Calcification and Hybrid Live-Coding

Avneesh Sarwate

ABSTRACT

This paper presents a design framework for building music performance systems that are both highly responsive and easily customizable during performance. We focus on systems that incorporate live-coding, the activity of writing code during the performance itself. We first discuss the merits and weakness of live-coding in isolation, and introduce the idea of "hybrid live-coding." We then introduce the ideas of "liveness" and "malleability" as they relate to musical interfaces, and then describe "calcification" - our framework for building interfaces that are both live and malleable. Next, after presenting case studies of the Tidal live-coding library and two other original live-coding libraries, we discuss hybrid live-coding environments and how calcification can help performs design custom interfaces in these environments. We conclude by positioning this paper as the start of a longer program of research into code-utilizing and malleable interfaces for musical performance

Author Keywords

Music Performance Interfaces; Domain Specific Languages; Live-Coding; Human Computer Interaction;

ACM Classification Keywords

User Interfaces; Human factors; Sound and Music Computing

INTRODUCTION

Live-coding [7] allows musical performers to generate, transform, and arrange musical material however they desire. This offers an incredible amount of freedom to performers. However, the lack of an existing musical structure can be daunting for novices. If novices cannot find useful software libraries as a starting point, they are forced to build up all of their organizational tools for performing and composing from scratch.

Furthermore, "pure" live coding, (i.e. live coding without the use of any interface than the code itself) has some performance limitations as compared with hardware controllers. Live-coding lacks the same immediacy of expression as is possible with a physical instrument or controller - when performing, improvising a melody on a guitar is much faster than typing one out into a text editor. Also, code is not always the most efficient interface for inputting information into a system. For example, it is typically much faster to play and record a melody on a digital keyboard than to type it in some ASCII representation.

Hybrid live-coding systems (i.e. live-coding systems that incorporate other interfaces) can help address some of these problems. For example, a physical bank of sliders could bring more spontaneity to audio performance by letting a performer play freely with audio-filter values, or, a step sequencer UI could speed up the process of annotating complicated arpeggio patterns. However, though the music interface literature documents instances of hybrid live coding environments [2, 9], these works provide no information to help performs design interfaces in these environments. In general, there is a dearth of research into interface-agnostic best practices for designing custom interfaces in hybrid live-coding environment.

To address this gap, we present the notion of calcification, a simple framework for designing both live coding libraries and hybrid live-coding systems. This paper will also present some libraries built with this framework in mind and will discuss how this framework can help integrate various types of physical interfaces into a hybrid live-coding system. We hope to use this work as a starting point for deeper research into hybrid live-coding environments.

CONCEPTUAL BACKGROUND

The design framework presented here incorporates two existing concepts in interaction design literature: liveness and malleability. Liveness refers to how quickly a system responds to user input. Steven Tanimoto investigated the concept with respect to programming environments [8]. Christopher Nash further developed this concept to analyze musical notation systems [6]. In particular, Nash characterized different levels of liveness and provided examples of such systems in the Table 1, reproduced from [6].

Table 1. Levels of liveness in programming and music.

level	informative	significant	responsive	live	description	in programming	in music
1	●				**"ancillary"** in which a visual representation is used as an aid to design, providing a basic graphical representation that is continuously visible, but not executable.	flow chart, UML diagram	composer shorthand, arrange window
2	●	●			**"executable"** in which the system uses a visual representation as an executable specification; offering input to a compiler, but not continuously interpreted.	code editor, script, compiler	score, data list, piano roll, CSound, OpenMusic
3	●	●	●		**"edit-triggered"** in which edits to the representation instantly trigger feedback, allowing users to make rapid actions and (after system response) a chance to correct mistakes.	auto complete, syntax highlights, edit & continue	soundtracking, live coding,
4	●	●	●	●	**"stream-driven"** in which the program is continually active, and where edits directly affect program execution, providing high visibility of the effect of actions.	macro recording	sequencer/DAW recording, live performance, mixing

A user interface is said to be "malleable" when it can be modified to better fit a user's particular context of use [1]. For example, the iPhone home screen exhibits malleability by letting users rearrange application icons into different positions and folders, allowing faster access to more frequently used applications. Dyck, et al in particular, highlights the importance of malleability when designing efficiently usable interfaces for real-time tasks:

> "Game interfaces are plastic [malleable]; they are designed to be changed [e.g. changing what button is used to shoot in a shooting game]. Gamers have learned that different interface configurations can greatly affect performance in different game situations, and that no single configuration can be appropriate for all tasks. This is equally true of complex conventional applications like Word or Photoshop; the difference is that gamers see the extra effort required for a suboptimal interface configuration as the difference between victory and defeat, or life and death." [1]

Interface-efficiency can thus be the difference between success and failure—and this notion also applies to musical improvisation, where starting a loop a split second late can sound out-of-sync and disruptive. Therefore, it is important that malleable music interfaces can be reconfigured into an optimally live form.

CALCIFICATION

It is with liveness and malleability in mind that we introduce the novel concept of calcification: the act of "freezing" a malleable interface in a particular, static configuration to use it in a more "live" way. The process of applying calcification to a system involves 3 main questions:

1. What parts of this system are malleable if we don't consider time constraints?
2. What well-defined instance or behavior of this system do I want to access under time constraints, and (if relevant), what information can be used to parameterize this instance or behavior?
3. How can I create an efficient way to store and access the different instances or behaviors?

"Calcification" produces a "calcified component" - which we define to be an interface whose mapping or representation is fixed during performance.

A straightforward example of calcification in a live coding environment is the creation of a library of functions for generating a random melody. There are many different algorithms one could use to create a random melody (the "well defined behavior" from question two), and the library API serves as a fixed interface for storing and accessing those algorithms. The arguments to the function calls are the parameters for the behavior. The API of function calls, unlike the code used to define the functions, is efficient enough to use in performance. Even in dynamic environments where the code defining the API can be modified at performance time, the API serves as a conceptual differentiator between "surface" and "deep" changes to the performance environment. Proper calcification is important because it takes functionality that would otherwise be level-2 live and wraps it in an interface that is level-3 live, thus making it usable during performance. Tidal, a live-coding library in Haskell, calcifies code via a domain specific language (DSL) - a particularly powerful approach discussed in the section "CASE STUDY: TIDAL".

Another powerful strategy for creating malleable, yet very "live" interfaces is to integrate various types of non-code interfaces, such as digital keyboards, DJ hardware, or even traditional GUIs, into a live coding system. The idea of calcification can help guide the design of APIs that can integrate with external interfaces. The section "HYBRID LIVE CODING AND CALCIFICATION" describes the advantages of hybrid live-coding environments and discusses the challenges of calcifying them.

CASE STUDY: TIDAL

Alex McLean's Tidal project [5] presents an interesting example of the calcification of code. It provides a domain specific language (DSL) for defining rhythmic patterns and a library of Haskell functions for manipulating them. The Tidal DSL is simply a Haskell string that is parsed into an internal data structure representing a rhythm. For example, the single line program:

```
d1 $ sound "bd sn sn"
```

would play a loop comprised of a bass drum hit on every first beat and snare hits on every 2nd and 3rd beat. `sound` is the Haskell function that parses the string `"bd sn sn"` into the internal rhythm representation, and d1 is the function that "plays" the rhythm. The Tidal DSL can be analyzed through the three questions of the calcification process.The "full system" in general is the Haskell language, and given no time constraints, virtually all parts of the system are malleable and any structure is eventually expressible. From this total system, the behaviors to be calcified are the expression of simple rhythms and their combination into poly-rhythms. In creating an interface for expressing these behaviors, the Tidal syntax is a much more efficient textual representation of rhythmic patterns than any representation possible using Haskell syntax.

Tidal also presents a library of functions for transforming rhythms. For example, in the single line program:

```
d1 $ rev sound "bd sn sn"
```

The function `rev` reverses the pattern object created by sound `"bd sn sn"`. The pattern transformation API, which is implemented as a set of Haskell functions, presents the more malleable part of the interface. Functions can be combined in various ways and new functions can be written on the fly in a bundled Emacs interpreter.

Though Tidal's initial setup is configured to run as a drum-machine, its core library is intentionally designed to let users send real-time OSC messages to any application, making Tidal a generic DSL for rhythmic patterns.

Tidal utilizes two particular design strategies that can be used in other contexts. Firstly, its creation of a DSL provides a powerful method for defining complex objects, and is much more efficient than a conventional API. Secondly, its design decouples the core structure (in this case, the rhythmic pattern generator) from instances of its use (i.e a drum machine). Though Tidal comes pre-configured to sequence drums, it can easily be used to sequence melodies, lighting, and more.

NEW LIBRARIES

The following libraries attempt to follow Tidal's example and create generic, structure-defining DSLs which are embedded within an environment that lets users "plug in" the specific behavior around those structures. In particular, we aimed to create Python libraries that could assist in the "high level" organization of a musical performance.

Trees

We wanted to build an efficient interface for working with tree structures, and we used the calcification framework to help build a DSL for the task.

Design Process

The specific use-case for the tool would be to store variations of melodies - a node would contain a particular melody, and its children would contain algorithmically generated variations on that melody. However, we also wanted the interface to be more generally useful for working with trees. We knew we wanted our environment to be the Python programming language, and thus any types of operations on trees were representable, given enough time.

We determined that the core behaviors we wanted to calcify were 1) traversing through the tree one node at a time (i.e moving from a node to its parent, a child, or a sibling) and, 2) generating a child node from a parent node by creating a variation on the parent node's melody. We realized that, if we viewed these two operations (traversing node to node and creating a new child) as functions, the only information they would require were 1) what the current position in the tree is, and 2) a function that creates a variation on a melody given a source melody. We also decided that, for reasons related to the initial musical use-case, we would want a tree where the children of a node are stored as a list (where order matters) rather than as a set.

To create an efficient representation of these behaviors, we decided to create a DSL where each of 5 functions (move to parent, move to a child, move "forwards" in the sibling list to a sibling, move "backwards" in the sibling list to a sibling, create a new child) would represented by symbols. We implemented a simple Node object in Python such that calling `node.execute("symbols from tree DSL")` would execute, in order, the functions specified by the symbols in the DSL string.

Implementation

A tree is instantiated with an initial melody object rootMel and some function `transFunc` that takes a melody as an argument and returns a variation of it.

```
t = TreeBuilder(rootMel, transFunc)
```

`t` is now a tree object with a single node, and that node's value is the object `rootMel`. `t.currentNode` is the pointer to the current position in the tree, and `t.root` is a pointer to the root.

The language itself has has 5 core operators:

`\/!` : creating a child for the current node. Each node has a list of children, and this places the new node at the end of the list. The value for that child is the output of `transFunc(t.currentNode.val)`

`^` : moves the the current node pointer to the parent of the current node (does nothing if the current node is the root)

`\/` : moves the current node the first of its children (does nothing if there are no children)

< : moves the current node to its previous sibling (rolls over to the end of the list if the current node is the first sibling)

> : moves the current node to its next sibling (rolls over to the start of the list if the current node is the last sibling)

Executing the line `t.executue("\/! ^ \/! ^ \/! < \/! ^ \/!")` would create a tree with the structure shown in figure 1:

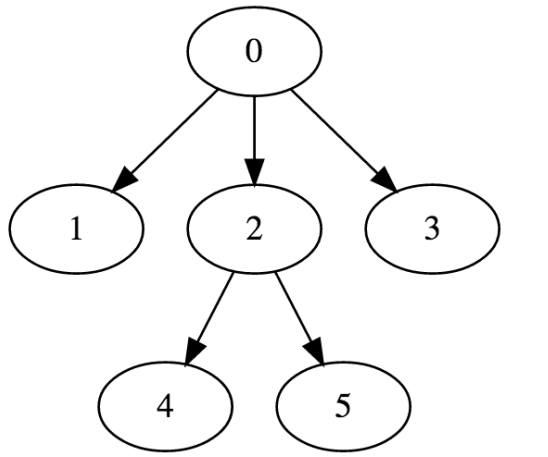

Figure 1. The tree structure produced by t.executue("\/! ^ \/! ^ \/! < \/! ^ \/!")

Further operators include :

>! : creating a new sibling after the current node. The value for that newly created node is the output of `transFunc(t.currentNode.val)`

<! : creating a new sibling before the current node. The value for that newly created node is the output of `transFunc(t.currentNode.val)`

\/:n : moves down to the nth child (wraps around if n > num children)

>:n and **<:n** : moves forwards/backwards n steps in the sibling list (wraps around if end/start of list is reached)

(...)*n : repeats the sequence of symbols in the parenthesis n times
e.g. `\/! (>! >! \/!)*2` becomes
`\/! >! >! \/! >! >! \/!`

The code in figure 2 will produce the tree in figure 3 (this example is intended to show the structure of the tree and order of the nodes created. It does not show the intended usage of node-values and transformation functions).

```
class Counter:
    def __init__(self):
        self.count = 0
    def __call__(self, arg):
        self.count += 1
        return self.count

inc = Counter()
tree = TreeBuilder(0, inc)
tree.execute("\/! \/! >! >! < < >!")
```

Figure 2

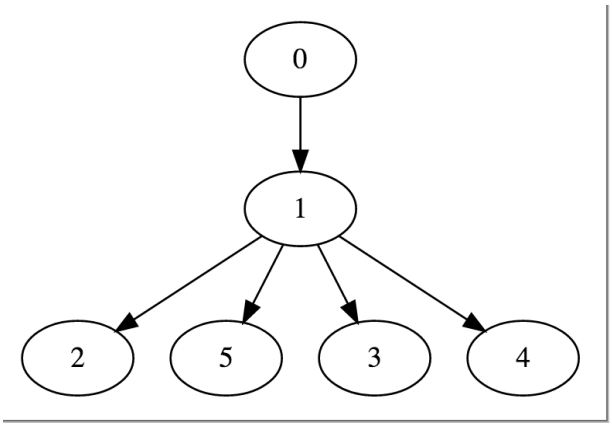

Figure 3

Extensibility
Though the author's intended use case for this library is to track variations of melodies, the tree can track the variation of any type of object, provided that users supply an appropriate function to the constructor for creating variations of said object type.

Users need not use the variational aspect of the tree at all. Calling the constructor with no arguments allows the user to build and traverse a tree with empty values. Also, a node's `val` property is publicly accessible, allowing users to store arbitrary values at any node in the tree.

State-transition networks
A second problem we wanted to tackle was creating an efficient interface for describing event-based state-transitions. A similar process of calcification led to the following DSL. We determined that it would be most efficient to model the state-transition network as a directed graph, where states correspond to nodes and events correspond to named edges. The language uses syntax similar to DOT[1] for describing event transitions. The line:

```
s1 --(e1)--> s2
```

Would indicate that an event of type `e1` would trigger a transition from state `s1` to `s2` (given that the system is in state `s1`). Compound statements such as:

[1] http://www.graphviz.org/doc/info/lang.html

```
s1 --(e1)--> s2 --(e2)--> s3
```

would be equivalent to the two separate statements:

```
s1 --(e1)--> s2
s2 --(e2)--> s3
```

The network setup in figure 4 is visualized by the transition graph in figure 5, with "event" names in parenthesis.

```
graphString = """
f1 --(s1)--> f2 --(pad1)--> f4

f2 --(seq1)--> f3

f4 --(r)--> f1
f4 --(r)--> f2
"""

stateToFunctionMap = {}

stateToFunctionMap["f1"] = func1
stateToFunctionMap["f2"] = func2
stateToFunctionMap["f3"] = func3
stateToFunctionMap["f4"] = func4

tg = TransitionGraph(stateToFunctionMap, graphString)
```

Figure 4

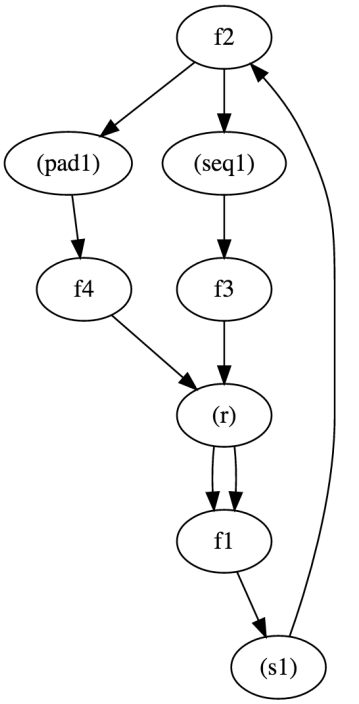

Figure 5

An "event" can be triggered by sending an OSC message to its address, which is "/" plus its string.

Each state corresponds to a function, which is called when the state is entered. When constructing a transition network, the state-function mapping can be passed into the constructor as a Python dictionary (as shown by `funcMap` in the code). Because Python supports a "callable" interface for objects, users can even pass in classes so that the "function" that executes for a state has "stateful behavior" (i.e. has memory).

If no default state is set, the start state is the "from" node of the first edge described. The network supports logic for being in multiple states at once, and the "state-transition" logic upon the occurrence of an event is analogous to that of an NFA (Nondeterministic Finite Automaton). Users can also explicitly set what state(s) they want the network to be in via a class variable.

HYBRID LIVE CODING AND CALCIFICATION

Integrating instrument-like interfaces (ILIs) into a live-coding system allows for both level-4 liveness and powerful malleability. We define "instrument-like interfaces" to be any interfaces that are physically actuated in real-time (e.g., analog instruments, digital keyboards, faders and sliders, etc.). A simple yet powerful way to integrate ILIs and live-coding is to record the actions taken on an ILI and make the time-series of events available as data for manipulation via live-coding. For example, a performer could play a melody on a piano and record it into the live-coding system, and then algorithmically manipulate the melody and replay it using some predefined functions.

Live coding is especially powerful when combined with digital ILIs, which are ILIs that send MIDI, OSC, or other such digital messages. The performer can define arbitrary handler functions for input events (MIDI, OSC, etc) from the ILI during the performance itself and execute arbitrary blocks of code with the push of a button. If the goal for calcification in pure live-coding environments is to create a level-3 live interfaces for a behavior, the goal for calcification in hybrid live-coding environments with ILIs is to create a level-4 live interfaces for a behavior.

This new goal presents new challenges. When trying to calcify behaviors and structures in pure live-coding, the parameters for those behaviors and structures are also expressed in code. For example, when trying to create an API, the arguments to those functions can be anything expressible by code. When trying to calcify behaviors and structures for level-4 liveness hybrid live coding systems, the behaviors to be calcified are parametrized by both level-3 live inputs (code, selections from a graphical menu, etc.) and level-4 live inputs (events from an ILI). However, the level-4 live inputs (for example, MIDI messages) will often be much less expressive than code, and there are a limited amount of physical inputs (sliders, keys, etc) on an ILI, thus limiting the number of input parameters to work with. The challenge in trying to calcify for level-4 liveness is to create an interface that is as expressive as possible with respect to its level-4 live inputs while being efficient about the number of physical inputs used.

For example, in an environment consisting of MIDI keyboard and a live-coding environment that allows live-coding handlers for MIDI events, we want to calcify the given behavior: given a chord progression that is playing in the background, "autocorrect" the notes played on a

keyboard such that a user will never play a note that is out of key with respect to the chords in the progression. In this behavior, the level-4 live input is the note being played, and the level-3 live input is the set of last 3 chords played in the progression. Since there are potentially several different strategies for determining the correct note, we could say that the function defining a particular strategy another level-3 live input, to be determined by code in the definition of the MIDI event handler. The user could even decide that they want to be able to "hot swap" the autocorrect strategy with the press of a button, thus making the choice of autocorrect function a level-4 live parameter. However, the cost is that there is now 1 less button available to map.

FUTURE WORK

Effective calcification of a behavior is highly dependent on the affordances [3] of the interface(s) used in a hybrid live-coding system (an object's affordances are the ways in which it can be interacted with). Going forward, we wish to analyze the affordances of various interfaces in order to find common design patterns that can arise in the building of live performance systems. In the long run, we wish to develop a vocabulary for describing music performance interfaces that is analogous to the Cognitive Dimensions of Notations (CDN) [4]. The CDN is a set of terms, each of which describes a relatively independent property, or dimension, of notation systems. The definition of a set of dimensions makes it easier to clearly describe the advantages and disadvantages of a design decision - a particular decision can be said to strengthen some dimensions and weaken others. The definition of a "Dimensions of Performance Systems" would be an invaluable tool in making the calcification process more rigorous - calcification could be formulated as a way to move from one point in the dimension space to another.

REFERENCES

1. Dyck, Jeff, David Pinelle, Barry Brown, and Carl Gutwin. "Learning from Games: HCI Design Innovations in Entertainment Software." In Graphics Interface, pp. 237-246. 2003.
2. Essl, Georg. UrMus-an environment for mobile instrument design and performance. Ann Arbor, MI: Michigan Publishing, University of Michigan Library, 2010.
3. Gibson, James J. The ecological approach to visual perception: classic edition. Psychology Press, 2014.
4. Green, Thomas R. G., and Marian Petre. "Usability analysis of visual programming environments: a 'cognitive dimensions' framework." Journal of Visual Languages & Computing 7, no. 2 (1996): 131-174.
5. McLean, Alex, and Geraint Wiggins. "Tidal–pattern language for the live coding of music." Proceedings of the 7th sound and music computing conference. 2010.Nash, Chris, and Alan Blackwell. "Liveness and Flow in Notation Use." InNIME. 2012.
6. Collins, Nick, Alex McLean, Julian Rohrhuber, and Adrian Ward. "Live coding in laptop performance." Organised sound 8, no. 03 (2003): 321-330.
7. Tanimoto, S. VIVA: A Visual Language for Image Processing. In Journal of Visual Languages and Computing. Academic Press. pp. 127-139, 1990.
8. Wang, Ge, et al. "Yeah, ChucK it!⇒ dynamic, controllable interface mapping."Proceedings of the 2005 conference on New interfaces for musical expression. National University of Singapore, 2005.