

Live Coding on a Modular Synthesizer

First Author

Affiliation1

author1@leeds.ac.uk

ABSTRACT

Code is a hardware module for the 3dPdModular system that enables writing code live in Python using the Pyo package on a hardware modular synthesizer. It provides a minimal interface for displaying the code, and four potentiometers implemented as Pyo objects that can be input in any other Pyo class for controlling parameters like frequency, amplitude, etc. Its aim is to combine live coding on a computer with live patching a modular synthesizer, all in the same system. By introducing the Pyo package for Python to the 3dPdModular system, it expands the functionalities of the system, as it provides a wide range of utilities, including unit generators, filters, effects, phase vocoding, FFT analysis, and others, plus it provides the flexibility of improvising with computer code. By integrating a live coding interface to a modular synthesizer, the coder can utilize features from the tangible interface of the modular synthesizer to enable making sound with minimal writing of code, since all the hardware of a 3dPdModular setup can be used in combination with this module. The tangibility of the synthesizer also provides an intuitive way to input values to code and fine tune processes.

1. INTRODUCTION

Live coding on a computer and live patching a modular synthesizer are two practices of electronic music improvisation that share a lot in common (Hutchins 2015, 147). Even though each follows a completely different paradigm, their approach is very similar in that they both aim to change the connections within a system during a performance. When live patching, the circuitry of the modular synthesizer is changed. When live coding, the DSP graph of a programming environment is changed. They both result in re-routing audio or control signals between nodes. The connection between live coding and live patching can be highlighted with the Automatonism system (Eriksson 2019), a set of Pure Data abstractions created with the modular synthesizer approach. Despite their similarities, these two practices differ in certain aspects. Live patching provides a tangible interface with dials and switches, whereas live coding on a computer depends mostly on the computer keyboard, and, occasionally, the mouse. Another difference is that live patching is limited by the available hardware during a performance. In live coding, this limitation is imposed by the available functionalities of the used environment, although this can be enriched by writing functions and classes live, and the CPU of the machine (White 2019, 71).

Aiming to combine these two practices, I created the *Code* module. This is a hardware module that functions only within the 3dPdModular system (3dPdModular 2021). The inspiration for this module was the *Teletype* Eurorack module (Crabtree and Kelli 2021), and the *Pyo* software module for DSP in Python (Bélanger 2016). The *Teletype* enables live coding in a Eurorack system. *Pyo* includes an extensive library of classes for various DSP tasks, from unit generators, to filters, Phase Vcoders, and others. It “allows the creation of sophisticated signal processing chains with all the benefits of a mature and widely used general programming language” (Bélanger 2016, 1215). With these two starting points I created a module that enables writing Python code live in the 3dPdModular system.

The aim of the *Code* module is twofold. On one hand, it aims to augment the hardware of the 3dPdModular system by providing functionalities not present in the other modules by introducing all the *Pyo* classes to the system. Be it unit generators, filters, FFT processes, or effects like distortion, delay, reverb, and others, *Code* can introduce a wide range of features that might be missing. On the other hand, it aims to enable the

coder to make sound without writing too many lines of code. This is achieved by combining the hardware of the synthesizer system with the coding in a live session. For example, the coder can create a simple sequencer, and patch its outputs to oscillator modules, thus start making sound within a short time from the beginning of the session.

Code also supports projection of live coding sessions, following the “show us your screens” paradigm. Projecting the code is done with an openFrameworks application that simulates a Python interpreter running on a computer terminal, with the “>>>” and “...” prompts displayed for single lines and loop, function, and class definitions respectively.



Figure 1. The *Code* module.

2. RELATED WORK

There is a substantial body of work combining live coding with modular synthesizers, or more broadly computer code with hardware interfaces. Diapoulis and Zannos created a hardware interface that controls SuperCollider code (Diapoulis and Zannos 2014), where the authors mention that their experiments open new ways to approach live coding (Diapoulis and Zannos 2014, 442). Smith and Lawson created an Arduino device called “The Force” that bridges a live coding environment for writing OpenGL shaders with a modular synthesizer (Smith and Lawson 2016). This device receives OSC messages created by parsing the text of OpenGL shaders. These messages are then converted to control signals via a 12-bit DAC. It also functions in the opposite direction, where it receives CV signals from a modular, converts it to digital values, and sends them to a computer over OSC. Haddad and Paradiso created a Eurorack module for

connecting various synthesizers remotely over the Internet (Haddad and Paradiso 2019). This module can be used with live coding by writing Javascript code and generate audio signals that are transferred to the module. Paradiso created a hardware module that connects to a browser based interface and can be controlled by users over the internet (Mayton et al. 2012). He connects this module to his massive modular synthesizer, so users who load the web client remotely, can control some of the parameters of the entire synthesizer, and they can also listen to the resulting sound, as it is being streamed over the internet. Aaron et al. connect their computers during a live coding session through a monome, which sends various events to any computer in the network (Aaron et al. 2011). A notable mention is Eldridge and Kiefer’s self-resonating feedback cello (Eldridge and Kiefer 2017). This cello has a speaker attached to its back and a transducer to its front, as well as pickups for capturing the string vibrations. This instrument can be combined with live coding in SuperCollider.

Other related projects include commercial modules for the Eurorack system that can be programmed by the user in a variety of programming languages. The *Bela Salt* (Bela 2021) is a module based on the Bela Cape for the Beaglebone Black mini-computer. It can be programmed in a number of languages, including Pure Data, SuperCollider, C++ and Python. The *OWL Modular* (Rebel Technology 2018) is a module similar to the *Bela Salt* that can be programmed in Pure Data, C++, Faust, and Max Gen patches. The *QU-Bit Nebulae* (QuBit Electronix 2021) is another programmable module, although it differs slightly from the two previous ones in that it is mainly a granulator and its front panel is labeled based on this functionality, whereas the *Bela Salt* and *OWL* have a very generic labeling on their panels. Still, the *Nebulae* can be programmed in Pure Data, SuperCollider, and Csound, for functionalities other than granular synthesis.

The three modules mentioned above are similar to the *Code* module in that they can be programmed by the user, the same way *Code* can load user defined Python scripts for various tasks. None of these modules though supports live coding. The only Eurorack module that supports live coding is the *Teletype*. This module is very similar to *Code* as it can load predefined scripts, but can also be programmed on the fly. It uses a bespoke language and can receive trigger signals and output trigger and CV signals. In contrast to *Code*, it does not support projecting the code in a live coding session, a convention within the live coding community.

3. THE 3DPDMODULAR SYSTEM

The *Code* module is part of the 3dPdModular system. This is an integrated modular system running on a single Raspberry Pi for all audio computations, and a single Teensy micro-controller for controlling the tangible interface. Exceptions are modules with on-board displays that use an additional Teensy to control their display. This system was inspired by the rePatcher by Open Music Labs. Apart from the Arduino code that runs on the Teensy, and does not change, the rest of the system is programmed in Pure Data. Every hardware module has a respective Pd abstraction that is loaded in the main patch of the system on boot. Being integrated, this system needs specific hardware to send and receive control signals to/from other hardware, like the Eurorack system. Nevertheless, there is a number of unique characteristics among modular synthesizer systems that are found in 3dPdModular. Namely, this system can save patches in textual form, with the possibility of storing them in a bespoke mobile app, called “Patches”. The patch information contains the various connections within the system in the following form:

```
TRAPEZOID:o1->VCA_MIXER:i2
```

This means that the first output of the module called “TRAPEZOID” connects to the second input of the module called “VCA_MIXER”. The application also contains information on the positions of the potentiometers of the modules in a patch. Figure 2 demonstrates the potentiometer positions of the module “TRAPEZOID”. The user can refer to this information to recreate a saved patch, without needing to take any sort of notes manually.

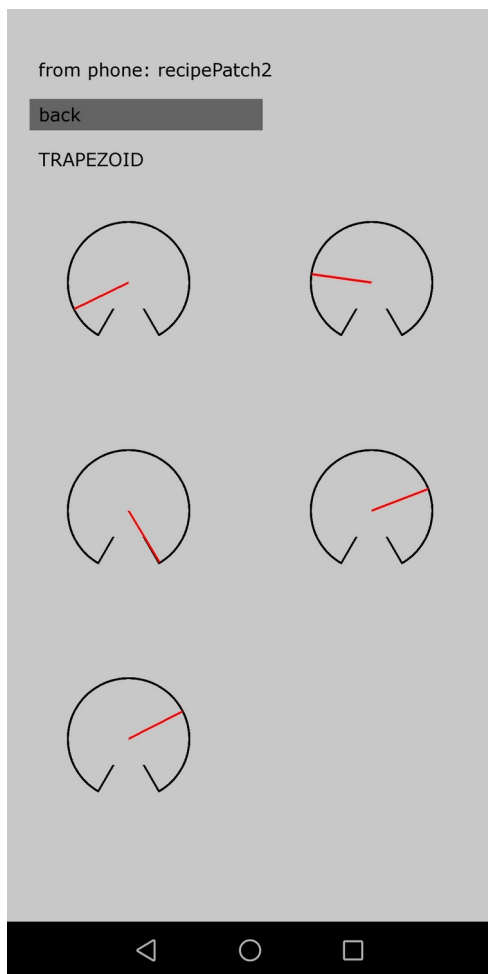


Figure 2. Screenshot of the *Patches* mobile application of the 3dPdmodular system.

Another characteristic is cross-fading between different patches. The user can disable the patching update, change the wiring of the synth, and re-enable the patching update by setting the cross-fade time with a potentiometer. The synthesizer will cross-fade smoothly from the previous patch to the current one. Module cloning is also possible via the CLONE module. This module clones the software of any number of networked modules connected to it. The hardware interface of the modules that are cloned is used to control any clone instance, with the CLONE module determining which clone instance is currently controlled.

Metadata concerning the type of the signals is another feature of this system. This is used to distinguish audio from CV, triggers, gates, and whether a signal is received through a VCA module. This information is used to determine the functionality of a module in a given input. For example, a VCA will use a linear curve in its control potentiometers for CV signals, and a logarithmic one for audio signals. Automating the switching between the two potentiometer profiles minimizes the hardware on the module's panel, and is less error prone. Lastly, since the entire system runs on a Raspberry Pi, the synthesizer can be connected to the Internet, and any module can exploit this in many different ways, *Code* being one of them.

4. THE CREATIVE POTENTIAL

In contrast to live coding programming languages that aim to facilitate the creation of rhythmic and melodic patterns, like the Ixi Lang (Magnusson 2011), the Sonic Pi (Aaron and Blackwell 2013), Tidal Cycles

(McLean 2021) and FoxDot (Kirkbride 2021), *Code* aims at a wider range of music styles by introducing the Python module Pyo, intact. It does not abstract code, but aims to achieve fast prototyping by combining software with hardware. Even with the aesthetic restrictions imposed by the hardware setup, the user is provided with an extended palette of utilities that can supplement their system.

Code also integrates a minimal hardware interface that connects to the live coding session intuitively. This consists of four potentiometers and can be loaded in the Python code via the Pot() class. This class contains methods for defining the profile of a potentiometer, by setting arbitrary value ranges and curvature coefficients. For example, the user can type the following:

```
pots = [Pots(i) for i in range(4)]
pots[0].setRange(200)
pots[0].setExp(2)
```

This will create a list of four Pot() objects and set the range of the first one to go from zero to two hundred, and an exponential curve, by raising the pot's values to the power of two, before they are scaled. The range and exponent functions can be combined in one function with rangeExp(float, float). The values of these objects can be input to any method of the classes provided by Pyo. This way the user can easily and intuitively tune frequencies and other elements of the sound. Once a satisfactory result is achieved, the potentiometer values can be stored in the methods they control as constants, and can be unbound for use elsewhere. This is done with the following example lines:

```
vals = potvals(pots[0], pots[1])
meths = [sin.setFreq, amp.setVal]
setMeths(vals, meths)
```

The lines above will store the values of the first two potentiometers and set their current values as constants to the setFreq() method of an object called sin, and the setVal() method of an object call amp. After the last line, the two potentiometers will be unbound and will not affect the two methods any longer.

Lastly, Python provides a wide range of native and external modules for various tasks, whether musical or extra-musical. Including tasks such as machine learning, data mining, geolocating, web browsing, and many others, to your DSP algorithms should be a trivial task, as it all happens in the same programming language.

5. CHALLENGES

The language used to live code with the *Code* module is Python. This choice was made for a few different reasons. First and foremost, Python is the main textual programming language I personally use. Additionally, the developer of Pyo has embedded Python with his module in various different programming environments, including Pure Data, the programming language of the 3dPdModular system. In Pd, Pyo is integrated with the [pyo~] external object. This object facilitated the creation of the *Code* module greatly, as most of the hard work for importing Python with DSP functionalities into Pd was already done. An issue was that [pyo~] does not support live coding, but it is built to load existing Python scripts into the object. There is a minimal support for creating objects, setting values to variables and calling functions, but that does not account for live coding. What [pyo~] does offer though is loading Python scripts on top of others. For example, the first script loaded to the object can be the following:

```
a = Sine().out(0)
```

This will create a sine wave oscillator object with the default frequency of 1000Hz, output through the first outlet of [pyo~]. The second script can be the following:

```
a.setFreq(200)
```

This script will set the frequency of the sine wave object of the previous script to 200Hz. This feature enabled me to store small Python scripts whenever a line, function, loop or class is completed, and load that to the object, on top of all the previous scripts. This is achieved by capturing the keyboard connected to the Pi via a Python script, accumulating the typed characters in a string, and storing a file with the .py extension and an incrementing index in its name whenever the return key is hit once for single lines, and twice for functions, loops, and classes. When a file is saved, the Python script sends an OSC message to the Pd abstraction of the *Code* module with the index of that file and [pyo~] loads the respective script. This way the user can write Python code in a similar way as in a Python interpreter running in a terminal of a computer.

Another challenge was the small size of the on-board display. This is used to display the code to the user. Apart from its small size, the maximum number of characters that fit in one line is 20. Some Pyo classes as well as their variables have long names. The following line is an example:

```
s1=SineLoop(freq=100,feedback=.1)
```

This line creates a SineLoop object and contains 33 characters. When typed in the *Code* module, inevitably, it will be split in two. That might not sound like a serious issue, but the display has space for only five lines on code, splitting most of the lines is not very practical. To remedy this, I created alias classes for many of the Pyo classes, with shorter names for the class itself and its variables. For example, the previous line of code is translated to the following:

```
s1=SL(fr=100,fb=.1)
```

The last challenge was to provide a method to print information to the standard output. This has been currently applied to the projection of the code only, and is not included in the on-board display of the module. Since calling Python's print() function does not work with the [pyo~] object, a different approach was necessary. Printing to the standard output is done by sending what has to be printed via OSC from the Python code to the Pd patch, where the latter forwards it to the openFrameworks application. This mediation of Pd is reserved for a possible future development that will introduce this feature to the on-board display as well. It is also possible to split the screen of the standard output and print information to the right side of it. The two printing methods are sent at the OSC addresses "/print" and "/print2" respectively.

6. ISSUES

The main issue concerns the responsiveness of the on-board display. This is rather slow, as communication between the Teensy controlling the display and the main Teensy of the synthesizer is done with the I2C protocol, in between various other tasks, like reading all the potentiometers of the synthesizer, controlling shift registers for every module, and requesting data from other I2C slaves that might be present in a setup. The way the code of the system is currently written, makes the I2C communication, a protocol that is natively not known to be fast, rather slow. For a coder who types fast, this slow responsiveness can be problematic. Also, if the display has filled all five lines that it contains, when a new line is typed, the Teensy must move the displayed strings one position back in the array that holds the typed strings, clear the display, and write all strings one by one. This process adds time to the already slow responsiveness. A possible solution is to apply threads to the Arduino code of the main Teensy of the system, so that the I2C communication between it and the slave Teensies that control displays in some modules is isolated from the rest of the program.

The small size of the display mentioned in the Challenges section, leading to a set of alias classes with short names can render the code less self-explanatory and makes following the train of thought of the coder harder, in a live coding session. Writing FShift() instead of FreqShift(), or Pat() instead of Pattern() is

less self-explanatory. This issue can be addressed by using a larger display, but that requires a complete remake of the module's hardware and software.

The last issue is the feedback provided to the user when bad code is written. In a Python interpreter, when an error occurs – e.g. the user calls a function `func()` that has not yet been defined –, the computer provides the following feedback:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'func' is not defined
```

If such an error, or any error, occurs in a script loaded in the `[pyo~]` object, the only feedback provided is the following:

Bad code in file `<pythonized_line0.py>`

By adding one extra outlet to the object, I was able to retrieve the error code value in the Pd patch that uses the `[pyo~]` object. This value is sent over I2C to the Teensy controlling the on-board display, so the text “Bad code!” can be shown to the user. This information is also sent to the openFrameworks application so it can be printed in the projection of the code. This is shown in figure 3. This feedback though is very minimal. Retrieving the entire feedback returned by a Python interpreter is not a trivial task, but even if this is realized, the slow responsiveness of the display does not allow for such long strings to be sent from the Pd patch to the Teensy.

<pre>>>> eu.play() >>> latlng=gm.coords() >>> meta(0,dc()) >>> meta(1,dc()) >>> tf=TrgFun(ins[6],latlng[0].get) >>> ri=RandInt(max=8,add=8) >>> def newtap(): ... eu.setTaps(int(ri.get())) ... >>> pat=Pat(newtap,time=2) >>> Bad code! >>> pat.play() >>> Bad code! >>> pat=Pat(newtap,t=2) >>> pat.play() >>> dl=[Del(ins[i],dl=pots[0],fb=pots[1]) for i in range(2)] >>> dist=[Dist(dl[i],dr=pots[2]).out(i+4) for i i n range(2)] >>> fs=[FShift(dl[i],sh=pots[3]) for i in range(2)] >>> for i in range(2): ... dist[i].setIn(fs[i],ft=5) ... >>> meta(2,vca()) >>> █</pre>	<pre>St. Peter Port 49.4541677, -2.5497069 George Town 38.9076089, -77.072258499999999 Tripoli 32.8872094, 13.1913383 Yaren -0.54668569999999999, 166.9210913 Amman 31.9539494, 35.910635 Luxembourg 49.815273, 6.12958299999999999 Warsaw 52.2296756, 21.0122287 Cairo 30.0444196, 31.2357116 Oslo 59.9138688, 10.7522454</pre>
--	--

Figure 3. The openFrameworks application simulating a Python interpreter on a computer terminal. The “Bad code!” error message has been printed twice in the left side of the screen. In the right side latitude and longitude information retrieved from the Python Google Maps API are printed by being sent to the “/print2” OSC address.

Combining *Code* with other hardware systems, like the Eurorack, is not possible without additional hardware. Currently, two modules that bridge the gap between the 3dPdModular and the Eurorack systems

are being developed. Combined with these, *Code* can send and receive CV, triggers and gate signals to and from a Eurorack system. Combined with the *Communicate* module of the 3dPdModular system, *Code* can communicate with other computers via OSC. Without extra hardware though, *Code* is kind of isolated within the 3dPdModular system.

7. FUTURE WORK

Future plans for the *Code* module include supporting other programming languages. The top candidates are Lua, Javascript, and Scheme, as these languages have already been imported to Pd. Lua is the first among these three to be imported to Pure Data and there is a stable release of the [pdlua] object. Javascript and Scheme are imported through the [pdjs] and [s4pd] objects respectively. Both these objects are at a beta stage at the time of writing. When they reach a stable release I will definitely consider importing them to the *Code* module. All three objects run in the control domain and none of them supports audio signals. The 3dPdModular system outputs and receives audio signals in all modules. The two domains are interoperable, but this means that no unit generators can be produced using these languages the way they are imported to Pd, plus they cannot be sample-accurate. Still, there is a substantial level of flexibility they can provide to the 3dPdModular system.

The hardware of the module will possibly change to include a bigger on-board display, four additional potentiometers that will sum up to eight, four push-buttons, and four LEDs. The additional hardware, especially the push-buttons and the LEDs, will provide both greater flexibility but also better visual feedback to the user. LEDs are the basic element of visual feedback in modular synthesizers, and they can provide sufficient and very responsive visual feedback in certain occasions.

8. CONCLUSIONS

Code is a hardware module for the 3dPdModular system that enables live coding in Python within a modular synthesizer. By importing the Pyo module for DSP, a wide range of utilities is imported into the modular system. The aim is to expand the capabilities of a given system, by providing additional functionalities to those supported by the rest of the hardware modules of the system, and to enable the coder to produce sound quickly, without typing lots of lines of code, by combining live coding with live patching the rest of the modules. Its musical outlook is arbitrary and does not focus on specific genres or styles. By including a minimal hardware interface of four potentiometers, coders can easily import the tangibility of a hardware synthesizer into their live coding sets. Python's wide range of modules can be utilized within the *Code* module by simply importing them and calling their methods, importing Python's general-purpose nature to DSP algorithms. Finally, projection of the code is supported, following the convention of the live coding community.

REFERENCES

- 3dPdModular. 2021. "3dPdModular." 2021. <https://3dpdmodular.cc/>.
- Aaron, Samuel, and Alan F. Blackwell. 2013. "From Sonic Pi to Overtone: Creative Musical Experiences with Domain-Specific and Functional Languages." In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*, 35–46. FARM '13. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/2505341.2505346>.
- Aaron, Samuel, Alan Blackwell, Richard Hoadley, and Tim Regan. 2011. "A Principled Approach to Developing New Languages for Live Coding." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, 381–86. Oslo, Norway. <https://doi.org/10.5281/zenodo.1177935>.
- Bela. 2021. "Bela Salt." <https://shop.bela.io/collections/modular/products/salt>.

- Bélanger, Olivier. 2016. "Pyo, the Python DSP Toolbox." In *Proceedings of the 24th ACM International Conference on Multimedia*, 1214–17. MM '16. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/2964284.2973804>.
- Crabtree, Brian, and Cain Kelli. 2021. "Teletype." <https://monome.org/docs/teletype/>.
- Diapoulis, Georgios, and Ioannis Zannos. 2014. "Tangibility and Low-Level Live Coding." In *Proceedings of the International Computer Music Conference, ICMC*, 440–44. Athens, Greece.
- Eldridge, Alice, and Chris Kiefer. 2017. "Self-Resonating Feedback Cello: Interfacing Gestural and Generative Processes in Improvised Performance." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, 25–29. Copenhagen, Denmark: Aalborg University Copenhagen. <https://doi.org/10.5281/zenodo.1176157>.
- Eriksson, Johan. 2019. "Automatonism: Towards Dynamic Macro-Structure in Generative Music for Modular Synthesizers." PhD Thesis, Birmingham City University.
- Haddad, Don Derek, and Joe Paradiso. 2019. "The World Wide Web in an Analog Patchbay." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, edited by Marcelo Queiroz and Anna Xambó Sedó, 407–10. Porto Alegre, Brazil: UFRGS. <https://doi.org/10.5281/zenodo.3673013>.
- Hutchins, Charles Celeste. 2015. "Live Patch / Live Code." In *Proceedings of the First International Conference on Live Coding*, 147–51. Leeds, UK: ICSRiM, University of Leeds. <https://doi.org/10.5281/zenodo.19346>.
- Kirkbride, Ryan. 2021. "FoxDot." 2021. <https://foxdot.org/>.
- Magnusson, Thor. 2011. "The IXI Lang: A SuperCollider Parasite for Live Coding." In *Proceedings of the International Computer Music Conference, ICMC*. Huddersfield, UK.
- Mayton, Brian, Gershon Dublon, Nicholas Joliat, and Joseph A. Paradiso. 2012. "Patchwork: Multi-User Network Control of a Massive Modular Synthesizer." In *Proceedings of the International Conference on New Interfaces for Musical Expression*. Ann Arbor, Michigan: University of Michigan. <https://doi.org/10.5281/zenodo.1178345>.
- McLean, Alex. 2021. "Tidal Cycles." 2021. <https://tidalcycles.org/docs/>.
- QuBit Electronix. 2021. "QuBit Nebulae." 2021. <https://www.qubitelectronix.com/shop/nebulae>.
- Rebel Technology. 2018. "OWL Modular." 2018. <https://www.rebeltech.org/product/owl-modular/>.
- Smith, Ryan Ross, and Shawn Lawson. 2016. "Closing the Circuit: Live Coding the Modular Synth." In *Second International Conference on Live Coding*.
- White, Alex. 2019. "ANALOG ALGORITHMS: GENERATIVE COMPOSITION IN MODULAR SYNTHESIS." In *Proceedings of the ACMC 2019*, 68–73. Melbourne, Australia: Melbourne, Monash University.