Musical Concurrent Programming With Sonic Pi

Interim Report

Eleanor Vincent

June 2, 2015

Contents

1 Introduction

1.1 Motivation

For many years it has not been a requirement that children should learn much about the vast field of computing during their formative years in UK Education. In 2012, the Government began to re-recognise the signifiance of Computer Science and has replaced the National ICT curriculum with a revised Computing cirruculum. The new Computing Program of Study [13] aims to enable pupils to understand the world of computing, giving them the ability to think logically and apply the fundamental principles of the discpline to their real-world environments.

The movement is not purely restricted to the UK. In the US there is a similarly led campaign calling to recognise the topic as relevant to all contempary sciences. It calls "Computational Thinking" a universally applicable attitude and skill set everyone, not just computer scientists, would be eager to learn and use [28].

There is a general struggle within the humanties courses available in schools to remain relevant in light of a quickly developing technical world. Music schemes within the UK frequently report to have suffered funding cuts and education is focused on learning a variety of specific instruments, with little focus on musical technology.

As well as the necessity of Computer Science and Music within schools, there is an increasing recognition of the power of programming amongst the general populace. A growing hacker and maker movement has been making programming a much more accessible skill [6]; it presents itself as a viable and useful hobby amongst a vast range of ages and professions and this is as much because of the availability of useful resources that would be frequently used in such situations as a school classroom. There is existing research that has gone into the viability of programming tools for professional artists, as reported at PPIG [10, 8], and investigations into the craft practices of existing professional software developers who work in professional art contexts [26].

Sonic Pi is one of many projects designed to support both computing and music lessons within schools. Sonic Pi is an environment for creating live-coded music at a level of complexity which is well suited to a first-programming language [2]. Many available languages are suitable as a first-programming language, but not all attempt to make themselves an inviting gateway into the realm of technical programming and innovation. Sonic Pi seeks to provide an exploratory and invigorating introduction into programming whilst being complex enough to lead a user through into much more complicated programming ideas and use cases with a managable learning curve. By presenting a musical system it seeks to break down the barrier between the technical elite and the hobbyist. In this report Sonic Pi will be presented in terms of the formalisation of its timing effects and the existing concurrency primitives of the language.

1.2 Objectives

In the current version of Sonic Pi there is no means to verify the timing effects or the commu-nication patterns of the pattern. This is to say, user's must judge themselves that the program

is timed correctly and that the program is free from concurrency bugs. The first point can be difficult for novice programmers, potentially moreso for those with little musical experience. The later can be difficult for even some experienced programmers due to the nondeterministic nature of some concurrency bugs.

In building analysis for the timing effects of the program, we hope to develop something that is sufficiently lightweight so that it may eventually be run during a live performance of the program. This leads to interesting challenges in terms of the heuristics used when parsing the program and what solutions will lead to the best possible results. We plan to utilise the speed of the platform and program architecture and efficient design for this matter.

In developing the communication patterns of Sonic Pi we aim to utilise the theory of Session Types and apply them to this context. Session Types will enable us to define a clearly understandable structure to the communication patterns of the program; this will enable the tool to identify difficult bugs such as deadlocks and thrashing behaviour. This is possibly the first application of this concept to the live programming paradigm of which Sonic Pi is a part.

1.3 Report Structure

The remainder of this report is broken down as follows:

2 Background

In this section we begin with a short history of Computing in Schools followed by explaining the ideas and features of Sonic Pi, the live coding IDE that forms the basis of this project. We then move on to explain the subject of both Living Programming and Session Types in further detail and seek to relate them back to the current aims of the project. We conclude with details on some of the related work in these areas of research.

2.1 Computing in Schools

Some of the earliest significant pieces of work towards the education of computing started with the invention of Logo, an adaptation of the LISP language, most remembered for its use of "turtle graphics". Some time after this came the Computers in the Cirriculum Project, funded from 1972 to 1991 by the Schools Council with addition aid from the Microelectronics Education Programme introduced in 1981. The first microcomputers appeared in both UK Primary and Secondary Schools in 1979. The Commodore Pets aided both spelling and arthimetic practice as well as the ability to teach either BASIC or Logo. The 1980's saw a huge amount of legislative reform and technical efforts to give young people the ability to work with computers. Unfortunately, over the course of the late 80's and early 90's, this focus on programming ability gave way to simply the education of practical use of existing computer software; there was little to no focus on the programming fundamentals behind these applications [1].

In 2013, Ofsted published a report documenting their findings relating to ICT in UK schools from 2008 to 2011 and found that in half of all secondary schools, school leavers had not been given adequete education to move into a technical career in their future. In 2007, 81,100 pupils were enrolled in the ICT GCSE but this had fallen to 31,800 pupils by 2011 [12] with a notable lack in the education of key skills such as computer programming itself. This lack was found to be as much a lack in knowledge from the teacher's as much as the cirriculum's failure to address the issues.

2.2 Sonic Pi

Sonic Pi is an imperative live programming language designed as an educational first language. It is a Ruby-based Domain-Specific Language designed for manipulation of synthesisers through time [4]. It is interpreted through a VM rather than being directly compiled. It is currently in its second iteration, with the main extension between the two languages being the work done to improve the timing system of the project, which is discussed in more detail below. Sonic Pi is built on top of the SuperCollider synthesis server to enable it to define and manipulate synthesisers in real time; an important feature for a musical language. Some of the concepts that Sonic Pi is well suited to teach, in direct relevance to the current UK Computing in Schools Cirriculum, are conditionals, iteration, variables, functions, algorithms and data structures. Sonic Pi also extends beyond these concepts to include such things as multi-threading and hot-swapping of code as these are likely to be of crucial importance in the future of programming contexts [5].

This section first gives a brief description of the Raspberry Pi then explores the implementation of Sonic Pi V1.0, mainly to give appropriate context to the timing system that Sonic Pi V2.0 currently imeplments. There is then be some short discussion on the particular features of Sonic Pi V2.0.

2.2.1 Raspberry Pi

The Raspberry Pi was developed as a very low cost computer system to enable technical experimentation amongst people who have little access to computers they could tinker with or expose the programming layer. The idea came about in 2006 as an answer to the steadily decreasing levels of pupils applying to take up Computer Science after their A-Levels. The reasons for this were attributed to many contributing events such as the end of the "dot com" boom, the focus of IT lessons on Microsoft software and building very basic HTML websites and the increased availability of out-of-the-box games consoles over the Amigo, BBC Micro, Spectrum ZX and Commodore 64 machines that promoted individual experimentation so freely in the past [3].

The Pi is provided as a bare circuit board costing roughly $25, able to boot into a Linux environment with very little other commerical equipment required. The Raspberry Pi Foundation is a non-profit organisation and has sold over a million products since 2012. The main objective is to develop genuine technical competency by allowing the freedom to experiment with the whole system rather than taking the locked box approach of other systems. Learning becomes self directed for pleasure rather than at the behest of a mark molded system. It is this style of engagement that brought the Raspberry Pi to the attention of educational campaigners and has since enabled it to be used so successfully within the new movement towards better Computing education within Schools.

2.2.2 Sonic Pi V1.0

Sonic Pi V1.0 was designed to port the language Overture to the Raspberry Pi in order to focus on driving specific educational objectives; the idea in mind was to teach a target audience of 12-year-olds that had no previous experience with programming; it would bring them from the state of "this is a computer" through to the ability to write a full length program, that generated satisfying music, over the course of a few weeks. Sonic Pi is a Ruby-based language both due to the author's existing experience with the language and to keep it in line with the languages already used within the industry. Python is well regarded as an education language and the semantic similarity between the two makes Ruby easy to defend as a language choice.

```
play 60
play 61
play 62
```

Snippet 1: Chord Form: Successive Notes

```
play 60
sleep 0.5
play 61
sleep 0.5
play 62
```

Snippet 2: Arpeggio Form: Sleep Separation

The immediate feature that Sonic Pi focuses on is sequential ordering in imperative programs; demonstrated in musical theory by the sequential playing of notes. Code Snippets 1 and 2 demonstrate two small Sonic Pi programs. The first demonstrates the situation where the MIDI notes would be played together. Sonic Pi v1.0 takes advantage of fast clockspeeds of modern processors in order to assume the sequence of instructions would execute quickly enough to sounds together. In order to separate them into sequential notes they must be separated with a sleep instruction as demonstrated by Snippet 2. The notation for sleep in Sonic Pi V1.0 is similar to that of the POSIX sleep operation [18]. The default sound that Sonic Pi uses in these contexts is a pleasant bell sound from the SuperCollider synthesiers.

It can be seen that the ability to skip the arguably verbose nature of structured syntax makes it much more relevant in the contexts of a classroom; pupils are able to start creating meaningful programs with much more speed. From the point of view of providing any future debugging interfaces, it has a small benefit in removing the code overhead for finding and correcting such small syntactical issue such as missing ;.

These semantics work well in an educational context but do not allow for the correct timing of musical notation which puts them at odds with user expectations. To demonstrate this more concretely, below are two further code snippets demonstrating Sonic Pi's basic threading abilities. Sampling is a feature of Sonic Pi V2.0 but is presented here in the context of V1.0 programs as the two versions are syntaxtically similar.

In Snippet 3 the desired outcome is to play MIDI note 60 together with the drum kick with an interval of 0.5s between each hit. Unfortunately, this does not take into account the execution time of each statement; there is a short amount of execution time for each line of code, plus the desired sleep of 0.5s. Because of this the rhythm will gradually shift with each loop as the clock time and the "virtual time" becomes further out of sync. The actually length of time each line execution takes is variable between processors depending on their speed and overall load. Regardless, we can see that each loop in Snippet 3 will actually take longer than the desired 0.5s.

```
loop do
    play 60
    sample :drum_heavy_kick
    sleep 0.5
end
```

Snippet 3: Repeating Bass and Drum

This is further apparent when running Snippet 4. The desired outcome is to play the drum kick every second and the MIDI 60 note at every half second, so the two notes will play at the same time every second MIDI note, the threads remain synchronised. On running the threads it is quickly apparent that this is not the true result; due to differing execution times the thread rhythms drift apart very quickly.

```
in_thread
    loop do
        play 60
        sleep 0.5
    end
end

in_thread
    loop do
        sample :drum_heavy_kick
        sleep 1
    end
end
```

Snippet 4: Concurrent Threads

The *play* and *sample* calls are asynchronous and this compounds the present timing issue due to additional costs in sending and interpreting the messages. These issues are summarised in Figure 1. The left-most column represents the real computation time of the statement whilst the right-most column shows the point at which each statement would be run. Each statement duration is unique as processor speed and system load variations affect the duration of each statement separately. The durations are therefor non-deterministic in nature and also not consistent across different runs of the same program.
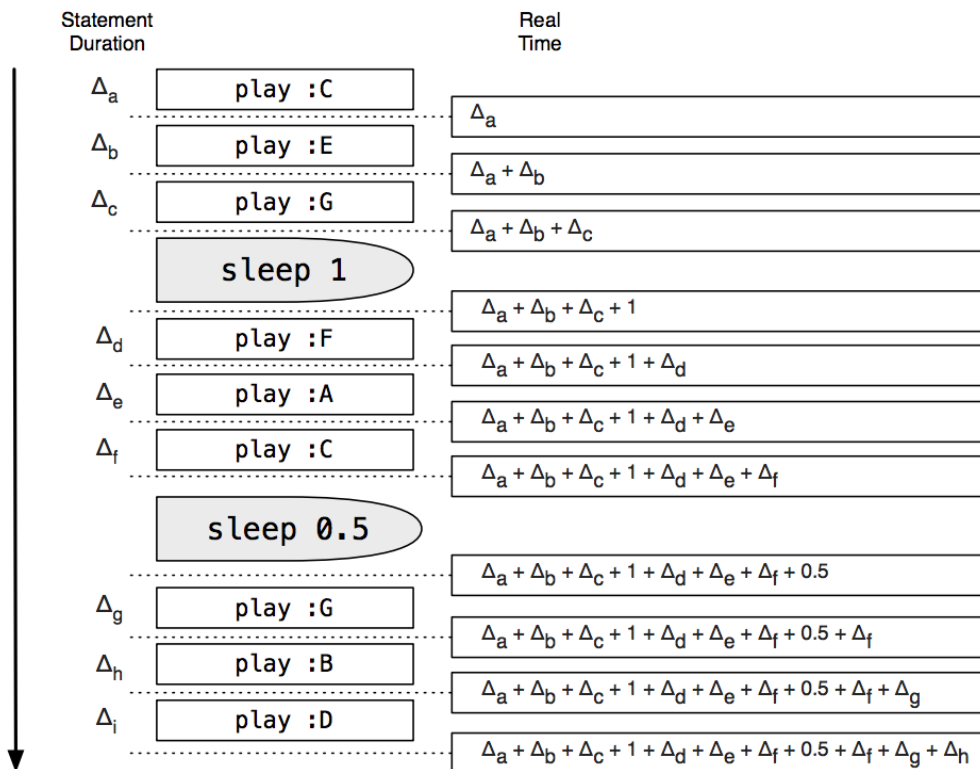

Figure 1: Timing in Sonic Pi V1.0 [5]

2.2.3 Sonic Pi V2.0

Timing Sleep

Sonic Pi V2.0 sought to address this temporal issue and introduces the interesting concept of a *time system* and associated *time safety*, which draw an analogy from the familiar programming concepts of *type systems* and *type safety*. As noted before, V2.0 maintains syntactic compatability with V1.0, so it conceptually as useful to apply in the educational context as before. The power behind V2.0 is that its temporal semantics now react as a user would expect them to, making it a viable program for the musically experienced who have some concept of what they want to achieve as well as for those beginners who desire to learn.

In Sonic Pi V2.0 the sleep command no longer mimics the POSIX command as commented

earlier. Instead the programming model allows a separation of the ordering of effects from the timing of effects. Snippet 5 shows a simple example that combines the different kind of effects; parallel, timed and ordered effects. It also demonstrates further syntactic abilities of V2.0 as we are now treating the code snippets as V2.0 programs.

```
play :C ; play :E ; play :G
sleep 1
play :F ; play :A ; play :C
sleep 0.5
play :G ; play :B ; play :D
```

Snippet 5: V2.0 Program: Playing three chords (C major, F major, G major)

Each chord line demonstrates Sonic Pi's ability to play notes in parallel, the system still taking advantage of the capabilities of modern processors. `sleep` now acts as a "temporal barrier" between statements; it works by blocking computation from proceeding until the given time has elapsed *since the program began running*. It does *not* block from the end of the notes played. In terms of Snippet 5, this means that the second chord is played once one second has elapsed and the third chord will not be played until 1.5 seconds have elapsed. One can think of `sleep` as an "at least" timing. In other words, once `sleep t` has been evaluated, we can state that at least `t` seconds have elapsed since the last `sleep` statement was called. This is similar to how the language ChucK handles the interaction of time using its (multiply) overloaded `=>` operator [27].

For completeness it is worth noting that Snippet 5 demonstrates the ordered effect as the three chords are played in the order written.

The semantics introduced here are achieved by implementing the concept of "virtual time". In Sonic Pi V2.0, virtual time is a thread-local variable that is only advanced by a new `sleep` command, which means that the programmer has explicit control over the timing of the program. Each thread maintains access to both the real time elapsed and the virtual time elapsed whilst running a given program and used the virtual time variable to scheduled requested effects. In order to keep time with the explicit timing requirements of the program the `sleep` command will take account of the execution time between the last `sleep` statement and the current execution point. Referring back to Snippet 5, at the point at which the program executes the second `sleep` command, if execution of the F major chord took 0.1s of execution time then the program will only sleep for 0.4s. This is to ensure there is no rhythm drift; the only overhead in the rhythm is from the play statements following the last `sleep` executed. This is demonstrated visually with Figure 2.

To deal with the non-deterministic execution times within a sleep barrier, and also to deal with the time cost for the synthesiser to schedule output effects, a constant `scheduleAheadTime` value is added to the current virtual time for all asynchronously scheduled effects. If the execution time between `sleep` commands never exceeds this value then the temporal requirements of Sonic Pi are met.

It is possible that the time between `sleep` commands may over-run - this may be common
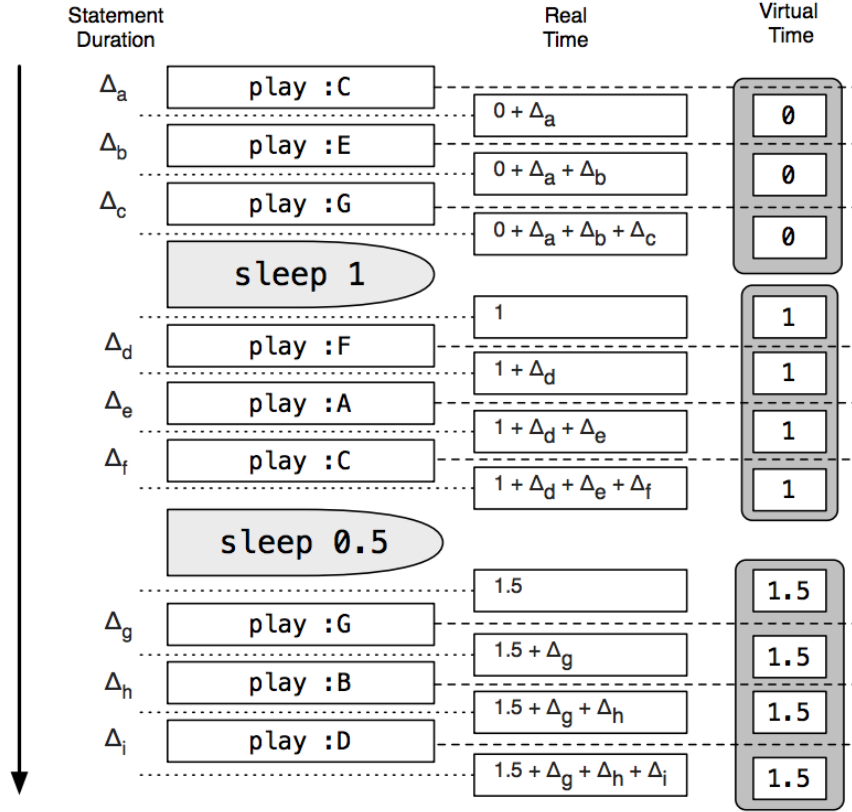
Figure 2: Timing in Sonic Pi V2.0 [5]

in the event someone requested a short `sleep` time such as 0.1s or even 0.05s. In this case, the described programming model is not useful for providing hard deadlines but can function with "soft" deadlines (along the vein of Hansson and Jonsson [14]). In the event a thread falls behind in execution time then the user is given explicit warnings. Should the amount of time the thread is behind exceed a specific fall -behind value, as defined within Sonic Pi, then the system will stop that thread and throw a time exception. This provides essential temporal information about the program and its behaviour to the users. This is a positive in terms of educational use but is something a live coder will aim to avoid during a real performance. This feature also provides a safety mechanism against common errors such as placing isolated `play` calls between `sleep` commands which would have the problem of taking up all of the system resources; instead the thread self-terminates and allows any other threads to continue executing.

Threading

Threading has already been introduced breifly; this section focuses on the further improvement that Sonic Pi V2.0 brings to its threading primitive. Within V2.0 there are multiple commands which allow for easy synchronisation of threads whilst the program is running. Threads also implement thread- based inheritance wherein they take all of the synthesiser settings of the thread they were spawned from.

The keyword `in_thread` enables Sonic Pi users to run pieces of code concurrently. Threads can be named as shown in Snippet 6 in a similar way to how functions are named in Sonic Pi. The simple nature of its loop syntax enables this to be as easy a concept to grasp as the previously defined temporal semantics. These threads alone do not allow for the live coding that Sonic Pi has been created for. With a simple `in_thread` it is more verbose to change the program sound as the program is running. For this, Sonic Pi V2.0 provides `live_loop`.

An interesting point to note about Sonic Pi V2.0 is that actually running the program starts the current program in another thread. Because of this one can actually press run multiple times and have the program layered over the top of itself. There are no particular synchronisation primitives defined over these "global" threads, but it makes for an interesting performance if timed correctly.

```
in_thread(name: :bass) do
    loop do
      use_synth :prophet
      play chord(:e2, :m7).choose, release: 0.6
      sleep 0.5
    end
end

in_thread(name: :drums) do
    loop do
      sample :elec_snare
      sleep 1
    end
end
```

Snippet 6: Named Threads

`live_loops` can be named much more cleanly than `in_threads` and are where Sonic Pi really comes into its live coding roots. Whilst running a program, Sonic Pi's `live_loops` will automatically update the program without skipping any beats. This gives the users a great amount of freedom to experiment with different sounds without having to stop the program and wait for anything to compile and run, as is the case with standard programming. This feature, however, is directly affected by the previously described temporal semantics, in that loops can easily become out of time with each other while a performance is happening. To combat this, Sonic Pi provides synchronisation semantics in the form of the `cue` and `sync` commands. Each time a live loop loops it will generate a new `cue` event which we are able to `sync` on to.

```
live_loop :foo do
    play :c1, release: 8, cutoff: rrand(70, 130)
    sleep 8
end
```

Snippet 7: Live Loops

The Snippets to the right demonstrate a rough workflow of how a user would use the `cue` and `sync` features. To begin with we assume the loops `foo` and `bar` are out of time.

To work with the current example one would begin to fix the situation by changing the sleep time in `foo` to 0.5s. Most likely, this will still sound incorrect. This is because the two loops are likely to now be out of time with each other. Both `foo` and `bar` are producing `cue` events, but these are not being used and so both loops are running with no regard to the other. We can fix this by *syncing* one thread to the other, so that it will only fire when the other thread has looped. In this case we have synced `bar` onto `foo`.

This gives way to some very obvious deadlock scenarios (shown in Snippet 10) that users must avoid. If you create two live loops, `one` and `two` writing a situation that results in either `cue :one` in the second thread and `cue :two` in the first, or similar with `sync` then this will result in a deadlock as both threads wait for the call from the other.

```
live_loop :foo do
    play :e4, release: 0.5
    sleep 0.4
end

live_loop :bar do
    sample :bd_haus
    sleep 1
end
```

Snippet 8: Out of Sync Live Loops

```
live_loop :foo do
    play :e4, release: 0.5
    sleep 0.5
end

live_loop :bar do
    sync :foo
    sample :bd_haus
    sleep 1
end
```

Snippet 9: Synced Live Loops

```
live_loop :foo do
    cue :bar
    play :e4, release: 0.5
    sleep 0.5
end

live_loop :bar do
    cue :foo
    sample :bd_haus
    sleep 1
end
```

Snippet 10: Deadlock - both loops waiting for cue

The IDE

The IDE is as much a part of the educational experience as the language itself. Sonic Pi features a bespoke environment that contains only the bare minimum features required to enable pupils to start coding quickly and with minmal confusion. For this reason the IDE only consists of

Figure 3: Sonic Pi V2.1.1 IDE - Apple Mac View

five key components:

- Control Buttons

- Workspace Tabs

- Editor Pane

- Information Pane

- Error Pane

Sonic Pi originally choose not to have a file save system as the workspaces automatically save the current session. There were no other usual IDE features such as project structure, macro system, refactoring wizard, etc... This is to keep the system as simple as possible to make it quick for young pupils and non -IT teachers to get to grasps with.

Since then, in Sonic Pi V2.1.1, it has added the ability to save the current workspace to your file system, the ability to record what is playing and save as a .wav file type, basic formatting buttons, the ability to import other music files as custom samples, information and tutorial sections, all whilst remain simplistic and easily useable. Features are non-intrusive but incredibly useful in terms of allowing those pupils who thrive with the system to continue to experiment outside the scope of the regular cirruculum lessons.

2.3 Live Programming

For much of the prevailing history of programming there has been an idea that the programmer is inherently separated from the system that they are producing. The task of the programmer is to create a system based on some formal specification that will take effect at some unknown point in the future, and the time between implementation and action has no effect on the results that the system will produce. In this way, there is a strong sense of separation between the program, process and task domains where the progam is the code implementation and specifications, the process is the running of the code on a specific machine and the task is the visible real world results [25]. This is a viewpoint that many would not think to challenge as it is natural to assume that the methodology of a computer programmer would naturally lend itself to implementation of actions that were set for execution in the future and, in general, would process a deterministic set of results that can be repeatedly used as the users required.

Live programming (also referred to as With Time Programming or Just In Time Programming) seeks to apply the improvisional nature of time to the existing methodology of the programmer. With this idea it becomes possible to define a tigher system of feedback between the program and task domains through means of whatever process domain is most suitable. The improvisional nature of the acitity also removes the inherent requirement of a specific program specification and allows for new level of freedom and creativity in the programs being created.

Given the nature of Live Programming the languages that invoke it are often dynamic languages which allow for the flexibility, conciseness and ease of development [20] to enable the act of live programming to feel as natural as standard programming practice.

Live Programming lends itself also to acts of performance, where Live Coders perform to live audiences, often producing such things as live improvisational music or artwork, whilst having some means by which the audience will also see the code written at the same time. For a Live Coder there is frequently no desire to create a final software product or even a set musical score as live programming is about the experimentation rather than the manufacture. From a social and cultural viewpoint, Live Programming lends itself to breaking down the barriers that have been built up between software technology and the creative users [21].

It is interesting that the user's level as a musician will have specific impacts on their experience with the systems in use in the context of live programming and audio languages. Musical scores provide an implicit time representation whilst most musical languages tend to use explicit temporal structures. This generally makes the representation of rhythm within the language guarantee only the ordering of the notes and not the time elapsed between playing each one; the temporal structure provides no clear guarantee as to execution length. In terms of musical experience, non-experienced users may be able to identify an odd sound within this system but be unable to pinpoint exactly what causes the problem.

The question of time and temporal semantics within computing has been in existence for some time, but live programming as a field is a very young research area, with most of the popular languages appearing over the last decade. This comes as part of a wider movement of programming reaching more of the general populace and as it is applied with more and more creative outlets in mind, this separate way of thinking about the programming environment is likely to produce more interesting projects as time develops.

2.4 Session Types

Over recent years there has been a massive increase in the amount of communication based systems, from network protocols over the Internet to server -client systems in local area networks to distributed applications on a global scale. The crucial observation amongst all of these systems is that while there may be some way to describe a one-time interaction between processes there is no real construct to structure a series of reciprocal interactions between two parties [16].

Session Types present a solution to the issue of structing communication-based software. In a similar way to how Object Oriented paradigms sought to solve the issues presented by large scale system written largely with spaghetti code, Session Types seek to restructure existing complex behaviours in a manner which is more lucid, readable and ultimately more easy to verify. Honda et al [16] present this in terms of simple concurrent primitives that build up a basic structuring method for communication-based concurrent programming. In terms of this project, we will initially be using simple communication primitives to check the program for such concurrency bugs as deadlocks. Sonic Pi appears a simple language to apply the theory of Session Types to but the dynamic and concurrent nature of the language give it some interesting communication patterns to formalise. Session Types are also designed in a language agnostic manner, meaning it will be possible to apply to Sonic Pi's Ruby-like syntax.

Session Types consist of the following key ideas:

A basic structural concept known as *sessions*. These are designated via *channels*. The collection of session interactions is what constitutes a program; those interactions are performed via the channels. As well as the session, other concurrent programming contructs are provided: parallel composition, name hiding, conditional and recursion. The combination of recursion and sessions allow for the expression of an unbounded thread of interaction as a single abstraction unit.

Three basic communication primitives that all other structures will be built from: *value passing* - standard synchronised message-passing, *label branching* - purified method invocation, devoid of value passing - and *delgation* - the passing of a channel to another process. Alongside sessions, these allow for complex communication structures to be defined and described with clarity.

Finally there is a basic type discipline for the communication primitives. Without this there would be no way to guarantee the typability of a program, ensuring that two communicating processes always have compatible patterns of communication. It is the incompatibility of inter-action patterns that is one of the main reasons for bugs in commmunication-based programming.

2.4.1 Binary Session Types

Syntax Summary

The base sets of the syntax used to describe session types are as follows:

names: ranged over by *a,b...*
channels: ranged over by *k, k'...*
variables: ranged over by *x, y...*

constants: ranged over by *c, c'...*
expressions: ranged over by *e, e'...*
labels: ranged over by *l,l'...*
process variables: ranged over by *X, Y... u, u'...*

processes, ranged over by `P, Q...`, are given by the following grammar:

```
P ::= request a(k) in P          session request
    | accept a(k) in P           session acceptance
    | k![ẽ]; P                    data sending
    | k?(x̃) in P                  data reception
    | k ◁ ℓ; P                    label selection
    | k ▷ {ℓ₁:P₁[] ... []ℓₙ:Pₙ}   label branching
    | throw k[k₀]; P             channel sending
    | catch k(k₀) in P           channel reception
    | if e then P else Q         conditional branch
    | P | Q                      parallel composition
    | inact                      inaction
    | (νu)P                      name/channel hiding
    | def D in P                 recursion
    | X[ẽk̃]                       process variables

D ::= X₁(x̃₁k̃₁) = P1 and ...      declaration for recursion
          and Xₙ(x̃ₙk̃ₙ) = Pₙ
```

"|" is the weakest association; the others listed have the same association. Parenthesis denote binders which bind the corresponding free occurrences. We use standard simulataneous substitution, written . The sets of free names/channels/variables/process variables of, for example program P, are written as `fn(P)`, `fc(P)`, `fv(P) and fpv(P)` respectively. Processes without free variables or free channels are called programs.

Operational Semantics

The consise definition of the operational semantics of session types requires the definition of the concept *structural congruence*, ≡ [7, 19], as the smallest congruence relation on processes. The following equations are presented as examples:

$$P \equiv Q \text{ if } P \equiv_\alpha Q$$
$$P \mid \text{inact} \equiv P,$$
$$P \mid Q \equiv Q \mid P,$$
$$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$$
$$(\nu u)\text{inact} \equiv \text{inact},$$
$$(\nu u u)P \equiv (\nu u)P, \quad (\nu u u')P \equiv (\nu u' u)P,$$
$$(\nu u)P \mid Q \equiv (\nu u)(P \mid Q) \text{ if } u \notin \text{fu}(Q),$$
$$(\nu u)\text{def } D \text{ in } P \equiv \text{def } D \text{ in } (\nu u)P \text{ if } u \notin \text{fu}(D)$$
$$(\text{def } D \text{ in } P) \mid Q \equiv \text{def } D \text{ in } (P \mid Q) \text{ if } \text{fpv}(D) \cap \text{fpv}(Q) = \emptyset$$

```
    def D in (def D' in P) ≡ def D and D' in P
                                    if fpv(D) ∩ fpv(D') = ∅
```

The operation semantics are then represented using the *reduction relation*, $\rightarrow$. It is assumed the *evaluation relation*, $\downarrow$, is given and $\rightarrow^*$ represents the reflexive, transistive closure of $\rightarrow$. Reduction is the smallest relation on processes and is demonstrated by the following equations:

```
 [Link]   (accept a(k) in P1) | (request a(k) in P2)
             → (νk)(P1 | P2)
  [Com]   (k![ẽ]; P1) | (k?(x̃) in P2)
             → P1 | P2[c̃/x̃]                 (ẽ ↓ c̃)
[Label]   (k ◁ ℓᵢ; P) | (k ▷ {ℓ₁:P₁[] ... []ℓₙ:Pₙ})
             → P | Pi                         (1 ≤ i ≤ n)
 [Pass]   (throw k[k₀]; P₁) | (catch k(k₀) in P₂)
             → P1 | P2
  [If1]   if e then P1 else P2 → P1       (e ↓ true)
  [If2]   if e then P1 else P2 → P2       (e ↓ false)
  [Def]   def D in (X[ẽk̃] | Q)
             → def D in (P[c̃/x̃] | Q)      (ẽ↓c̃, X̃ (x̃k̃) = P ⊂ D)
 [Scop]   P → P' ⇒ (νu)P → (νu)P0
  [Par]   P → P' ⇒ P | Q → P' | Q
  [Str]   P ≡ P' and P' → Q' and Q' ≡ Q ⇒ P → Q
```

Basic Concepts

The central idea to presenting structure is the *session*. A *session* is a set of reciprocal interactions between two parties and serves as a unit of abstraction for describing that interaction. Communications of sessions are done via a *channel* and can be initiated as follows:

```
request a(k) in P    accept a(k) in P    : Initiating a Session
```

The `request` first requests the initiation of a *session* (with *name a*) as well as the generation of a fresh *channel* (with *name k*) which `program` P will then use for later communication. On the other hand, `accept` receives the request to initiate the session a and generates channel k for use in P. In this function, parenthesis and the keyword `in` show scope binding.

Via this generated channel, we can then perform three kinds of atomic operation:

```
1) k![e₁, ..., eₙ]; P      k?(x₁, ..., xₙ) in P
2) k ◁ ℓ; P                k ▷ {ℓ₁:P₁[] ... []ℓₙ:Pₙ}
3) throw k[k₀]; P          catch k(k₀) in P
```

1. Data Sending/Receiving

2. Label Selection/Branching

3. Channel Sending/Receiving

Data sending/receiving is the standard synchronous message passing. e_i denotes expression and we assume the x variables are all distinct.

Brancing/Selection is the minimisation of method invocation in object- based programming. The labels are assumed to be pairwise distinct.

Channel Passing, also referred to as *Delegation*, passes a channel which is being used in a session to another process, enabling us to radically chance the structure of a session. *Delegation* is the generalisation of the concept of the same name in object-based programming [29].

We futher define the session types with the following standard constructs, organised by session:

```
1) P1 | P2
2) (va)P (vk)P
3) if e then P else Q
4) def X₁(x̃₁ k̃₁) = P₁ and ... and Xₙ(x̃ₙ k̃ₙ) = Pₙ in P
```

1. Concurrent Composition

2. Name/Channel Hiding

3. Conditional

4. Recursion

Hiding declares a name/channel to be local in its scope (here P). Whilst not used in regular programming it is needed in the Operation Semantics of session types. Recursion could be achieved through replication but multiple is preferable for well-structured programs.

2.4.2 Multi-Party Session Types

Multi-Party Asynchronous Session Types are a class of behavourial types specifically targeted at describing protocols in distribute systems based on asychrounous communication[11]. As described previously, communication interactions are intended to occur within the scope of many private channels following strict protocols which we have labelled as *sessions*. In it's simplest form this takes place between just two peers, hence our previous focus on "binary" session types (also known as "dyadic"). In practice, a session can involve a variable number of peers and so we extend the concept of session types and communication protocol descriptions to involve the idea of *multi-party* session types.

Multi-party session types have extended binary session types in a manner that retains the intuitive natures of the syntax of the interactions. In binary session types this came from the inherant notion of "duality" in the interactions; a notion that is no longer effective in multi-party sessions as the whole conversation between processes cannnot be constructed from a single behaviour. To handle this, multi-party session types introduces the concept of a global type; an abstraction of the global scenario, whereby we can construct the local types of each process and again ensure the composability of the interactions[17].

First we will outline the formal syntax and operational semantics of multi-party session types. After this we will detail the concepts behind multi-party session types by way of examples.

Extended Syntax

The base sets for multi-party session types are as found in the binary definition with one addition:

participant number: ranged over by *p, q...*

We also define *channel with role*, $s[p]$, as a runtime requirement (detailed fully after the refined process grammar).

Process syntax, such as session request/acceptance, value send/receive, and channel delegation/reception, has been updated to reflect the multi-party nature of this communication. As in [11], the calculus described here uses communication actions that specify exactly one receiver (rather than a non-empty set of receivers) and recursive definitions with exactly one data parametre and one channel parametre. Presenting the syntax in this manner simplifies the presentation of the material as allowing multiple receivers is largely a matter of syntactic sugar. The grammer for processes is presented next, with unchanged definitions ommitted for brevity:

```
P  ::= ū[p](y).P              multicast request
     | u[p](y).P              multicast accept
     | c!<p,e>.P              value sending
     | c?(p,x).P              value reception
     | c!<<p,c'>>.P           channel delegation
     | c?((q,y)).P            channel reception
     | c ⊕ <p,l>.P            selection
     | c & (p,{lᵢ:Pᵢ}ᵢ∈I)     branching
     | X <e,c>                process call
     | (νa)p                  service name hiding
     | (νs)P                  session hiding
     | s : h                  message queue
```

Session hiding and message queue are elements of *runtime syntax*, as was channel with role. The operational semantics of session types are defined by reduction rules, given in the next section. With these, it is helpful to define these runtime states to account for communication situations that arise within a programs lifetime but are not necessarily visable from the program's source. Message queues are used to enable the modelling of TCP-like asynchronous communications.

In (νs)P all occurances of s[p] and the queue s inside P are bound. A process is *closed* if the only free names in it a are service names. fn(Q)(fn(D)) is the set of free names in Q(D). The set of process variables declared in D and the set of process variables that occur free in Q are dpv(D) and fpv(Q) respectively.

19

Extended Operational Semantics

In addition to the standard rules of structural congruence, multi-party session types include a rule for rearranging messages in a queue when the senders or receivers are not the same:

$$
\begin{array}{c}
\texttt{P | 0} \equiv \texttt{P} \\
\texttt{P | Q} \equiv \texttt{Q | P} \\
\texttt{(P | Q) | R} \equiv \texttt{P | (Q | R)} \\
(vr)\texttt{P | Q} \equiv (vr)\texttt{(P | Q)} \text{ if } r \notin \texttt{fn(Q)} \\
(vr)(vr')\texttt{P} \equiv (vr')(vr)\texttt{P}, \; (va) \; \texttt{0} \equiv \texttt{0}, \; (vs)\texttt{(s:}\emptyset\texttt{)} \equiv \texttt{0} \\
\text{where } r ::= a \mid s
\end{array}
$$

$$
\begin{array}{cc}
\texttt{def D in 0} \equiv \texttt{0} & \\
\texttt{def D in } (vr)\texttt{P} \equiv (vr)\texttt{def D in P} & \text{if } r \notin \texttt{fn(D)} \\
\texttt{(def D in P) | Q} \equiv \texttt{def D in (P | Q)} & \text{if } \texttt{dpv(D)} \cap \texttt{fpv(Q)} = \emptyset
\end{array}
$$

$$
\texttt{def D in (def D' in P)} \equiv \texttt{def D' in (def D in P)}
$$
$$
\text{if } (\texttt{dpv(D)} \bigcup \texttt{fpv(D)}) \cap \texttt{dpv(D')} = \texttt{dpv(D)} \cap (\texttt{dpv(D')} \bigcup \texttt{fpv(D')}) = \emptyset
$$

$$
\texttt{s:h·(q,p,}\xi\texttt{)·(q',p',}\xi'\texttt{)·h'} \equiv \texttt{s:h·(q',p',}\xi'\texttt{)·(q,p,}\xi\texttt{)·h'} \text{ if } \texttt{p} \neq \texttt{p'} \text{ or } \texttt{q} \neq \texttt{q'}
$$

Next we present the rules for reducing processes where $\rightarrow^*$ and $\rightarrow^k$ have the usual expected meaning. These rules describe the initialisation of sessions, value passing, output rules and their complementary input rules. These operations ensure that the senders match and that the message is meant for the specified receiver:

$$
\begin{array}{ll}
\texttt{[Init]} & \texttt{a[1](y).P}_1 \texttt{ | ... | a[n-1](y).P}_{n-1} \texttt{ | } \bar{a}\texttt{[n](y).P}_n \\
& \rightarrow (vs)\texttt{(P}_1\texttt{\{s[1]/y\} | ... | P}_{n-1}\texttt{\{s[n-1]/y\} | P}_n\texttt{\{s[n]/y\} | s:}\emptyset\texttt{)} \\
\texttt{[Send]} & \texttt{s[p]!<q,e>/P | s:h} \\
& \rightarrow \texttt{P | s:h·(p,q,}v\texttt{)} \hspace{4em} (e \downarrow v) \\
\texttt{[Deleg]} & \texttt{s[p]!<<q,s'[p']>>.P | s:h} \rightarrow \texttt{P | s:h·(p,q,s'[p'])} \\
\texttt{[Sel]} & \texttt{s[p]}\oplus\texttt{<l,q>.P | s:h} \hspace{2.5em} \rightarrow \texttt{P | s:h·(p,q,l)} \\
\texttt{[Rev]} & \texttt{s[p]?(q,x).P | s:(q,p,}v\texttt{)·h} \rightarrow \texttt{P\{}v\texttt{/x\} | s:h} \\
\texttt{[Srev]} & \texttt{s[p]?((q,y)).P | s:(q,p,s'[p'])·h} \\
& \rightarrow \texttt{P\{s'[p']/y\} | s:h} \\
\texttt{[Branch]} & \texttt{s[p]\&(q, \{l}_i\texttt{\}:P}_i\texttt{)}_{i \in I}\texttt{) | s:(q,p,l}_j\texttt{)·h} \\
& \rightarrow \texttt{P}_j \texttt{ | s:h } (j \in I) \\
\texttt{[If-t]} & \texttt{if e then P else Q} \rightarrow \texttt{P} \hspace{4em} (e \downarrow \texttt{true}) \\
\texttt{[If-f]} & \texttt{if e then P else Q} \rightarrow \texttt{Q} \hspace{4em} (e \downarrow \texttt{false}) \\
\texttt{[Proccall]} & \texttt{def X(x,y) = P in (X<e,s[p]> | Q)} \\
& \rightarrow \texttt{def X(x,y) = P in (P\{}v\texttt{/x\}\{s[p]/y\} | Q) } (e \downarrow v) \\
\\
\texttt{[Str]} & \texttt{P} \equiv \texttt{P' and p'} \rightarrow \texttt{Q and Q} \equiv \texttt{Q'} \Rightarrow \texttt{P} \rightarrow \texttt{Q}
\end{array}
$$

Extended Concepts

2.5 Related Work

This section discusses much of the work discovered during the intial research of the project. There are many interesting live programming environments out in use in the world, with each having an interesting approach to the technical challenges surrounding the issue of temporal semantics. We briefly discuss some languages/environments that have made use of the Session Types structures as described previously as this gives a clearer picture of their use and viability within the real world. It is worth noting that, to date, there is no apparent system which seeks to apply the Session Typing structure to a live programming paradigm to address some of the concurrency issues faced in that field.

As noted by Rorhruber [9], there have been many publications and discussions relating to alternative approaches for temporal semantics and timing within Live Programming. There is much to be said about choosing between explicit and implicit representation of time as well as between the description of time using either internal or external state.

The Tidal language [21] uses an interesting formalisation of cyclic time. Whilst drawing a continuing analogy with the act of knitting, McLean describes the DSL for musical pattern embedded in the pure functional language Haskell. Tidal represents music as a pure function, enabling the mapping of the single dimension of time into multidimensional music, making full use of iterative language to formalise cyclic time using both analogue and digital pattern.

Impromptu is a much fuller system, able to produce both audio and visual outputs in the context of Live Programming [25]. It uses "temporal recursion" as a style of time-driven, discrete-event concurrency with real- time interrupt scheduling. This recursion acts as an extension to the already existing support for real-time execution of arbitrary code blocks, with the real-time scheduler being responsible for the execution of the blocks in the correct ordering. Impromptu is designed to provide a reactive system with timing accuracy and precision based on constraints of human perceptions. Its choice of asynchronous concurrency allows a flexible architecture wherein Impromptu's co-operative concurrency model leaves the programmer responsible for time keeping and meeting real-time deadlines.

One of the closest other languages to Sonic Pi V2.0, as mentioned earlier, is the similarly imperative styled language ChucK [27]. ChucK is a strongly -typed, imperative programming language whose syntax and semantics are governed by its flexible type system. The strength of the language lies in its (multiply) overloaded `=>` operator and its support of such features as dynamic control rates and strong concurrency principles. The ordering of a program is captured naturally by the logic of the operator and ChucK allows the programmer, composer and performer to write truly concurrent code using the framework of the timing semantic (as controlled by this overloaded operator and a few select timing keywords). The manner in which ChucK advances time allows a level of granularity that makes it a stronger system than Sonic Pi in terms of musical performance but it is much less useable as a first programming language.

The timing effects of ChucK and the inherent expectation within music remind us that we must be able to speak clearly about the location of events in time. Therefore, any musical programming language must prove some form of time semantics, even if only informally. As previously mentioned with Sonic Pi V1.0, in the context of Live Programming this consideration extends to both situations where the code runs too late and where the code runs too early.

An overlap between execution and creation time is a value of broader concern in software engineering, as noted by the Glitch system [22]. This system allows the user to adjust notional execution time relative to a point in the source code editing environment. It proposes the idea that programming languages should address state update order by abstracting away from the computer's existing model of time; i.e. they should manage time in a way that draws analogy with memory management.

Live Programming is not purely limited to musical languages and has similar applications is such areas as Logic, Dataflow and Artificial Intelligence.

As a demonstration for the potential of the application of Session Types within applications we breifly mention the protocol language Scribble [15], whose protocols are very clearly defined using this theory. This has come about through the recognised need for widescale structing of protocols in light of the increasing amount of communication-based networking. We also present Pabble [23]: a system based off of the Scribble design and implemented using multiparty session types; a further extension to the binary session types discussed earlier.

3 Technical Details

3.1 Ruby/C++

3.2 Timing Tree

3.3 Session Graph

4 Evaluation

5 Conclusion

5.1 Achievement Summary

5.2 Future Work

References

[1] *http://www.naec.org.uk/events*

[2] *http://sonic-pi.net/*

[3] *http://www.raspberrypi.org/about/*

[4] Aaron, S., and Blackwell, A.F., *From Sonic Pi to Overtone: Creative Musical Experiences with Domain-Specific and Functional Languages*, The First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design, Boston, Massachusetts, USA, ACM, pp. 35-46, (2013)

[5] Aaron, S., Orchard, D., and Blackwell, A.F., *Temporal Semanics for a Living Coding Language*, Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design, Sweden, ACM, pp. 37-47, (2014)

[6] Blackwell, A.F., Aaron, S., and Drury, R., *Exploring Creative Learning for the Internet of Things Era*, In B. du Boulay and J. Good (Eds) Proceedings of the Psychology of Programming Interest Group Annual Conference, pp. 147-158, (PPIG 2014)

[7] Berry, G., and Boudol, G., *The chemical abstract machine*, TCS, 96 pp.217248, (1992)

[8] Blackwell, A.F., and Collins, N., *The Programming Language as a Musical Instrument*, In Proceedings of the Psychology of Programming Interest Group Annual Conference, pp. 120-130, (PPIG 2005)

[9] Blackwell, A., McLean, A., Noble, J., and Rohrhuber, J., *Collaboration and Learning Through Live Coding*, Dagsthul Seminar, Dagstuhl Reports 3, no. 9, pp. 130-168, (2014)

[10] Church, L., Rothwell, N., Downie, M., deLahunta, S., and Blackwell, A.F., *Sketching by Programming in the Choreohraphic Language Agent*, In Proceedings of the Psychology of Programming Interest Group Annual Conference, pp. 163-174, (PPIG 2012)

[11] Coppo, M., Dezani-Ciancaglini, M., Padovani, L., and Yoshida, N., *A Gentle Introduction to Multiparty Asynchronous Session Types*, SFM 2015, LNCS 9104, pp. 146-178, (2015)

[12] Department for Education and Ofsted, *ICT in schools: 2008 to 2011*, Piccadilly Gate, Manchester, 110134, (2013)

[13] Department for Education, *National curriculum in England: computing programmes of study (key stages 1 - 4)*, (2013)

[14] Hansson, H., and Jonsson, B., *A Logic for Reasoning About Time and Reliability*, Formal Aspects of Computing 9, no 5., pp. 512-535, (1994)

[15] Honda, K., Mukhamedov, A., Brown, G., Chen, T., and Yoshida, N., *Scribbling Interactions with a Formal Foundation*, In 7th International Conference on Distributed Computing and Internet Technology, p. 55-75, (ICDCIT 2011)

[16] Honda, K., Vasconcelos, V.T., and Kubo, M., *Language Primitives and Type Disciplines for Structured Communication-Based Programming*, ESOP'98, LNCS 1381, pp. 22-38, (1998)

[17] Honda, K., Yoshida, N., and Carbone, M., *Multiparty Asynchronous Session Types*, POPL'08, San Francisco, Calfornia, USA, pp. 273-284, (2008)

[18] The IEEE and The Open Group, *Sleep - The Open Group Base Specifications Issue 7, 2013*, http://pubs.opengroup.org/onlinepubs/9699919799/functions/sleep.html, Retrieved 15 May, (2014)

[19] Milner, R., *Functions as processes*, MSCS, 2(2) pp.119141, (1992)

[20] McDirmid, S., *Living it Up with a Live Programming Language*, Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, New York, USA, ACM, pp. 623-638 (2007)

[21] McLean, A., *The Textual X*, Proceedings of xCoAx2013: Computation Communication Aesthetics and X, pp. 81-88, (2013)

[22] McDirmid, S., and Edwards, J., *Programming with Managed Time*, Tech. Report, Microsoft, (2014)

[23] Ng, N., and Yoshida, N., *Pabble: Parameterised Scribble for Parallel Programming*, In 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing, p. 707 - 714, (PDP 2014)

[24] Lee, I., Davidson, S., and Wolfe, V., *Motivating Time as a First Class Entity*, Technical

Reports (CIS), pp. 288, (1987)

[25] Sorensen, A., and Gardner, H., *Programming with Time: Cyber-Physical Programming with Impromptu*, ACM Sigplan Notcies 45, no. 10, 822-834, (2010)

[26] Woolford, K., Blackwell, A.F., Norman, S.J., and Chevalier, C., *Crafting a Critical Technical Practice*, Leonardo 43(2), 202-203, (2010)

[27] Wang, G., and Cook, P.R., *ChucK: A Concurrent, On-The-Fly Audo Programming Language*, International Computer Music Conference, pp. 1-8, (2003)

[28] Wing, J.M., *Computational Thinking*, Communication of the ACM, Vol. 49, pp. 33-35, (2006)

[29] Yonezawa, A., and Tokoro, M., *Object-Oriented Concurrent Programming*, MIT Press, (1987)