space^{*} \\
 & \quad naturalNumber \\
VariableAssignment & \rightarrow variableName\ space^{*}\ equals\ space^{*}\ Music \\
Scheduling & \rightarrow TimeMarker\ space^{+}\ MusicInstruction \\
TimeMarker & \rightarrow AtMarker \\
AtMarker & \rightarrow at\ space^{+}\ BarMarker \\
BarMarker & \rightarrow bar\ space^{*}\ naturalNumber \\
MusicInstruction & \rightarrow InstrumentScheduleMusic \\
InstrumentScheduleMusic & \rightarrow InstrumentName\ space^{+}\ Music \\
InstrumentName & \rightarrow letters \\
Music & \rightarrow variableReference \mid Melody \mid Transformation \\
Transformation & \rightarrow Transposition \mid Inversion \mid Retrograde \\
 & \quad \mid RetrogradeInversion \mid Mirror \mid Join \\
Transposition & \rightarrow transpose\ space^{+}\ Interval\ (space^{+}\ Ascendence)^{?} \\
 & \quad space^{+}\ Music \\
Interval & \rightarrow IntervalQuality\ naturalNumber \\
 & \quad \mid naturalNumber \\
Ascendence & \rightarrow ascending \mid descending \\
IntervalQuality & \rightarrow intervalMajor \mid intervalMinor \mid intervalPerfect \\
 & \quad \mid intervalAugmented \mid intervalDiminished \\
Inversion & \rightarrow inverse\ space^{+}\ Music \\
Retrograde & \rightarrow retrograde\ space^{+}\ Music \\
RetrogradeInversion & \rightarrow retrogradeInverse\ space^{+}\ Music \\
Mirror & \rightarrow mirror\ (space^{+} mirrorLast)^{?}\ space^{+}\ Music \\
Join & \rightarrow join\ space^{+}\ Music\ space^{+}\ Music \\
Melody & \rightarrow MelodyIntegratedDuration \\
MelodyIntegratedDuration & \rightarrow melody\ space^{+}\ NotesWithDuration \\
NotesWithDuration & \rightarrow NoteWithDurationGroup \\
 & \quad (space^{+}\ NoteWithDurationGroup)^{*} \\
NoteWithDurationGroup & \rightarrow NoteWithDuration\ TimesRepetition^{?} \\
 & \quad \mid leftParenthesis\ NotesWithDuration\ rightParenthesis \\
 & \quad TimesRepetition\ TimesRepetition \\
NoteWithDuration & \rightarrow Note\ Duration^{?} \\
TimesRepetition & \rightarrow times\ naturalNumber \\
Note & \rightarrow (absoluteOctave\ space^{+})^{?}\ pitch\ Accidental^{?}\ relativeOctave^{?} \\
 & \quad \mid rest \\
Accidental & \rightarrow sharp^{+} \mid flat^{+} \\
Duration & \rightarrow PowerOfTwo\ dot^{*} \\
PowerOfTwo & \rightarrow naturalNumber \\
equals & \rightarrow \text{'\_'} \\
space & \rightarrow \text{' '} \\
newLine & \rightarrow [\backslash n \setminus r \setminus u2028 \setminus u2029] \\
tab & \rightarrow [\backslash t] \\
leftParenthesis & \rightarrow \text{'('} \\
rightParenthesis & \rightarrow \text{')'} \\
slash & \rightarrow \text{'/'} \\
stop & \rightarrow \text{'stop'} \\
start & \rightarrow \text{'start'} \\
exit & \rightarrow \text{'exit'} \\
startListeningToMIDI & \rightarrow \text{'startListeningToMIDI'} \\
stopListeningToMIDI & \rightarrow \text{'stopListeningToMIDI'} \\
sharp & \rightarrow \text{'\#'} \\
flat & \rightarrow \text{'b'} \\
dot & \rightarrow \text{'.'} \\
at & \rightarrow \text{'at'} \\
bar & \rightarrow \text{'bar'} \\
melody & \rightarrow \text{'melody'} \mid \text{'mel'} \\
times & \rightarrow \text{'x'} \\
time & \rightarrow \text{'time'} \\
tempo & \rightarrow \text{'tempo'} \\
pitch & \rightarrow \text{'a'} \mid \text{'b'} \mid \text{'c'} \mid \text{'d'} \mid \text{'e'} \mid \text{'f'} \mid \text{'g'} \\
rest & \rightarrow \text{'r'} \\
absoluteOctave & \rightarrow \text{'o'}\ naturalNumber \\
relativeOctave & \rightarrow \text{'}^{+}\text{'} \mid \text{','}^{+} \\
naturalNumber & \rightarrow [0-9]^{+} \\
letters & \rightarrow [a-az-Z] + [a-az-Z0-9]^{*} \\
variableName & \rightarrow [v][a-zA-Z0-9]^{+} \\
variableReference & \rightarrow variableName \\
intervalMajor & \rightarrow \text{'M'} \\
intervalMinor & \rightarrow \text{'m'} \\
intervalPerfect & \rightarrow \text{'P'} \\
intervalAugmented & \rightarrow \text{'A'} \\
intervalDiminished & \rightarrow \text{'d'} \\
ascending & \rightarrow \text{'ascending'} \mid \text{'asc'} \\
descending & \rightarrow \text{'descending'} \mid \text{'desc'} \\
transpose & \rightarrow \text{'transpose'} \mid \text{'trans'} \mid \text{'T'} \\
inverse & \rightarrow \text{'inverse'} \mid \text{'inv'} \mid \text{'I'} \\
retrograde & \rightarrow \text{'retrograde'} \mid \text{'retr'} \mid \text{'R'} \\
retrogradeInverse & \rightarrow \text{'retrogradeInverse'} \mid \text{'retrInv'} \mid \text{'RI'} \\
mirror & \rightarrow \text{'mirror'} \mid \text{'mir'} \mid \text{'M'} \\
join & \rightarrow \text{'join'} \mid \text{'J'} \\
mirrorLast & \rightarrow \text{'mirrorLast'}
\end{array}
$$

Figure 2: BachTracking grammar

Implementation

Language

In this system [1], the language works as a server that listens to text coming from the UI. This text is parsed and sent to a score object. The score controls a MIDI scheduler, which communicates with any connected instruments. It also handles MIDI note input (Figure 4).

User interface

The UI has been developed as a Visual Studio Code extension, given its active support (as of 2019). The UI (Figure 3) has three main software-based UI elements[2]: text editor, notifications panel, and information/error messages.

Evaluation

Qualitative user evaluation was applied, and it was split into two phases:

- Online interviews: users followed semi-structured interviews about their experience making music with technology and live coding. This was done first to improve the prototype before testing it.

- In-lab user evaluation: users were asked to read a tutorial explaining how to use BachTracking during one day, before trying the system. After that, I gave them a small crash course on how to use BachTracking with the MIDI keyboard (this was done in my house, using a laptop and a MIDI keyboard that had been tested beforehand). Then they were asked to try the system by playing some music with BachTracking. After this, they followed a semi-structured interview to get feedback about their experience creating music with this prototype.

Online interview

Demographics were balanced in this case. 3 out of 6 participants were professional composers (in their mid-twenties) with studies at graduate level from Kings College London, Conservatorium van Amsterdam (Netherlands) and Katarina Gurska music academy (Spain). Two participants were intermediate classical piano students (18-22 years old) at the Professional Music Conservatory of Getafe (Spain). The remaining participant was a music hobbyist in their late fifties with intermediate studies in clarinet, who played in brass music bands in Pinto and Getafe (Spain).

No participants knew about live coding previously, but a few had programmed before. Composers were used to music technologies (VST's, DAW's...), but students mostly used recording and notation software as their music tools; they explicitly stated they preferred acoustic instruments. Participants were asked how they create music, and they described it as first relying on muscle memory and intuition, and then structuring the idea in a more cerebral way.

Online interviews were conducted after starting to develop the prototype and before testing it with in-lab user evaluation, thus helping to refine features and functionality.

In-lab user evaluation

Composers used all of the language's available functions, and they even felt the freedom to experiment and force the system by using notes with an incredibly small duration (thus creating blazing fast

[1] The code can be downloaded from `https://mega.nz/$#$F!RbpizSxB!EOIbhFjPXB8QbwqzrsZGSw` (notice it has only been tested in Ubuntu Studio).

[2] There are also physical UI elements such as the MIDI keyboard, keyboard or mouse

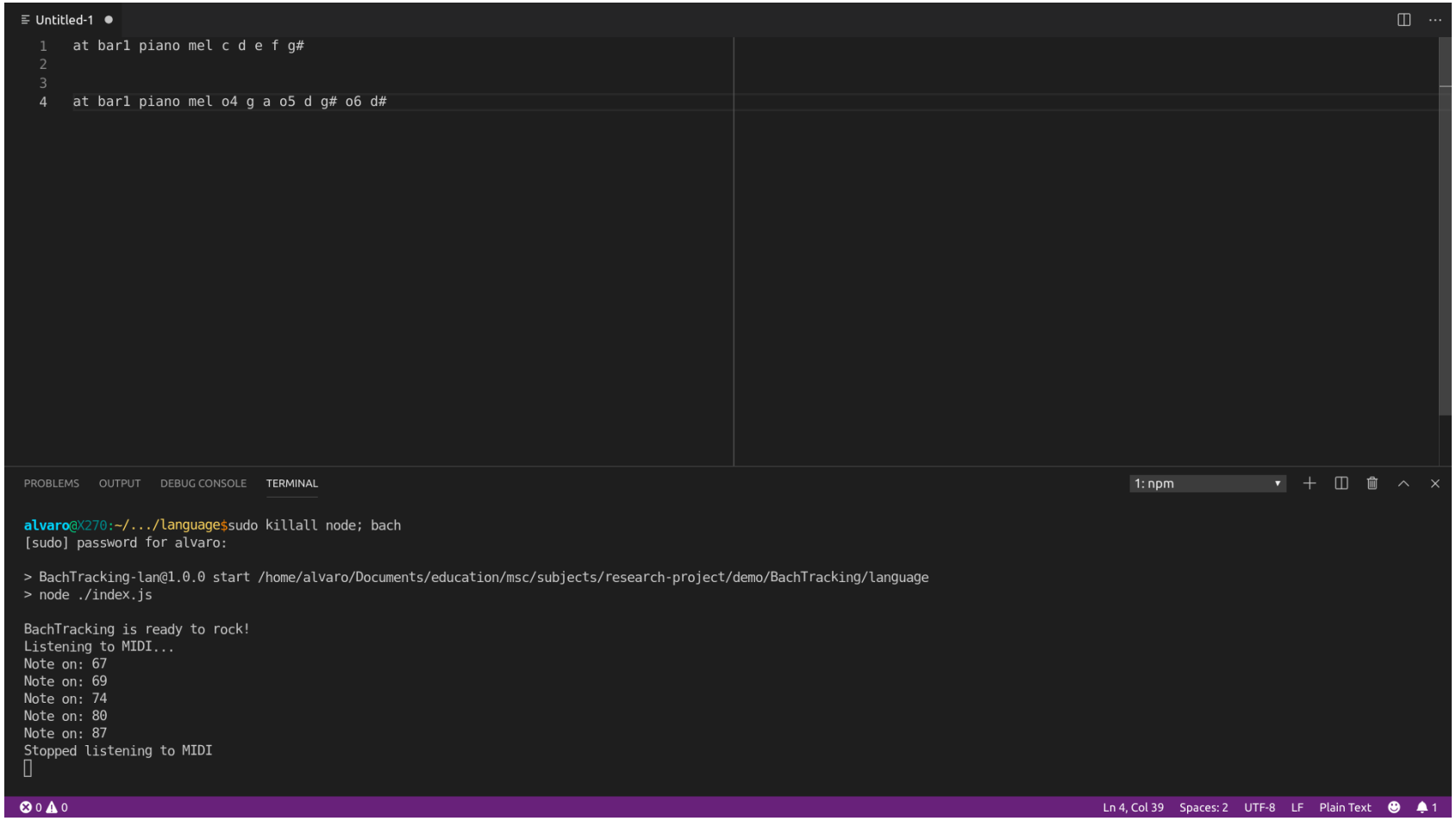


Figure 4: MIDI note input

sequences of notes). On the other hand, music students used MIDI input for the most part of their performances.

Participants found some aspects of the system confusing, like not seeing the bar number printed on the screen, and having to write the octave number before the note duration.

Suggestions included having access to a live graphical score, a graphical preview of the code to be executed, and accidentals omission. The possibility of writing code with MIDI input was proposed to users to see if they would be interested in that feature; composers and players thought it could be powerful, but students found it confusing.

Most participants composed rather than improvising live, perhaps due to their classical background, or to insufficient practice time.

Conclusions

Key results and significance

The following main points can be extracted from this study:

- Expressiveness of the initial musical motif (with traditional instrument input), conciseness and transformation composability can improve live coding usability for classical musicians.
- Classical musicians may prefer using live coding for fast prototyping when writing music.
- Feedback and situation awareness are crucial to provide the responsiveness classical musicians find in the instruments they are used to play.
- User-centered design may help to create live coding programming languages that best adapt to the musicians who are going to use it.

Future work

This project could benefit from more extensive user evaluation, which could point out features to be included in the system, or usability errors that should be corrected. Increasing training time would allow participants to feel more comfortable improvising with BachTracking. Jazz musicians could be considered as participants in the future as well. The language could incorporate more features: scales, tempo transformation, harmony (voice leading), microtonality, larger structural transformations, form... Developing two syntax modes (large/understandable, and concise/ergonomic) could make the system more flexible, and suitable to both compose and improvise.

References

Aaron, S. (2016). Sonic Pi – performance in education, technology and art. International Journal of Performance Arts and Digital Media, 12(2), pp.171-178.

Avid. (2018). Sibelius Reference Guide version 2018.1. [online] Available at: `http://resources.avid.com/SupportFiles/Sibelius/2018.1/Sibelius_2018.1_Reference_Guide.pdf` [Accessed 18 Apr. 2019].

BBC. (2019). Melody - Edexcel - Revision 6 - GCSE Music - BBC Bitesize. [online] Available at: https://www.bbc.com/bitesize/guides/zwj2jty/revision/6 [Accessed 9 Apr. 2019].

Bresson, J., Agon, C., and Assayag, G. (2010). OpenMusic – visual programming environment for music composition, analysis and research. ACM MultiMedia (MM'11).

Encyclopedia Britannica. (2019). Inversion | music. [online] Available at: `https://www.britannica.com/art/inversion-music` [Accessed 13 Apr. 2019].

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

GitHub. (2016). serialist. [online] Available at: https://github.com/irritant/serialist [Accessed 5 Mar. 2019].

GitHub. (2019). Orca: Esoteric Programming Language. [online] Available at: https://github.com/hundredrabbits/Orca [Accessed 5 Mar. 2019].

Jackson, R. (2013). Counterpoint | music. [online] Encyclopedia Britannica. Available at: https://www.britannica.com/art/counterpoint-music [Accessed 11 Apr. 2019].

Magnusson, T. (2019). Sonic writing. New York, NY: Bloomsbury Publications.

Pane, J., Myers, B., and Miller, L. (2002). Using HCI techniques to design a more usable programming system. Proceedings IEEE 2002 Symposia on HumanCentric Computing Languages and Environments. IEEE, pp. 198–206.

Roberts, C., Wright, M., and Kuchera-Morin, J. (2015). Music program-ming in gibber. ICMC.

Sorensen, A. (2014). GOTO 2014 Programming In Time - Live Coding for Creative Performances

Andrew Sorensen. [online] Youtube. Available at: https://www.youtube.com/watch?v=Sg2BjFQnr9s [Accessed 20 Apr. 2019].

Sorensen, A., and Gardner, H. (2017). Systems level liveness with extempore. Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. ACM, pp. 214–228.

Spiegel, L. (1981). Manipulations of musical patterns. Proceedings of the Sym-posium on Small Computers and the Arts, pp. 19–22.

SuperCollider Docs. (2019). 02. First Steps | SuperCollider 3.10.3 Help. [online] Available at: http://doc.sccode.org/Tutorials/Getting-Started/02-First-Steps.html [Accessed 23 Apr. 2019].

Wanderley, M. and Orio, N. (2002). Evaluation of input devices for musical expression: Borrowing tools from hci". Computer Music Journal 26.3, pp. 62–76.

Wilson, S., Collins, N. and Cottle, D. (2011). The SuperCollider book. Cambridge, Mass.: MIT Press.

Zimmer, F. (2007). Bang. Hofheim: Wolke, p.133.