Musical Concurrent Programming With Sonic Pi

Interim Report

Eleanor Vincent

January 26, 2015

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. 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 "turle graphics". Some time after this came the Computers in the Cirriculum Project, funded from 1972 to 1991 by the Schools Council and subsequently by the Microelectronics Education Programme 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 in the vein of giving young people the ability to work with computers during their formative years but over the course of the late 80's and early 90's this focus on programming ability gives way to simply the education of practical use of existing computer software instead of a focus on the 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 [9] 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.

In 2012, the Government began to recog the signifiance of Computer Science and has replaced the National ICT curriculum with a revised Computing cirruculum. The new Computing Program of Study [10] 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.

Along a similar vein 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 until the later years of education.

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 [8, 7], and investigations into the craft practices of existing professional software developers who work in professional art contexts [16].

The movement is not purely restriced 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 [18].

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]. There are many languages in existence which are simply enough to also constitute as a good first-programming language, but many do not attempt to make themselves an inviting gateway into the realm of technical programming. 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

This project's aim is to formalise the concurrency and timing aspects of the language and develop a program analysis tool that can be used to identify program bugs such as deadlock and thrashing behaviour. The formalism and analysis will build on initial work describing timing and concurrent interaction via effect systems and session types. The project combines theoretical aspects of type system and analysis design, as well as practical work developing a responsive program analysis engine.

1.3 Report Structure

The remainder of this report is broken down as follows:

-Background: We detail the histories of Sonic Pi, Live Programming and Session Types as fields

of research and conclude with work relating to these subjects.

-Going Forward: The bulk of this section details the expected timeline of the project and ideas for how to evaluate progress during and at the conclusion of the project.

2 Background

In this section we begin by explaining the ideas and features of Sonic Pi, the living coding program 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 the related work in these areas of research.

2.1 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 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 Cirruculum, 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 breif description of the Raspberry Pi then explores the implementation of Sonic Pi V1.0, mainly to give appropriate context to the timing effects system that Sonic Pi V2.0 currently imeplments. There is then be some short discussion on the particular features of Sonic Pi V2.0.

2.1.1 Raspberry Pi

The Raspberry Pi was developed as a very low cost computer system to enable technical experimentation amongst young people who had little other contact with computer systems. 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 come 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 Amigos, 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 envi-

ronment 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.

2.1.2 Sonic Pi V1.0

The initial development time given to Sonic Pi v1.0 was roughly three weeks. It was designed largely as a port from the language Overture to focus on driving specific educational objectives with the idea in mind to teach a target audience of 12-year-olds that had no previous experience with programming; it would bring them from the introduction of a computer through to the ability to write a full length program over the course of a few weeks. Sonic Pi was built as a ruby-based language both through the author's existing experience with the language and also to keep it in line with languages already used within the industry. Python is well regarded as an educational language and given the semantic similarity between Python and Ruby it was easy to defend Ruby as a choice of language implementation [4].

The immediate feature that Sonic Pi focuses on is sequential ordering in imperative programs; demonstrated in musical theory by the sequential playing of notes.

The Code Snippets shown demonstrate two small Sonic Pi programs. The first demonstates 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 be likely to 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 the second code Snippet. The notation for sleep in Sonic Pi V1.0 is similar to that of the POSIX sleep operation[12]. The default sound that Sonic Pi uses in these contexts is a pleasant bell sound from the SuperCollider synthesiers.

```
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 monadic structure of the statements may not be necessarily elegant to many experienced programmers. However, 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 as 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 are presented here in the context of V1.0 programs as the two versions are semantically 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.

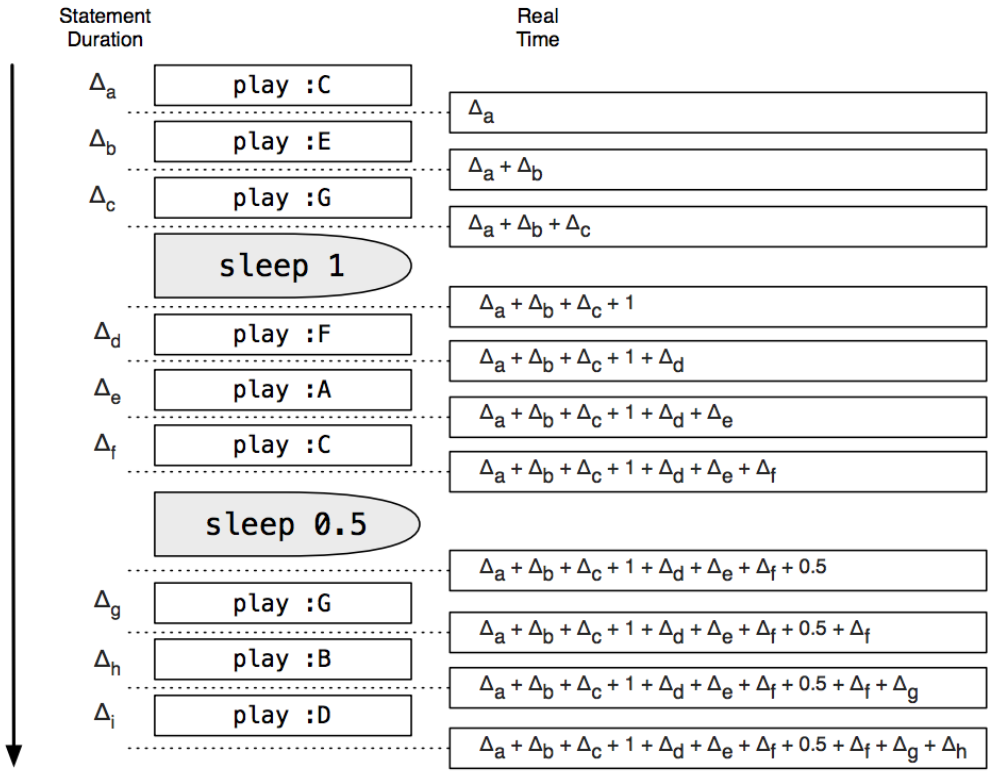

Figure 1: Timing in Sonic Pi V1.0

2.1.3 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 clocking 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 [17].

For completeness it was 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 a previously mentioned 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 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 are from the play statements following the last `sleep` executed. This is demonstrated visiually 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 jitter time and 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 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 [11]. 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

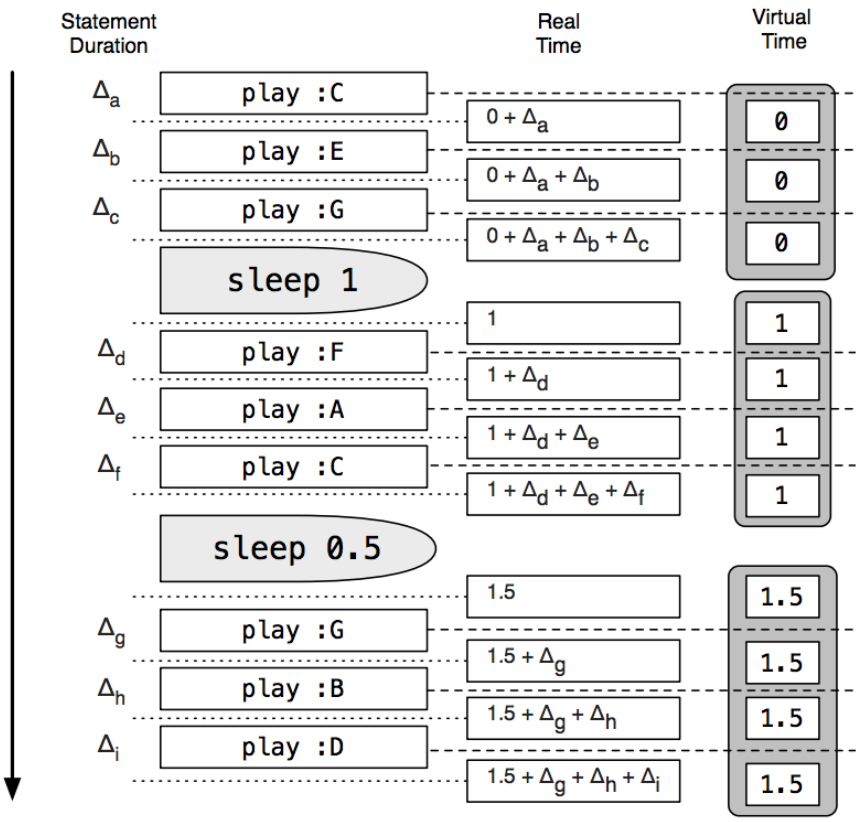

Figure 2: Timing in Sonic Pi V2.0

---

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 textttsleep commands which would have the problem of taking up all of the system resources; instead the thread self-termininates 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 programming is running. Threads also implement thread- based inheritance wherein they take all of the synthiser settings of the thread they were spawned from.

The keyword `in_thread` is 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 a `live_loop` functionality.

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 primitivs 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 it's 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 the loops `foo` and `bar` are out of time with one another.

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 scenario's 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

2.2 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 [15]. 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 language which allow for the flexibility, conciseness and ease of development [13] 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 will often 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. 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 [14].

2.3 Session Types

2.4 Related Work

3 Going Forward

This section illustrates the plan of action laid out for the rest of the project over the next few months. The first section illustrates the rough idea of how long each implementation of the program features will take, with documentation and evaluation time factored in. After that is the outline of how the project will be evaluated.

3.1 Project Outline

3.2 Evaluation Outline

4 Final Thoughts

References

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

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

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

[4] Aaron, S., 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., Blackwell, A.F., *Temporal Semanics for a Living Coding Language*, Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design, Boston, Massachusetts, USA, 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] 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)

[8] 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)

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

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

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

[12] 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

[13] 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

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

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

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

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

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