Estuary: Browser-based Collaborative Projectional Live Coding of Musical Patterns

David Ogborn
McMaster University
ogbornd@mcmaster.ca

Jamie Beverley
McMaster University
beverljw@mcmaster.ca

Luis Navarro del Angel
McMaster University
navarrol@mcmaster.ca

Eldad Tsabary
Concordia University
eldad.tsabary@concordia.ca

Alex McLean
Deutsches Museum
alex@.slab.org

ABSTRACT

This paper describes the initial design and development of Estuary, a browser-based collaborative projectional editing environment built on top of the popular TidalCycles language for the live coding of musical pattern. Key features of Estuary include a strict form of structure editing (making syntactical errors impossible), a click-only border-free approach to interface design, explicit notations to modulate the liveness of different parts of the code, and a server-based network collaboration system that can be used for many simultaneous collaborative live coding performances, as well as to present different views of the same live coding activity. Estuary has been developed using Reflex-DOM, a Haskell-based framework for web development whose strictness promises robustness and security advantages.

1 Goals

1.1 Projectional Editing and Live Coding

The interface of the text editor is focused on alphabetic characters rather than on larger tokens that directly represent structures specific to a certain way of notating algorithms. Programmers using text editors, including live coding artists using text-based programming languages as their media, then have to devote cognitive resources to dealing with various kinds of mistakes and delays, including but not limited to accidentally typing the wrong character, not recalling the name of an entity, attempting to use a construct from one language in another, and misrecognition of "where one is" in the hierarchy of structures, among many other possible challenges.

Structure editing (Khwaja and Urban 1993; Miller et al. 1994) is one response to this problem. In its strictest form, structure editing only allows a programmer to select valid and contextually meaningful tokens for insertion into a program. Instead of inserting characters into a list of characters (a text document), the programmer inserts structures into other structures. Projectional editing is a related but orthogonal response in which programmers are assisted in their efforts by the presence of interfaces providing multiple perspectives or "projections" of the structure upon which they are working (Voelter et al. 2014; Voelter et al. 2014; Walkingshaw and Ostermann 2014). Industrial programming over the past few decades has adopted structure and projectional editing features, typically in "relaxed" but widespread forms. Contemporary integrated development environments (IDEs) for mainstream languages like Objective C, C# and Java have sophisticated interfaces that parse programmers' text code in order to show errors, suggest corrections, and allow the computational entities within a project to be navigated in different ways (for example, through class hierarchies). At the same time, programming in these languages is still very much a matter of one-character-at-a-time typing operations.

Within the culture of live coding, the SuperCollider IDE invites some comparison to these industrial examples of structure and projectional editing, with some affordances that provide assistance (such as pop up tips about the naming and order of arguments) and introspection of running state (including metres showing audio input and output levels, reports on currently existing synthesis nodes, etc). Such liminal examples notwithstanding, live coding as of 2017 seems to most often involve typing text characters, without very much assistance, if any, from the software environment. The TOPLAP draft manifesto's decidedly ambiguous comment that "Live coding may be accompanied by an impressive display of manual dexterity and the glorification of the typing interface" comes readily to mind (Ward et al. 2004).

And yet, projectional editing has a special connection to live coding in so much as a certain prototypical live coding performance situation already involves multiple projections of the code: an audience sees a text notation of an algorithm or

data structure, and at the same time they hear or see the audio or video result of executing or evaluating that text notation. The presence of this hierarchically layered perception in the prototypical live coding event suggests, moreover, that the potential impact of projectional editing approaches to live coding is not limited to reducing cognitive load. New, additional and multiple projections around an algorithm may also introduce distinct aesthetic qualities, or communicate with specific audiences.

Indeed, live coding performance settings offer abundant opportunities to present multiple, additional projections. In the decentred environment of an algorave, for example, it is not uncommon to have multiple video projectors pointed at different surfaces. Collaborative settings such as laptop orchestras bring together numerous computers and displays, which need not be conceived only as identical interfaces for each participant. Internet performances, including but not limited to performances streamed as video over the Internet, can scale to involve hundreds or thousands of display interfaces as more and more audience members arrive. Audience members "at" such performances can not help but exercise some limited individualization of the way they view the event — for example, resizing browser windows to their satisfaction — and when streamed footage combines camera capture of a performer's body with their text interface this binary is necessarily navigated by each audience member's attention. One can also envision systems in which members of an Internet audience have a stronger agency in navigating a live coding performance through multiple interfaces. In short, for live coding, projectional editing can be not only a matter of poiesis (ie. supporting the production of the live coding event) but also aesthesis (ie. modifying when and how it is received, actualized, communicated, translated).

The openness of projectional editing — one can always imagine one more, different interface — may also present pedagogical opportunities that, like the possibility of presenting distinct aesthetic qualities, are not restricted to simply "making programming easier for beginners". An additional interface may be created in order to triangulate between diverse curricular concerns. A teacher in an elementary school may conceive of an interface variant that is tailored to a specific genre of music, as well as to general vocabulary and thematic concerns that are highly specific to the situation of their class (perhaps exploring concepts from social studies, science, literature, mathematics, etc). Live coding, in such a situation, becomes not only about making music/art nor only about making the computer do something, but rather (and also) about carrying through broader pedagogical and cultural intentions.

1.2 The Web Browser as Platform

To explore the above-mentioned general potentials of projectional editing for live coding, we created Estuary, a browser-based collaborative live coding system. The decision to make Estuary run in the browser had multiple motivations. Above all, it was felt that the pedagogical potential of projectional editing would be maximally unfolded when it was possible to use the system on a "zero-installation" basis, as installing new software may be difficult or even impossible in many school and workshop environments. The authors' recent experience in the Cybernetic Orchestra, the nearly eight years old continuously running laptop orchestra at McMaster University, reinforced this motivation: the practice of the orchestra in recent years has been to start each new semester, when there are typically new members in need of orientation, by live coding entirely through the web-based extramuros interface, allowing people to start to have successful experiences with live coding "in the first five minutes" and deferring the potential unpleasantness of installing software with complex dependencies to a later and more confident moment in the semester.

An additional set of advantages of targeting web browsers as the environment for a new live coding system pertains to security concerns. On the one hand, web browsers already enforce strict security policies on behalf of users, and so targeting web browsers gives us an extra measure of confidence that our system won't be used to do evil things to our or other people's computers and personal information. On the other hand, standard HTTP and WebSockets communication between browsers and servers may in some circumstances be given privileged access by home and institutional security systems (firewalls, routers, etc) and thus networked collaboration may be able to take place without special negotiation of security protocols, even in relatively security-conscious environments. In short, browser environments supply a working default that balances keeping things out and letting them in.

Given the advantages above, it is no surprise that the production of new live coding environments specifically targeting the browser is a vibrant sub-field of live coding research. The majority of these environments have targeted text editing as a way of specifying and interacting with programs. Some of these browser-based projects foreground and leverage JavaScript as the "first language" of the browser, including Gibber (Charles Roberts and Kuchera-Morin 2012), Alive (Wakefield et al. 2014), and EarSketch (Mahadevan et al., n.d.). Other browser-based live coding projects instead propose new domain specific languages, leaving JavaScript as an underlying language of implementation, Examples of this second approach including Lich.js (McKinney 2014) and LiveCodeLab (Davide Della Casa and John 2014). One recent exception to the text editing orientation of many browser-based live coding projects is the block-based structure editing approach of Snap Music (Fradkin 2016).

1.3 TidalCycles

After this focus on zero-installation execution in the web browser, Estuary is distinguished by a second major feature: it attempts to stay very close to the idioms of, and expectations drawn from, the TidalCycles language for musical pattern (A. McLean 2014; A. McLean and Wiggins 2010). In other words, instead of proposing a new independent language we have adapted an existing language around the idea of collaborative, projectional editing in the browser. The decision to base the new project on the TidalCycles language had several motivations.

To begin with, the domain of the TidalCycles language is relatively limited (notwithstanding the impressive diversity of music produced by its practitioners), and basically consists of cyclical patterns of sample/note events. TidalCycles core notations are not used, for example, to design graphs of unit generators, nor to map gestural controllers to musical outputs (of course nothing stands in the way of ambitious TidalCyclers who do such things alongside or in connection with their TidalCycles patterns). For a time-limited research project focused on "new ways of editing" an inherited and constrained domain represented a decided advantage: already knowing, at some level, what "result" the project is to produce allowed us to jump into imagining and implementing notations for that domain.

TidalCycles features a highly economical mini-language for notating sequences, which consists, prototypically, of distributing identifiers for parameters spatially, with that spatial distribution denoting an equal division of spans of time. This key notational concept in TidalCycles has no essential relationship to the interface of the text editor, even if its notational economy can be seen as closely adapted to the expense of typing operations for the individual live coder using a text interface. It is easy to jump from distributing text tokens along a linear space (as in TidalCycles) to distributing "ideographic" tokens along a linear space. To articulate the same question somewhat differently: There is a homegeneity in TidalCycle's core sequence notation that, like the constrained domain, makes a uniquely tractable target for the development of a system based on structure editing. Modeling this core sequence notation has been a primary focus of Estuary's structure editor, although we also include other elements of TidalCycle such as functions which combine sequences and transform patterns.

Finally, by basing the "new" system on TidalCycles we hope to provide lines of flight into and out of Estuary. Someone may have introductory live coding experiences with Estuary and then be able to take what they have learned into the differently constrained environment of a contemporary TidalCycles installation. Conversely, in putting the new system out there to stimulate discussion and research, we would be able to count on insights from a certain kind of expert sub-audience who might look at our projectional and collaborative editing ideas through the lens of their experience with TidalCycles. Inevitably, the notations and behaviours of Estuary depart from those of TidalCycles but in not trying to be different from an existing language where we don't need to be different there is the possibility for a kind of enhanced mobility between (live coding) linguistic communities.

Perhaps this is the place to note that this project occupies a liminal position with respect to what one might call the "theatre of code". There is a certain fetishization of alphabetic editing at work in the live coding community. The structure editing aspects of Estuary depart from this typing interface (it is possible to do "everything" by pointing and clicking). At the same time, the conception of this structure editing interface has been relatively faithful to the structure and alphabetic appearance of TidalCycles. We consider salutary any and all demonstrations that code and computation need not require the specific alphabetic script of a specific, historically delimited culture. At the same time, we have started where we are, with the specific alphabetic script of a specific, historically delimited culture…

2 Design Choices

2.1 Click-Only Structure Editing

Some common metaphors are to be found in well known structure editing projects. Closest perhaps to the artistic and pedagogical intentions of this project is the example provided by the Scratch environment, designed to provide foundational programming experiences in a creative context, for children across a wide range of ages, with documented pedagogical success (Maloney et al. 2004; Malan and Leitner 2007; Rizvi et al. 2011; Franklin et al. 2013; Y. B. Kafai and Burke 2013; Giannakos, Jaccheri, and Proto 2013). Scratch, the aforementioned Snap Music, and David Griffiths' SchemeBricks (A. McLean et al. 2010) all employ a metaphor of blocks with slots in them to contain other blocks. Although the design of Estuary would soon move away from the building block metaphor, block-based systems influenced our initial conception of what we were trying to accomplish, as well our first experiments with Reflex, the Haskell-based framework in which Estuary was ultimately implemented (see Implementation, below).

At a relatively early phase of Estuary's development we had the opportunity to present the software-in-progress at an international conference, with the specific proviso that it would be used to allow a large group of conference delegates to live code on their mobile phones, tablets, etc. While drag and drop is certainly possible on such devices, it can be quite awkward, especially on smaller devices. For that reason, we produced a provisional interface in which every programming operation (adding an element to a structure, removing an element, etc) could be completed simply by tapping (i.e. clicking)

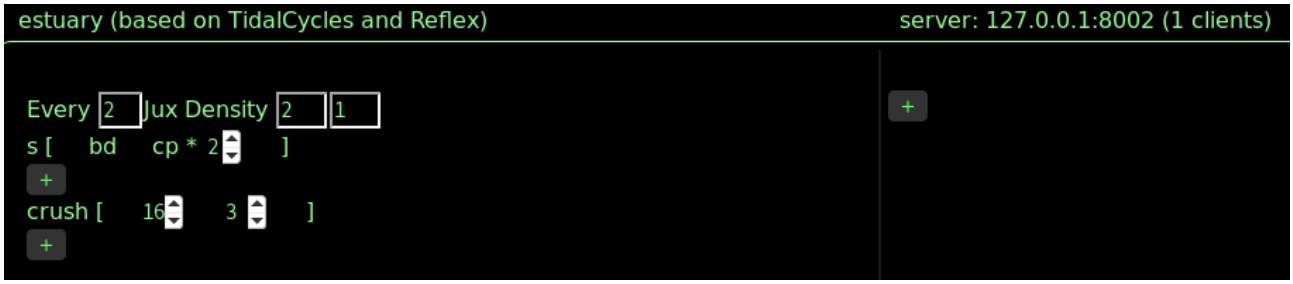

Figure 1: An Estuary structure editor. The equivalent TidalCycles code would be: every 2 $ jux (density) $ s “bd cp*2” # crush “16 3”

in a definite location within what is displayed. The chance constraint of this conference presentation became a defining feature of Estuary’s approach to structure editing: by expecting that all or most user interactions would take the shape of taps/clicks we would be able to create a structure editor that would work in a similar way across a very broad range of devices, essentially taking further advantage of the relative universality of the web browser platform.

2.2 Blank Space as Interface

A second key design question concerned how to visually articulate the structure being edited. Following the model provided by the block-based systems we at first explored approaches in which the elements within a musical pattern appeared as bordered and/or differently coloured blocks within other blocks. This introduced an element of visual clutter, especially as the detail of the structure increases. It should be noted that a variable level of detail is characteristic of programming in general, and is especially characteristic of live coding, wherein a common trajectory is to start at a “blank screen” (very low detail), then to become a simple structure, then to become a more complex structure, and finally perhaps to be dismantled again in the direction of simpler structures. We began to experiment with “border free” approaches in which, as much as possible, sharp edges were to be avoided. This removal of sharp edges had the additional consequence that our structure editing interface came to more closely resemble TidalCycles code as entered in a text editor.

From these two constraints — click-only interactions and “border free” — we arrived at the final distinctive feature of Estuary’s core structure editor: copious popup menus. Tapping either on blank space or on a “text” element in the structure editor brings up a context-sensitive menu. In this way, all of the blank space of the structure editor becomes latent with possibility (just as in contemporary operating system windowing environments typically every space on the screen has some context sensitive actions connected to it).

This reduces “standing clutter” in the interface, but comes with the additional consequence that the functionality connected to a given region of blank space is (until activated) somewhat opaque (compared, for example, to a block-based system where possible blocks may be arranged and colour-coded on the edge of the work area). However, we tend to consider this situation theatrically and pedagogically advantageous: prior to tapping there is a mystery about what is possible, and after tapping there is an explicit transparency about what is possible. Moreover, the risk of any given tapping operation is low, as the options that pop-up need to be tapped again in order to be selected, and a pop-up menu can easily be closed. Fear of “clicking in the wrong place” is an important issue for any pedagogically-minded project to consider, and it is to be hoped that an interface which encourages exploratory clicking in a low-risk way may encourage people to explore other software in a similar way.

2.3 Explicit Notations of Liveness

When programming with a text editor, it is most common for the evaluation/execution of code to be triggered by a special separate operation (for example, pressing Cmd- or Ctrl-Enter to evaluate a small region of code in the SuperCollider IDE). The nature of programming by typing alphabetic notations strongly encourages such “modify and then evaluate” workflows as a default, since any text being created or modified will likely occupy a number of invalid or undesirable states on its way to occupying a valid or desired state. From one standpoint, this can be seen as an encumbrance: the artist programmer must expend additional time and energy on the operation of signalling that the code should be evaluated/executed. From another standpoint, however, the non-immediacy of evaluation/execution is an advantage: artist programmers frequently operate in time-sensitive domains like musical improvisation where being able to wait for the right moment to trigger an operation can be extremely important.

By contrast, in graphical dataflow programming environments, such as Max or PD, the expectation tends to be that changes or additions to a programming structure will take effect immediately, for better and for worse. One text-oriented language

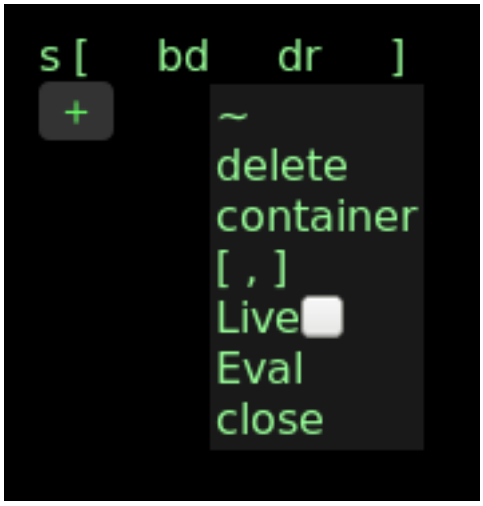

Figure 2: A popup menu showing Estuary's liveness controls. This sequence of a bass drum and drum sound is at L3 ("not live"). Clicking the Live box would make it L4. Clicking Eval would make its present visible state the executing state.

that adopts a similar "aggressive" approach to execution is LiveCodeLab wherein syntactically correct expressions are evaluated immediately (and repeatedly). Since syntactically correct expressions in LiveCodeLang (the domain specific language employed by LiveCodeLab) may nonetheless still lead to runtime errors, LiveCodeLab adopts an additional liveness-related measure: expressions that survive a certain amount of time without causing a runtime error are marked as stable, remembered, and fallen back to when replaced by later expressions that do cause a runtime error (Davide Della Casa and John 2014).

Given Estuary's strong emphasis on structure editing, liveness (the way in which changes to a programming notation take effect in time) becomes an unavoidable design choice, and the specific context of improvised musical performance suggests that different forms of liveness are applicable to potentially directly adjacent situations. Estuary responds to this by introducing explicit, editable notations for the liveness of different parts of a structure. In Estuary, any element that represents a subsequence (in other words, any element that can be a container for multiple other elements) has one of two possible liveness levels: at one level, changes made do not "take effect" until a later "evaluate" operation; at the other level, changes "take effect" immediately. Both in order to echo Tanimoto's taxonomy of levels of liveness (S. L. Tanimoto 1990; S. L. Tanimoto 2013) and in order to take advantage of the highly abbreviated names suggested by such taxonomies, Estuary denotes these two levels of liveness as L3 and L4 respectively.

"Take effect" is in quotation marks in the preceding paragraph because notations in Estuary may be embedded within other notations that have different liveness characteristics. If, for example, a part of a structure marked as L4 is contained within a higher level structure marked as L3, changes to that part of the structure will only propagate to the top, where they are allowed to take effect on the sounding result, when the containing structure is evaluated. At the same time, both the intended changes and the different liveness parameters of different parts of the structure are visible to all (although in the current design the liveness parameters are only directly visible when they are being changed). Changing a structure at L3 liveness without evaluating it has no immediate effect on the sounding result but it does have an effect on the visible structure experienced by the artist programmer, their collaborators and any audience members.

The development of new concepts and notations for the liveness of different parts of a programming structure could be an area of significant creativity in the development of live coding environments, more generally, as well as an area in which artistically-oriented experiments with unfamiliar models of liveness can inform and be informed by more industrially oriented research. The vision of tactically and strategically predictive programming environments (S. L. Tanimoto 2013; Church et al. 2016) makes us think of the future potential for the liveness of a notation to be modulated algorithmically, whether based on an inference about a programmer's intentions or as an independent aesthetic value operating alongside or even against those intentions. While strong structure editing forces a design choice about liveness (since the code is always in a valid "evaluable" state), text-based environments too could explore further and different models of liveness. It is easy to imagine a special notation operating in the SuperCollider IDE, for example, that would support L4-style continuous evaluation.

2.4 Panoptical, projectional collaboration

Live coding is deeply entangled with collaboration: sharing not only the result of a process but the specification of that process (for example, when code is projected for the benefit of an audience) is a collaborative maneuver. Or to put it

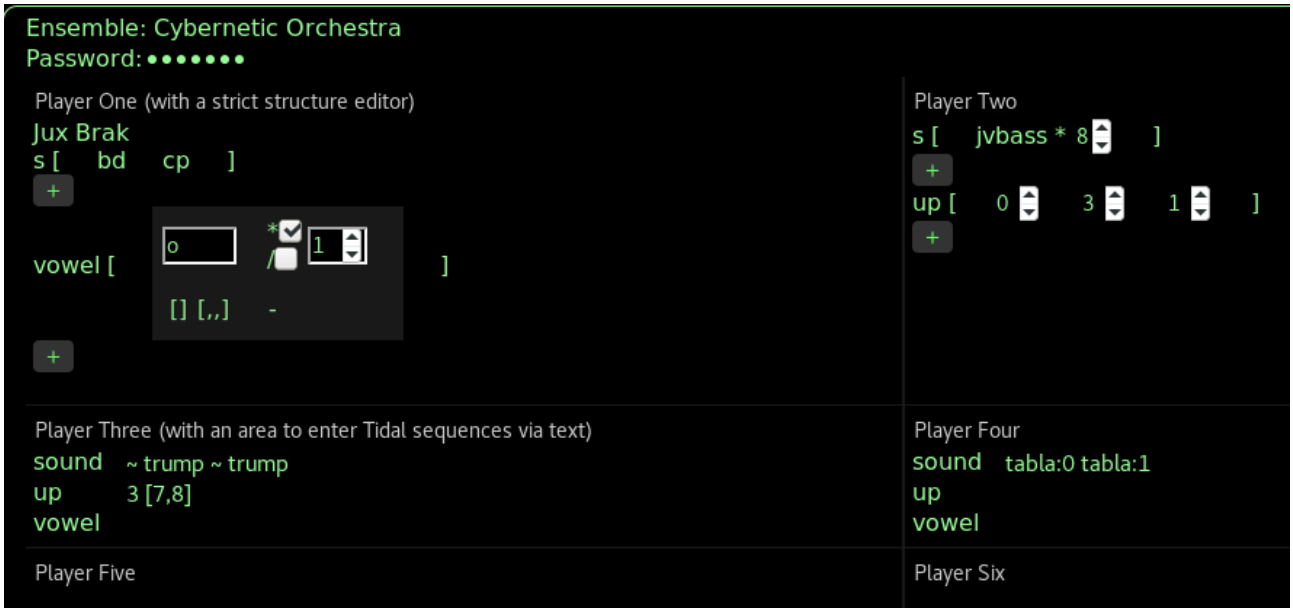

Figure 3: A hypothetical collaborative editing situation in Estuary. Any player can edit on any of the different panels, if they have entered the correct password for that ensemble. However, each panel is labeled and the usual performance practice is for people to use an individual panel.

differently: a collaborator is someone with whom we share code. Little surprise then that there are so many explicit forms of collaboration to be found in and around live coding activity, including small and large ensembles of co-performers, practices like roulette in which live coding performers take turns modifying a given base of code, and software systems developed to facilitate collaboration. Estuary was conceived from the beginning as something that would support networked collaboration in shared interfaces. In its current form, an Estuary server provides a web-based interface that supports three subtypes of interface: tutorial interfaces that display a single structure editor with highly visible help/orientation, solo performance interfaces that display one or more structure editors without much additional adornment, and collaborative interfaces wherein operations on the interface are shared, through the server, to a potentially large "ensemble" of other, connected web browsers.

The design of Estuary's collaborative editing platform was most directly informed by earlier work on extramuros, a browser-based interface for collaborative, text-based live coding that divides the screen into separately evaluable panels. Typically, each panel became a temporary work area for a specific person in an ensemble, an "N editors for N performers" approach also encouraged by Gabber, the networked collaboration-oriented extension to Gibber (Charlie Roberts et al. 2015). Our experience developing and using extramuros confirmed the utility of such a striated, networked interface not only for the original purpose of globally distributed, collaborative live coding performances, but also for use in co-located laptop orchestras and zero-installation workshop settings (Ogborn et al. 2015). Extramuros provides a "panoptical" situation in which everyone can see what everyone else in the group is doing. The idea of projectional editing suggests an amplification of this panopticism: everyone can have access to multiple projections of what everyone else in the group is doing. In aiming at such projectional panopticism with Estuary, the hope is that it will support the development of musical intersubjectivity, which may be particularly fragile in situations, such as electronic music, that are typically less scaffolded by traditional expectations and training (Lagerlöf, Wallerstedt, and Pramling 2014).

Estuary adopts a specific model and terminology for the objects that are the focus of collaborative performance. A single, centralized *Server* keeps track of a certain number of *Ensembles*. Each ensemble consists of a set of *Definitions* and *Views*. Definitions are structured expressions that can be translated into computational effects (for example, expressions modeling TidalCycles patterns that can be translated into streams of sound events) with each Definition in an Ensemble indexed by a numerical Zone. A View is a configuration for viewing and/or editing a given Definition or set of Definitions, with the possibility for defining Views recursively as collections of other Views. A View to be used by a laptop orchestra in performance, for example, might consist of a panelled arrangement of individual Views each of which allows a specific Definition in the Ensemble to be edited. Extending this example, additional Views of the same Ensemble of Definitions could be created in such a way that each Definition from the Ensemble fully occupies a browser window, so that the activity of a live coding ensemble can be shared with an audience via a large collection of display surfaces (but still navigated panoptically by the individual performers using a different view on their laptop). Note: At the time of writing, the basic functionality of multiple views is implemented, but user interface design to dynamically reconfigure and select different views is the focus of work in progress — for the time being a standard build of the Estuary server restricts users to a default view for an ensemble, a restriction that will be lifted shortly.

When changes are made to a given Definition within an Ensemble, these changes are immediately visible on all relevant Views. Changes in this case include not only those that lead to a different structure for evaluation (i.e. translation into a different sounding result) but also user operations that signal an intent to change the structure. Because of the click-only philosophy mentioned above, clicking in blank space frequently produces a pop-up menu that is not only a way to change the structure — the presence of the pop-up menu is treated as an explicit element of the notation. Popping open the menu to change the code is, already, changing the code. In this way members of an ensemble as well as audience members can see (at least in moments where there is a popup menu) the options that any member of the ensemble are considering. This might be considered an extra level of exposure or transparency in live coding performance - sharing not just your code but what your code might become. This also invites comparison with the act of typing out code expressions - where the motion of the cursor and the action of beginning to type suggests to an observer something about what is to come (but not necessarily a finite palette of options as in Estuary).

Knowing who is doing what is a significant challenge of networked collaboration systems (Xambó et al. 2016). In Estuary, this is partially ameliorated by another design feature inspired by performance practices observed with extramuros. It is common, with extramuros, for performers to enter their names in programming comments (ie. secondary notations) in order to "claim" a panel of the interface for their use. Mindful of this, Estuary's views include shared *Labels*: single-line text inputs whose purpose is to allow such secondary notations to be entered and shared across an Ensemble (and thus also with its audience), with no other effect on the system. This solution breaks down, however, if and when people operate on the very same structure. One approach would be to simply disallow people from operating on the same structure, as in Lich.js (McKinney 2014). That rules out a number of significant pedagogical and aesthetic possibilities tied to working in quick succession on the same code, however. Another approach is to colourize notations to denote the user responsible for adding/changing particular elements, as demonstrated in Troop (Kirkbride 2017). It is not straightforward to extend such an approach to the non-text-editing, click-only, border free aesthetic of Estuary, and delineating specific user agency remains an issue to be tackled more substantially in the future.

2.5 Security

Security is an important area of concern for all network music systems, albeit one that is presently under-researched (Hewitt and Harker 2012 is one recent exception). In general, a system needs to balance a requirement to prevent malicious abuse, while also not imposing security measures that defeat the basic purpose of the system. The earlier extramuros system used a single global password, established when the server was launched, and then shared informally with the members of the musical community using the server, who are typically small in number. The password was necessary to evaluate code but not necessary in order to edit, view the code in a browser window, nor receive the code and execute it with a client connected to an interpreter for SuperCollider, TidalCycles or another language. The absence of user-level authentication reduced the complexity of using the system with shifting populations of live coding performers and audiences. With a given running server, one only had to point the audience and performers to a given URL, and then provide the password to the performers (often in the form of an instruction to continue using the same password as before). This also made it easier to deploy extramuros, as there was no need for the software to connect to the database infrastructure that would likely be a part of implementing user-level authentication.

The intention to make the Estuary system a platform that is simultaneously used by a potentially larger number of collaborating Ensembles requires a somewhat more fine-grained approach. In the tradition of extramuros, we have continued to base security measures on shared passwords that are not associated with specific users. Unlike extramuros, however, there are multiple levels of shared passwords for different purposes. No password is required in order to visit an Estuary server, to access a tutorial or solo interface, to visit the lobby of current, collaborative Ensembles, nor to join an Ensemble as an observer. To create a new Ensemble requires a global password, and when creating an Ensemble it can, optionally, be given an ensemble password. If there is an ensemble password, then editing or chatting in that Ensemble requires that the correct password be provided, as does creating or changing the Views available in that ensemble. This approach to security leverages the fact that the web browser is already a "sandboxed" environment wherein security threats are relatively constrained, as well as the fact that live coding performers and audiences are already involved in dense networks of relatively secure communication parallel to their use of the Estuary platform (and thus able to share global and Ensemble passwords in security-conscious ways).

3 Implementation

There are three main parts to the Estuary framework: a server application, a client application, and a sampling engine used by the client. In typical use, the server provides the client application, in response to a request from a web browser, in its entirety (not including sound samples) and then continues to interact with the client application via a WebSockets connection (that is used, for example, to exchange information about edits to collaborative Ensemble). The client provides

a collection of interfaces for editing Definitions and Views, as well as a process for translating Definitions into a stream of sound events. The latter are translated into sound by library calls to the WebDirt sampling engine.

A definitive and distinctive aspect of Estuary's implementation is that both the client and the server have been authored in the functional programming language Haskell, using a specific toolchain for compiling Haskell into JavaScript that runs in the browser (GHCJS) as well as a specific set of libraries for functional reactive programming (Reflex and Reflex-DOM). With the exception of the WebDirt sampling engine and a few Haskell-to-JavaScript foreign function interfaces, the overwhelming majority of the Estuary code base is in Haskell. The decision to centre the project's development in Haskell was motivated by consideration of two advantages: (1) using Haskell allows us to use the entirety of the TidalCycles code base (written in Haskell) directly in the implementation of Estuary; and (2) Haskell's strict compile-time type checking gives us some protection against mistakes, especially mistakes introduced by the frequent refactoring that accompanies the iterative refinement and even outright rejection of earlier design choices.

The Reflex and Reflex-DOM Haskell libraries (hereafter just abbreviated to "Reflex") used by Estuary provide a framework for functional reactive programming (FRP) in the browser. Other web-based FRP platforms exist. We chose Reflex largely on account of its record of successful industrial application. Reflex is also visibly the target of ongoing development efforts and boasts a responsive, supportive online community. Reflex is intended to be used with the GHCJS toolchain, which compiles Haskell code down to Javascript that is then able to run in a web browser. One advantage of the use of GHCJS is that we can reuse Haskell definitions between the client and the server — those definitions concerning the networking protocol to be used, in particular — reducing development mistakes. When the network protocol is changed, those changes are subjected to Haskell's strict type checking, and we can enjoy an elevated confidence that client and server have been changed in corresponding ways. The server itself is not intended to be compiled with GHCJS but rather with the "normal" GHC into a binary application for a given architecture and operating system. However, compilation of the server code with GHCJS into a node.js application remains a possibility (albeit an unexplored one).

The GHCJS + Reflex-DOM platform does introduce its own set of challenges. As a relatively uncommon way of approaching the development of web applications, it imposes a potentially steep learning curve on contributors to the project. In this connection, we found it useful to step back from the implementation of Estuary and conduct sessions in which we hacked together on simpler "How do we do this with Reflex-DOM?" type problems. The difficulty of getting a working build system (with GHCJS plus Reflex-DOM) for Estuary, particularly on common "end-user" operating systems, is a continuing source of difficulty, only partially mitigated by our use of the program stack for managing and building Haskell projects and dependencies. However, this difficulty only affects those who intend to build and directly contribute to Estuary's code base. All others can use compiled Javascript releases, which are effectively cross-platform "binary" releases given the universality of the web browser platform. Given a set of audio samples, an Estuary build can even be loaded into the browser from the local filesystem, in the absence of network or running server.

The earlier extramuros software employed the share.js library for operational transforms in order to support collaborative editing. Observation of the use of the extramuros system suggested that such detailed mechanisms for resolving conflicting edits were not necessary in the case of Estuary. On the one hand, the division of a collaborative interface into multiple panels, expected to be used in a more or less dedicated way by an individual performer, means that conflicting edits will be extremely uncommon. On the other hand, the structures (Definitions) contained in each numbered zone of an Ensemble tend to be relatively compact, an echo of the compactness of Tidal's core notations. As such, Estuary's networked collaboration protocol handles edits by simply broadcasting the new value of a given Definition to all clients who have joined the same Ensemble. Similar "broadcast of the complete state of an entity" measures underpin other aspects of the networked collaboration protocol, such as shared tempo and chat.

A separate library, WebDirt, was developed in plain JavaScript as a sampling engine for Estuary, using the Web Audio API (and no additional libraries) to implement a rough re-creation of the Dirt sampling engine originally created for TidalCycles by Alex McLean (although now largely superceded in the TidalCycles community by the later, SuperCollider-based SuperDirt). Estuary creates a WebDirt object when loaded into the browser, and the Estuary client application calls a method of that object to schedule the playback of a given sample, with specific DSP options, at some point in the future. Estuary's structure editing approach allows a small advantage here in connection with the delivery of audio samples to WebDirt's memory over the network: as soon as a user touches anything connected to the name of a given folder of samples, a "Hint" event is generated and delivered to WebDirt. WebDirt is then able to request and, potentially, receive the audio data of a given sample before it has been definitely placed in the executing structure, i.e. before it is required, thus defeating the possibility for a "missing sound" while the data is transferred from the server to the client. We consider this hinting mechanism, like "zero-latency network music" (Ogborn and Mativetsky 2015) in another situation, to be a way in which the unavoidable temporal deferral at the heart of live coding can be exploited: in effect, we take advantage of things not happening at the same time in order to create the effect that things are happening "now". WebDirt has been released as an independent free-and-open-source software library under the GNU Public License version 3, in the expectation that it will probably be useful for other things (including but not limited to interactive TidalCycles documentation).

4 Deployment and Future Work

At the time of writing, the Estuary system is continuously available for public use from a research web server, and it can be downloaded as a ready-to-run project for deployment on other web servers. The system can also be downloaded and run "standalone", without access to a network (and without multi-user collaboration, of course). Estuary is free software, released under the terms of the GNU General Public License version 3, and is thus both able to incorporate the development efforts of new contributors, and available to be forked into projects or communities with radically different intentions or desires than the existing contributors.

Estuary has reached the point where it is a framework into which additional interface elements can easily be added, in response to general possibilities, but also in response to use cases that arise in particular settings as well as tracking the further development of TidalCycles. While the general categories of tutorial and solo interfaces are there as placeholders, we need to produce a large collection of specific tutorial and solo interfaces for this navigation system to really be useful. Tutorial interfaces in particular need to be implemented in multiple languages. Alongside and allied to that development work, there is a pressing need for in situ evaluation of the software, especially in the workshop and school settings that are most clearly served by the zero-installation, safety-through-structure-editing approach that is emphasized in Estuary.

In terms of Estuary's implementation, there are two areas where future development promises relatively easy gains in terms of performance and stability. Firstly, our use of the Reflex-DOM platform has involved a stable but quite outdated version of that platform. Anecodotal evidence suggests that a refactoring of the code to use recent, "bleeding edge" versions of Reflex-DOM would bring performance advantages. A second area where potential performance and stability gains are possible concerns the WebDirt sampling engine, which currently builds a new synthesis graph for each sound event that is triggered. Refactoring so that WebDirt employs a bank of reusable synthesis graphs would reduce the potential for issues related to garbage collection to interference with DSP performance, although the current Web Audio API's insistence that AudioBufferSourceNode-s can only be used once represents a small caveat to this approach. WebDirt also currently uses ScriptProcessorNode-s for some custom processing of audio samples (i.e. processing in ways not possible with the basic collection of standard unit generators provided by the Web Audio API), an approach that is known to be inefficient but expected to be eventually deprecated in favour of a new approach based on AudioWorker-s that run in a separate thread from the main thread.

At various points in its development, Estuary has been guided by the idea of producing not a definitive interface but rather a platform in which multiple ideas about the interface can coexist. In its current state, that idea has taken the shape of a navigation system that unites tutorial interfaces, solo performance interfaces and collaborative interfaces as different pages of a system unified by its orientation to creating patterns that employ "underlying" TidalCycles definitions. Estuary's tie to TidalCycles, while well-motivated, is somewhat artificial — theatrical, even. We have proposed notations to do (some of) what TidalCycles does which are based on (some of) TidalCycles notations (and which even use some definitions from the TidalCycles codebase) but which are nonetheless not the same as TidalCycles notations. At the same time, to make this work, we are compelled to introduce additional notations that have little to do with TidalCycles, such as those that concern the configuration of the system, collaborative editing, etc, and our overall framework has been assembled in such a way that targeting new domains with new notations would be considerably less work than it has taken to get to this point. That being the case, perhaps the most ambitious and exciting horizon for Estuary will be for it to evolve beyond being a TidalCycles parasite to being a host for a wider collection of notations and domains, unified by a common purpose of making it easier for people to get hooked on live coding.

A continuously updated deployment of Estuary can be accessed and evaluated at the following URL, with the admin password (for creating ensembles) being "morelia" for the duration of the ICLC 2017 review period: http://intramuros.mcmaster.ca:8002

A video of the Cybernetic Orchestra employing a somewhat older version of Estuary for a performance at NIME 2017 in Copenhagen, Denmark can be viewed here: https://www.youtube.com/watch?v=z17OE6YLzNQ&feature=youtu.be

4.1 Acknowledgements

This research was supported by the Social Sciences and Humanities Research Council of Canada (SSHRC) through the Insight Development grant program. Many thanks to Ryan Trinkle and other members of the community around Reflex for assistance at various points, to all of those who have contributed to the ongoing development of TidalCycles, to the members of the Cybernetic Orchestra for being the software's first collaborative users, and to those members of the live coding community who attended a pre-ICLC 2016 presentation of Estuary's structure editing features. We would also like to thank Jacques Carette, Ian Jarvis, Harold Sikkema, Matthew Paine and Tanya Goncalves for their assistance at various stages of this project.

References

Church, Luke, Emma Söderberg, Gilad Bracha, and Steven Tanimoto. 2016. "Liveness Becomes Entelechy - a Scheme for L6." In *Proceedings of the International Conference on Live Coding 2016*.

Davide Della Casa, Davide, and Guy John. 2014. "LiveCodeLab 2.0 and Its Language Livecodelang." In *Proceedings of the 2nd Acm Sigplan International Workshop on Functional Art, Music, Modeling & Design (Farm '14')*, 1–8. ACM. doi:10.1145/2633638.2633650.

Fradkin, Scott. 2016. "A Block Based Music Live Coding Environment for Kids." In *Proceedings of the International Conference on Live Coding 2016*.

Franklin, Diana, Phillip Conrad, Bryce Boe, Katy Nilsen, Charlotte Hill, Michelle Len, Greg Dreschler, et al. 2013. "Assessment of Computer Science Learning in a Scratch-Based Outreach Program." In *Proceeding of the 44th Acm Technical Symposium on Computer Science Education*. ACM. http://dl.acm.org/citation.cfm?id=2445304.

Giannakos, Michail N., Letizia Jaccheri, and Roberta Proto. 2013. "Teaching Computer Science to Young Children Through Creativity: Lessons Learned from the Case of Norway." In *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research*, 103–11. Open Universiteit, Heerlen. http://dl.acm.org/citation.cfm?id=2541927.

Hewitt, S., and A. Harker. 2012. "Security in Network Connected Performance Environments." In *Proceedings of the International Computer Music Conference*, 320–24.

Kafai, Yasmin B., and Quinn Burke. 2013. "The Social Turn in K-12 Programming: Moving from Computational Thinking to Computational Participation." In *Proceeding of the 44th Acm Technical Symposium on Computer Science Education*, 603–8. ACM. http://dl.acm.org/citation.cfm?id=2445373.

Khwaja, Amir Ali, and Joseph E. Urban. 1993. "Syntax-Directed Editing Environments: Issues and Features." In *Proceedings of the 1993 Acm/Sigapp Symposium on Applied Computing: States of the Art and Practice*, 230–37. ACM. http://dl.acm.org/citation.cfm?id=162882.

Kirkbride, Ryan. 2017. "Troop: A Collaborative Tool for Live Coding." In *Proceedings of the 14th Sound and Music Computing Conference*, 104–9. http://smc2017.aalto.fi/media/materials/proceedings/SMC17_p104.pdf.

Lagerlöf, Pernilla, Cecilia Wallerstedt, and Niklas Pramling. 2014. "Playing, New Music Technology and the Struggle with Achieving Intersubjectivity." *Journal of Music, Technology & Education* 7 (2): 119–216. doi:10.1386/jmte.7.2.199_1.

Mahadevan, Anand, Jason Freeman, Brian Magerko, and Juan Carlos Martinez. n.d. "EarSketch: Teaching Computational Music Remixing in an Online Web Audio Based Learning Environment." In *Proceedings of the Web Audio Conference 2015*. http://wac.ircam.fr/pdf/wac15_submission_3.pdf.

Malan, David J., and Henry H. Leitner. 2007. "Scratch for Budding Computer Scientists." In *ACM Sigcse Bulletin*, 39:223–27. ACM. http://dl.acm.org/citation.cfm?id=1227388.

Maloney, John, Leo Burd, Yasmin Kafai, Natalie Rusk, Brian Silverman, and Mitchel Resnick. 2004. "Scratch: A Sneak Preview." In *Creating, Connecting and Collaborating Through Computing, 2004. Proceedings. Second International Conference on*, 104–9. IEEE. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1314376.

McKinney, Chad. 2014. "Quick Live Coding Collaboration in the Web Browser." In *Proceedings of the International Conference on New Interfaces for Musical Expression 2014*, 379–82. https://pdfs.semanticscholar.org/637c/01d3a09a36867ee7730a80baff72831cbc8e.pdf.

McLean, Alex. 2014. "Making Programming Languages to Dance to: Live Coding with Tidal." In, 63–70. ACM Press. doi:10.1145/2633638.2633647.

McLean, Alex, and Geraint Wiggins. 2010. "Tidal—Pattern Language for the Live Coding of Music." In *Proceedings of the 7th Sound and Music Computing Conference*. Vol. 2010. http://server.smcnetwork.org/files/proceedings/2010/39.pdf.

McLean, Alex, Dave Griffiths, Nick Collins, and Geraint Wiggins. 2010. "Visualisation of Live Code." In *Proceedings of the 2010 International Conference on Electronic Visualisation and the Arts (Eva'10)*, 26–30. http://www.bcs.org/upload/pdf/ewic_ev10_s2paper1.pdf.

Miller, Philip, John Pane, Glenn Meter, and Scott Vorthmann. 1994. "Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University." *Interactive Learning Environments* 4 (2): 140–58.

Ogborn, David, and Shawn Mativetsky. 2015. "Very Long Cat: Zero-Latency Network Music with Live Coding." In *Proceedings of the First International Conference on Live Coding*, 159–62. ICSRiM, University of Leeds. doi:10.5281/zenodo.19348.

Ogborn, David, Eldad Tsabary, Ian Jarvis, Alexandra Cárdenas, and Alex McLean. 2015. "Extramuros: Making Music in a

Browser-Based, Language-Neutral Collaborative Live Coding Environment." In *Proceedings of the First International Conference on Live Coding*, 163–69. doi:10.5281/zenodo.19349.

Rizvi, Mona, Thorna Humphries, Debra Major, Meghan Jones, and Heather Lauzun. 2011. "A Cs0 Course Using Scratch." *Journal of Computing Sciences in Colleges* 26 (3): 19–27.

Roberts, Charles, and JoAnn Kuchera-Morin. 2012. "Gibber: Live Coding Audio in the Browser." In *Proceedings of the International Computer Music Conference*, 64–69.

Roberts, Charlie, Karl Yerkes, Danny Bazo, Matthew Wright, and JoAnn Kuchera-Morin. 2015. "SharingTime and Code in a Browser-Based Live Coding Environment." In *Proceedings of the First International Conference on Live Coding*, 179–85.

Tanimoto, Steven L. 1990. "VIVA: A Visual Language for Image Processing." *Journal of Visual Languages & Computing* 1 (2): 127–39.

———. 2013. "A Perspective on the Evolution of Live Programming." In *1st International Workshop on Live Programming (Live)*, 31–34. IEEE.

Voelter, Markus, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. "Towards User-Friendly Projectional Editors." In *Software Language Engineering*, 41–61. Springer. http://link.springer.com/chapter/10.1007/978-3-319-11245-9_3.

Wakefield, Graham, Charles Roberts, Matthew Wright, Timothy Wood, and Karl Yerkes Yerkes. 2014. "Collaborative Live-Coding Virtual Worlds with an Immersive Instrument." In *Proceedings of the International Conference on New Interfaces for Musical Expression 2014*, 505–8. http://www.nime.org/proceedings/2014/nime2014_328.pdf.

Walkingshaw, Eric, and Klaus Ostermann. 2014. "Projectional Editing of Variational Software." In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, 29–38. ACM. http://web.engr.oregonstate.edu/~walkiner/papers/gpce14-projectional-editing.pdf.

Ward, A., J. Rohrhuber, F. Olofsson, A. McLean, D. Griffiths, N. Collins, and A. Alexander. 2004. "Live Algorithm Programming and a Temporary Organisation for Its Promotion." In *Read_me – Software Art and Cultures*, edited by O. Goriunova and A. Shulgin. Aarhus: Aarhus University Press.

Xambó, Anna, Jason Freeman, Brian Magerko, and Pratik Shah. 2016. "Challenges and New Directions for Collaborative Live Coding in the Classroom." In *Proceedings of the International Conference on Live Interfaces 2016*. http://users.sussex.ac.uk/~thm21/ICLI_proceedings/2016/Papers/Long_Papers/94_Live_Coding_Classroom.pdf.