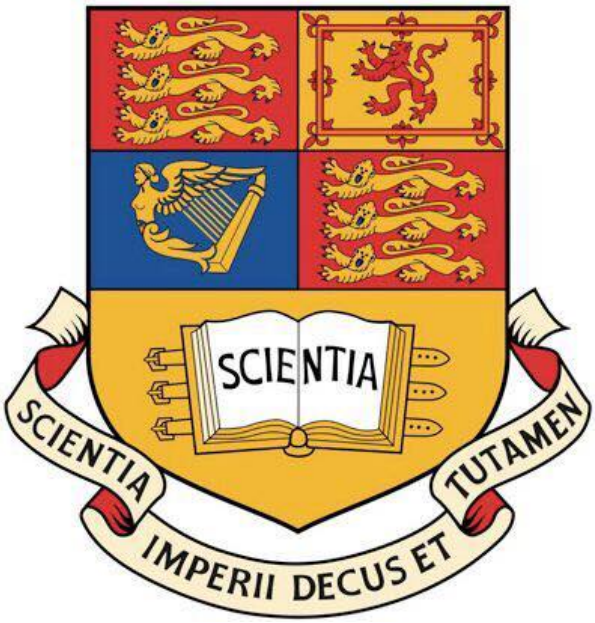
Department of Computing
Imperial College London

Individual MEng Project

Verifying Musical Concurrent Programming for Sonic Pi

Eleanor Vincent
Supervisor: Nobuko Yoshida

June 16, 2015



Abstract

The report details the improvements made to Sonic Pi: a musical live coding language. With this style of live, interactive improvisation, time becomes a crucial factor in giving an enjoyable, effective performance. Sonic Pi combines this with multiple concurrency primitives that affect the interactions between the different running threads. The current problem with Sonic Pi is the lack of ability to verify the program before running it; during a live performance this can be disastrous. The challenges this present involve the difficulty in handling concurrent code and how to define the concept of time in the context of musical coding. When musical pieces become very long and complex the length of time the piece runs for, and which music threads interact with which, become very difficult to reason about.

In this report we develop a lightweight static analysis and typing tool for Sonic Pi programs. We develop a formal specification of the temporal behaviour of Sonic Pi, and describe our implementation of this system. Alongside this we apply the field of session types to the underlying communication structure of Sonic Pi, formalise its ideas, and present the implementation of this typing system. By running the systematic testing suite we have designed, we prove the correctness of our analysis of the program against these two formalisations and also comment on the overall performance of the additions. We evaluate the results of our analysis in the context of an educational language. Many of the features implemented lend themselves to personal exploration while others are well suited to the goal of educating young people in the realms of concurrency and temporal reasoning.

Acknowledgements

Firstly, I wish to thank my supervisor, Nobuko Yoshida, both for proposing this project, and for being a constant source of motivation, inspiration and support; even through her recovery earlier this year. Secondly, I extend a warm thanks to Dominic Orchard, whose help and feedback throughout this whole project has been invaluable.

I would like to give a special mention to Sam Aaron as thanks for building and open-sourcing Sonic Pi. It has been an absolute joy to work with. My thanks also go out to my family, for their unwavering support of my degree, my housemates, for keeping me healthy, and my other friends, for their company.

Contents

1 Introduction **9**
- 1.1 Motivation . . . 9
- 1.2 Objectives . . . 10
- 1.3 Contributions . . . 10
- 1.4 Report Structure . . . 11

2 Background **12**
- 2.1 Live Programming . . . 12
- 2.2 Sonic Pi . . . 13
  - 2.2.1 Computing in Schools . . . 13
  - 2.2.2 Raspberry Pi . . . 14
  - 2.2.3 Sonic Pi V1.0 . . . 14
  - 2.2.4 Sonic Pi V2.0 . . . 16
- 2.3 Session Types . . . 22
  - 2.3.1 Multi-Party Session Types . . . 23

3 Related Work **24**

4 Design & Technical Implementation **26**
- 4.1 Pi Time . . . 26
  - 4.1.1 Building the AST . . . 27
  - 4.1.2 Formalising Abstract Time . . . 30
  - 4.1.3 `pTrace` . . . 32
- 4.2 Session Pi . . . 37
  - 4.2.1 Formal Session Types . . . 37
  - 4.2.2 Planning the Graph . . . 42
  - 4.2.3 Global Inference . . . 47
- 4.3 Integration . . . 50

5 Evaluation **51**
- 5.1 Ruby/C++11 & Performance . . . 51
- 5.2 Correctness . . . 52
  - 5.2.1 Chords . . . 52
  - 5.2.2 Sequences . . . 53
  - 5.2.3 Loops . . . 53
  - 5.2.4 Threads . . . 55
  - 5.2.5 Data Structures . . . 56
  - 5.2.6 Dead Code . . . 58
  - 5.2.7 Function Calls . . . 59
  - 5.2.8 Conditionals . . . 63
  - 5.2.9 ; Separation . . . 64
  - 5.2.10 Communications . . . 65
  - 5.2.11 Musical Score . . . 68
- 5.3 Visual Adaptation . . . 71

6 Conclusion **73**
6.1 Future Work 73
6.2 Achievement Summary 74

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, after the last Cirriculum Project had been shut down in 1991, and has replaced the National ICT curriculum with a revised Computing cirruculum. The new Computing Program of Study [16] 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 similar 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 [36]. A recent Microsoft article[1] claims that 75% of students in the Asia Pacific region wish that programming could be offered as a core subject in their schools [4].

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

There is an increasing recognition of the power of programming amongst other disciplines. A growing hacker and maker movement has been making programming a much more accessible skill [9]; it presents itself as a viable and useful hobby amongst a vast range of ages and professions, largely due to the increasing availability of resources. Existing research has examined the viability of programming tools for professional artists, as reported at PPIG [13, 11], and the craft practices of existing professional software developers who work in professional art contexts [34]. A powerful example of this is the popular uptake of Minecraft in both educational and personal areas. Many uses focus on playing Minecraft on maps designed to teach various scientific concepts [30, 31] but there is the ability to learn programming through mod development itself. Official statistics are difficult to locate but a popular game download website listed 2,000+ Minecraft mods, and 14,000+ texture packs for download; the most popular mod has almost 3 million downloads [5], proving its popular reach.

In parallel with this we have entered an age where concurrency and distribution have become some of the most pressing issues in computing. As we begin to process hugely increased amounts of data the ideas of parallelism and exploiting concurrency are becoming more and more pronounced. In this day and age it is almost impossible to find a person who does not own a mobile, most likely a 'smart' device, which relies on communication with other such devices to function effectively.

Concurrency has been an active field of research since the 1960s, producing many formalisms over that time which have improved the field of computing. One such formalism is that of process calculi, a field that π-calculus is a strong member of. Another type of process calculi,

[1] Part of the #WeSpeakCode campaign.

CSP, was highly influential in bringing to light such languages as Go: a relatively young language but praised for its concurrency primitives and performance. Go can be found in the backend of many notable technology companies. Models of concurrency are being used for reasoning and specification as well as at all stages of the development life cycle: design, implementation, testing, etc...

Whilst there has been much in the field to support a programmer as they produce type-safe code, the same cannot be said for communication-safe code. Concurrency bugs such as deadlocks and thrashing behaviour can be as hard to locate now as ever before. Some tools and techniques exist to handle these situations, such as mutexes and locks [33], but even then there is no guarantee that it will work.

1.2 Objectives

In the current version of Sonic Pi [2] there is no means to verify the temporal behaviour or the communication patterns of the program. This is to say, users must judge themselves how long the current program is, that the program is timed correctly, and that the program is free from concurrency bugs. This can be difficult enough in a static environment; in Sonic Pi there is the additional requirement to do all of these things during a live performance (studio audience included)! The first point can be difficult for novice programmers, potentially moreso for those with little musical experience. The latter can be difficult for even some experienced programmers due to the non-deterministic nature of some concurrency bugs.

We aim to develop a lightweight library to analyse the temporal behaviour of a Sonic Pi program. There are several 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; it is important that the project is still able to run on Sonic Pi's target architecture. Temporal behaviour should be able to handle basic sequencing, funtion calls, nesting and variable time operations.

We aim to build an overview of the communication patterns within a Sonic Pi program by utilising the theor of Session Types. Session Types will enable us to define a clearly understandable structure to the communication protocol of the program; this will enable the tool to identify difficult bugs such as deadlocks and thrashing behaviour. In identify problems in this manner, we will be able to reduce the number of crashes developers experience whilst writing code 'on-the-fly'.

The final requirement will be for the analysis to run within the Sonic Pi IDE, and to run quickly enough to be utilised during a real perforamnce environment. The challenge here is to make the analysis sufficiently lightweight enough not to slow the IDE while the synth servers are working.

1.3 Contributions

We have produced a lightweight shared library, often affectionately referred to as Sonic-Verify, that enables a variety of different analysis of a program written for the Sonic Pi IDE. The project analyss the timing effects of the program, outputting such information as the total

'virtual time' elapsed by the program and more specific detail, such as function durations. This implementation of a formal timing system presents an improved environment in which developers may code more asthetically pleasing music in a live environment.

This is the first instance of the theory of Session Types being applied in a (musical) live programming context. With this analysis we are able to output the decomposed types of all processes in the program and the associated global type, presenting an intuitive description of the interactions of processes within the source code. This presents an exciting new realm of possibility in terms of concurrency verification and the application of Session Types in existing distributed paradigms. We also present two new extentions to Multi-Party Session Types based on the conceps of replication types and broadcasting, as they appear in Sonic Pi. We also present a method of *Global Inference*, an area of Multi-Party Session Types that is still largely undeveloped.

We have implemented a set of tests that systematically verifies the results that our project produces; it covers a range of features starting from the most basic of sequential programs through into full music sources, with complex function call structures and multiple interacting process threads. Finally, we have fitted this library into the existing Sonic Pi IDE in a manner that is beneficial to both new coders and 'old hats' alike.

1.4 Report Structure

The remainder of this report is broken down as follows:

- Background: We detail the basic concepts of Sonic Pi and its timing effects, and Session Types to give an adequate foundation for the work presented in this project. We also explore the field of Live Programming as part of the background research.

- Related Work: We detail the existing work in the field Live Programming with a particular focus on those with musical contexts. We also detail some of the existing applications of Session Types, outlining the approaches and differences therein.

- Design & Technical Implementation: Here the chapter is broken into two core sections, focusing on both the timing effects of Sonic Pi and the session types of Sonic Pi in turn. We discuss chosen languages and key libraries used in the design of the program and then move into general implementation and interesting algorithms and code from each section. The chapter concludes with a brief discussion of the architecure of the Sonic Pi IDE and how we chose to integrate our verification library.

- Evaluation: Here we evaluate the success of the project. We discuss the limitations of the current state of the project and outline qualitive and quantative metrics for the project, including a systematic test of the correctness of our approach.

- Conclusion: Here we present possible improvement for the project in the future work section before closing with a summary of those objectives achieved.

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 Live Programming and Session Types in further detail and seek to relate them back to the current aims of the project.

2.1 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 [32]. 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 [23] 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 [24].

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.

2.2 Sonic Pi

Sonic Pi is an imperative live programming language designed as an educational first language. It is a Domain-Specific Language based on Ruby designed for manipulation of synthesisers through time [7]. 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 Cirruculum, are conditionals, iteration, variables, functions, algorithms and data structures. Sonic Pi also extends beyond these concepts to include such features as multi-threading and hot-swapping of code as these are likely to be of crucial importance in the future of programming contexts [8]. It also presents an inviting environment to teach concepts such as parallelism and concurrency; a core strength of the program as these can be notoriously difficult concepts to explain.

2.2.1 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 1980s 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 [15] 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 teachers as much as the cirriculums failure to address the issues.

2.2.2 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 Amiga, 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[2], 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.

2.2.3 Sonic Pi V1.0

Sonic Pi V1.0 was designed to port the language Overtone 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

[2] About £16.

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 command [21]. 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 syntactically 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 therefore non-deterministic in nature and also not consistent across different runs of the same program.

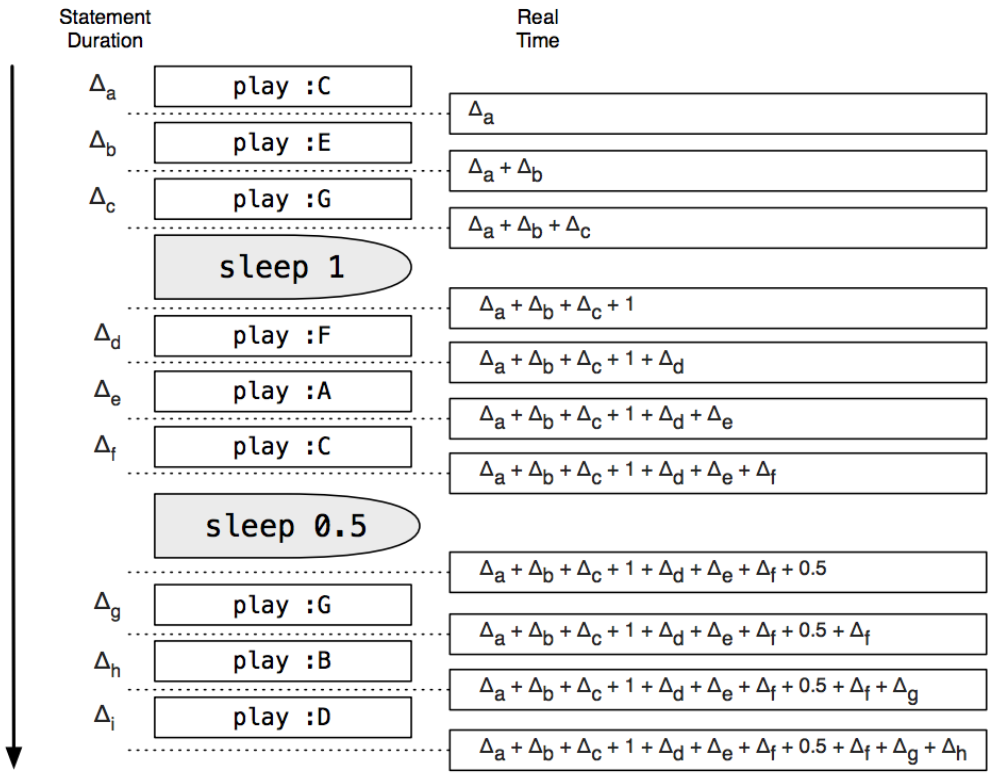


Figure 1: Timing in Sonic Pi V1.0 [8]

2.2.4 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*. V2.0 maintains syntactic compatability with V1.0, so it as conceptually 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

Each chord line demonstrates Sonic Pi's ability to play notes in parallel, the system still taking advantage of the capabilities of modern processors[3]. `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.

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 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 [17]). In the event a thread falls

[3]Snippet 5 demonstrates the ordered effect, as the three chords are played in the order written, and also Sonic Pi's abiltity to run with or without semi-colons.

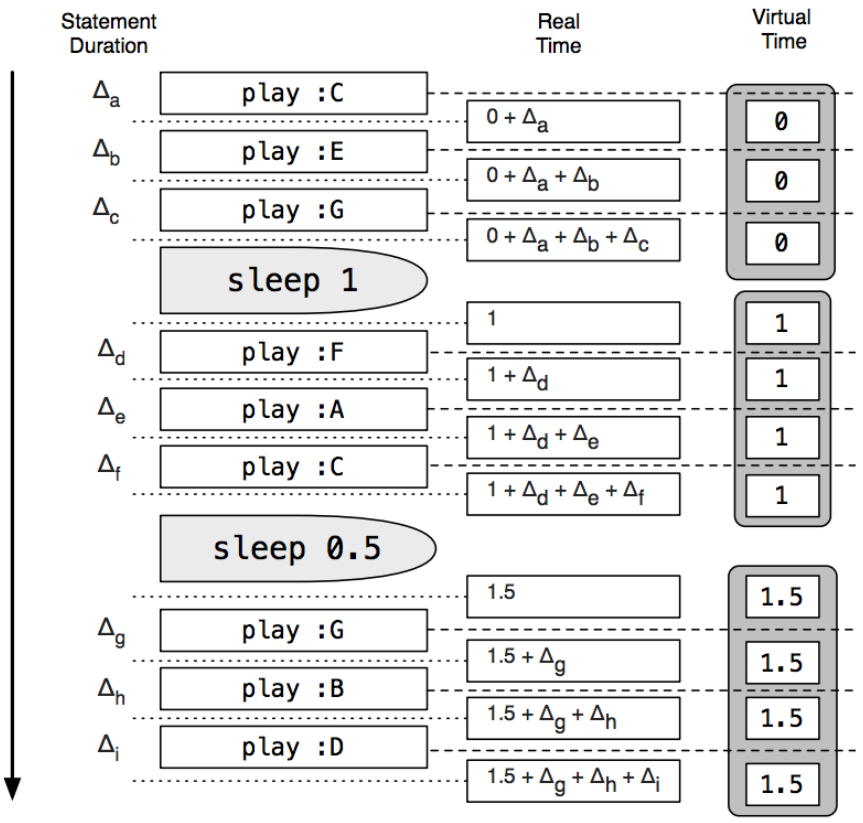


Figure 2: Timing in Sonic Pi V2.0 [8]

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 briefly; 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; a developer must combine each thread with a loop in order to produce constant sound. Sonic Pi V2.0 provides the `live_loop` keyword to capture the same effect with less writing.

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

Snippet 7: Live Loops

While 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 reset the program with every small edit, as is the case with standard programming. This feature, however, is directly affected by the previously described temporal semantics; loops can easily become out of time with each other during a performance. 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. `cue` is an asynchronous, non-blocking operation and `sync` is a blocking operation.

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.

We can start 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 (since a `live_loop` will send a `cue` message at the start of each loop). In this case we have synced `:bar` onto `:foo`'s `cue` message.

This gives way to some very obvious deadlock scenarios (example shown in Snippet 10) that users must avoid. Here we have created two `live_loops`, `:foo` and `:bar`. Both threads finish with a blocking call, waiting for a `cue` message from the other loop. The issue here is, since both loops are blocked at the same time, neither loop will ever trigger another `cue`.

```
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
    play :e4, release: 0.5
    sleep 0.5
    sync :bar
end

live_loop :bar do
    sample :bd_haus
    sleep 1
    sync :foo
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 five key components:

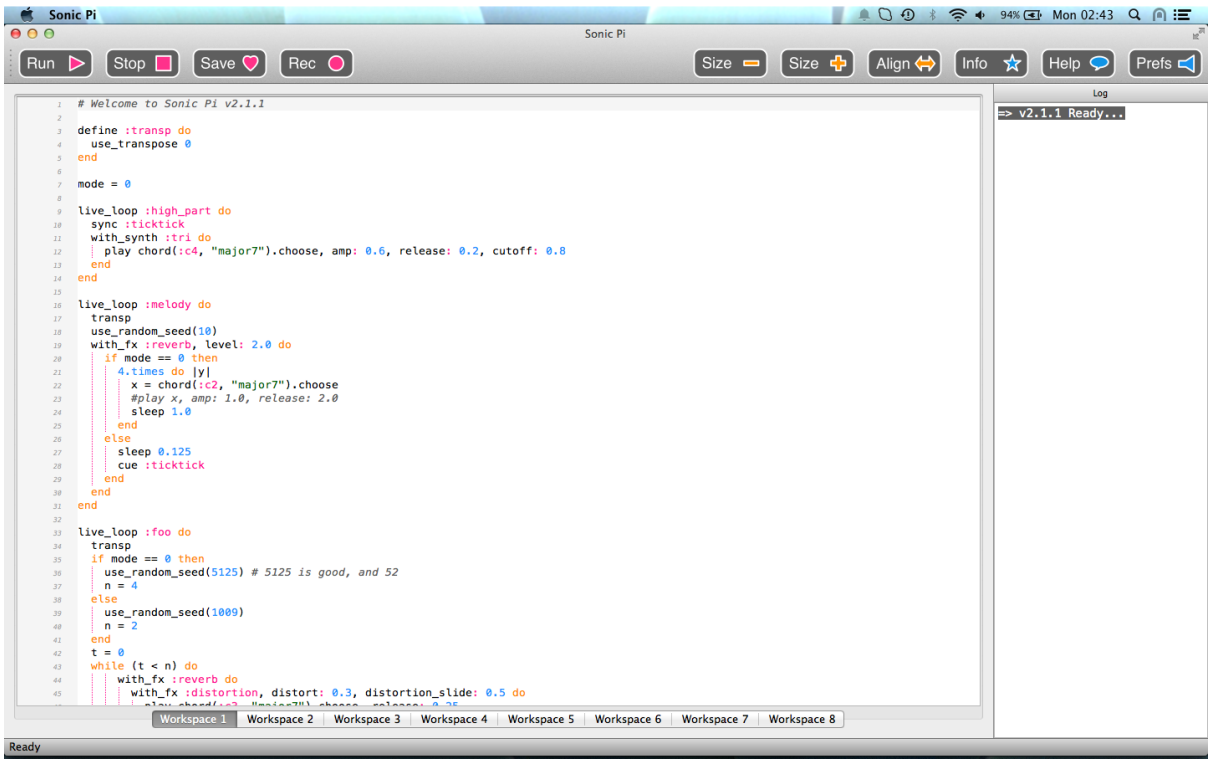


Figure 3: Sonic Pi V2.1.1 IDE - Apple Mac View

- Control Buttons (Musical/Interface)
- 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 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 [19].

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, readible and ultimately more easy to verify. Honda et al [19] 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 interaction patterns that is one of the main reasons for bugs in commmunication-based programming[4].

[4]We choose not to present the full formalisation of the type system in this report as the focus is much more on the construction of the communication protocols than on their underlying types. Interested readers can read on this futher in such papers as [19] and [20].

2.3.1 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 asynchrounous communication[14]. 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 [20].

3 Related Work

This section discusses much of the work discovered during the intial research of the project. There are many interesting live programming environments 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 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 Type protocol to a live programming paradigm and their associated concurrency features.

As noted by Rorhruber [12], 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 language Tidal [24] 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 [32]. 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 is the language ChucK [35]. 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 similarity between this operator and the use of `sleep` to advance time make this language a very interesting source of comparison. 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 [25]. 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 in terms of temporal reasoning. We do not list any such examples here as this is not within the scope of this project[5].

As a demonstration for the potential of the application of Session Types within applications we briefly mention the protocol language Scribble [18], 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. Also of interest is Pabble [28]: a system based off of the Scribble design and implemented using multiparty session types; a further extension to the binary session types discussed earlier.

[5]Interested readers are encouraged to read [8] as they do detail some related lanuages in their closing sections.

4 Design & Technical Implementation

This chapter of the report focuses on the high level design of the project. The sections are split into two core parts, one focusing on the design and implementation of the temporal behaviour, and the second focused on the session types and associated graph structure.

The temporal behaviour of Sonic Pi is formalised through an effects system, also referred to as an abstract representation of time. We occassionally refer to this as the timing effects system of Sonic Pi, meaning the features implemented to handle reasoning about timing. We first describe this formal abstraction and then relate it to the approach of our implementation, which we present alongside the class structure of the project.

Similarly in the second section we present the formal definitions of session types and then relate the theory to our implementation. The implementation is again presented alongside the class structure of this section.

We finish the chapter with a short focus on how the library is finally integrated into the Sonic Pi IDE[6].

4.1 Pi Time

Before being able to process the source to analyse the virtual time, we must parse the source into a suitable data structure. A natural approach to this mimics compiler technology: we define some grammar by which to parse our programs, and then build some Abstract Syntax Tree to store all of the program information. Once this is built successfully we are able to traverse the tree as often as we require to build the trace of the program, described in more detail in Section 4.1.3.

Whilst ruby has been valued in Sonic Pi for its hugely forgiving syntax in a classroom setting, this feature increases the difficulty of parsing it. The ability to write function calls either with or without parentheses or omit semi-colons between statements increases the complexity of the ruby grammar to the degree it was infeasible to write such a parser specifically for Sonic-Pi. However, given Sonic-Pi's ruby-like nature, we elected to use the open-source ruby-parser[7] for its simplicity and ease of set-up. It is relatively lightweight whilst still providing useful statistics such as line numbers, statement counts and column numbers.

[6] As found at https://github.com/samaaron/sonic-pi
[7] As found at https://github.com/whitequark/parser

4.1.1 Building the AST

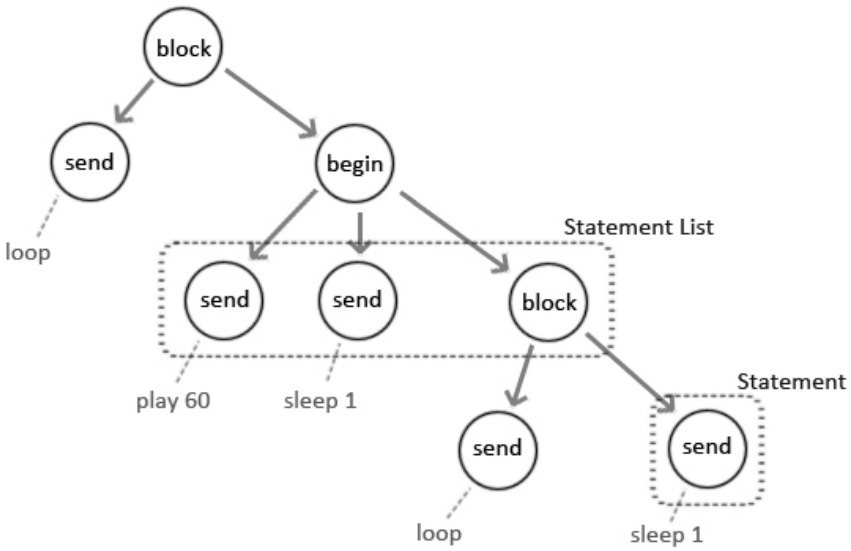


Figure 4: A Simple Program AST

The ruby-parser builds the AST with a rose like structure which we continue to utilise throughout our project as it provides many useful node types. The most interesting nodes are `block` and `begin`; `block` for its use in analysis and `begin` for how it manages statements.

```
loop do
    play 60
    sleep 1
    loop do
        sleep 1
    end
end
```

Snippet 11: Simple Loop

Figure 4 captures the AST of Snippet 11, shown above. Loops and function definitions are denoted by `block` types. This `block` can act as a root node, enabling us to work with each loop and function as separate trees during analysis, compounding the results at the end. The full structure of a `block` is shown in Figure 5. A `block` will always have three children: the first names the type of block, the second contains the arguments given, and the third is the statement list. This statement list is either marked with `begin` or contains a single statement node. Figure 4 demonstrates both of these examples. The outermost `loop` contains three statements, so the AST marks the start of that statement list with `begin` which holds all the following statements as children. The innermost `loop` holds only a single statement, so `begin` is omitted.

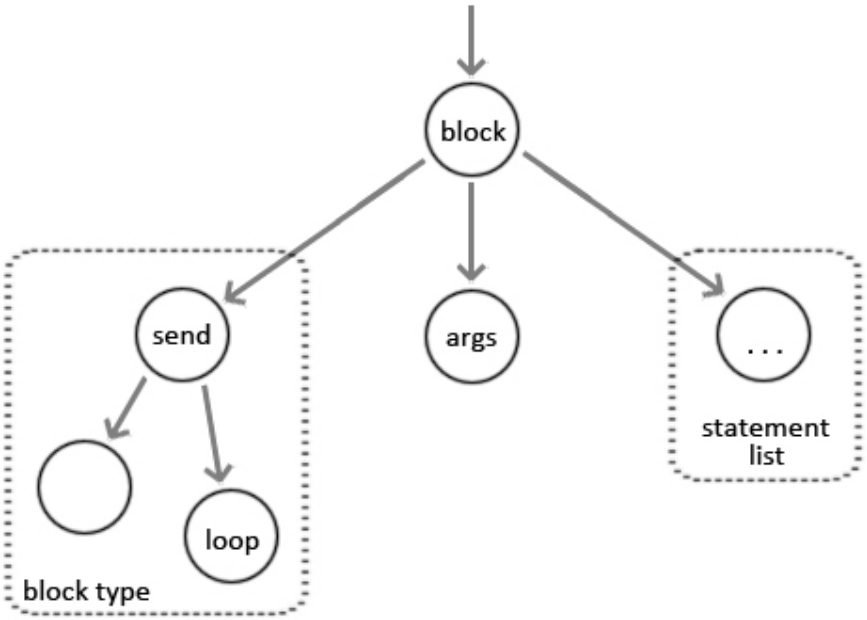
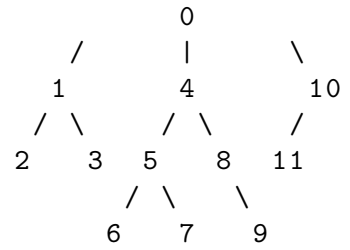


Figure 5: Block Node Structure

This rose-like structure means that our tree is searched in a breadth-first manner. Take the following tree diagram, which demonstrates how node numbering is managed. When searching for index 8, breadth-first search means that we will search each child, 1 then 4 then 10, before visiting any of their subsequent children. By testing that the index is greater than the current index but less than the next child, we reach our target quickly.



This child structure makes it simple to detect sequential and nested `blocks`. As shown in Figure 6, sequential `blocks` will always share a common parent (`begin`). Otherwise, as with the nesting in Figure 4, encountering a new `block` with a differing parent suggests nesting. This is leveraged later when we detail the timing analysis in 4.1.3.

In transferring this data to our own C++ application we simplify the node count of our AST by wrapping certain nodes into tighter definitions. For example, the ruby-parser outputs an integer as a node of type `int` with a child holding the actual integer value. We fold these definitions

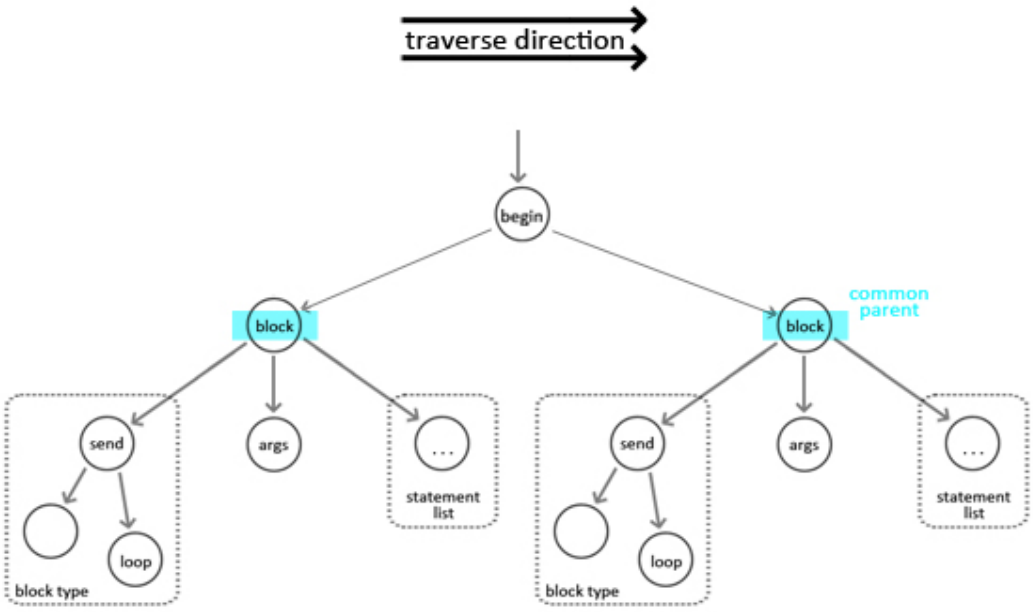


Figure 6: Block Detection

into a single node of type `IntNode` with held the integer value as a member field. `float` and `symbol` nodes act in a similar fashion, as demonstrated with 7.

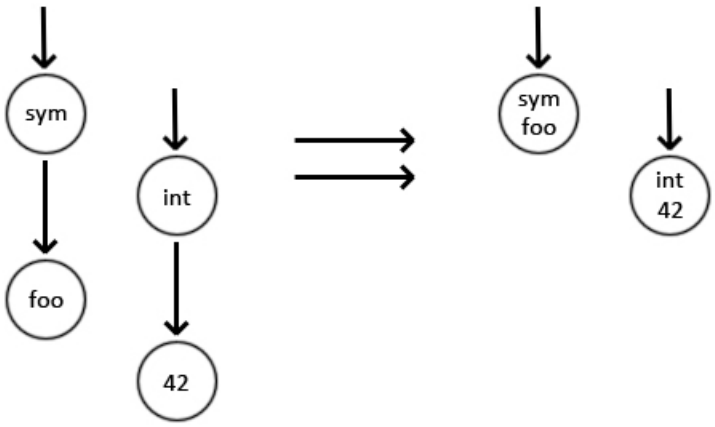


Figure 7: Node Folding

4.1.2 Formalising Abstract Time

Before writing the temporal behaviour, we provide an abstract definitions of time for Sonic Pi to work with. This representation can be thought of as a 'time system' and enables us to define a concept of 'time safety'. By proving the semantics sound, with respect to this system [8], we are able to reject those programs with bad temporal behaviour. This static verification enables developers to reason about their program's timing intuitively.

The syntax of Sonic Pi V2.0 can be outlined simply.

```
P ::= P; S | ∅
S ::= E | v = E | def f : P
E ::= sleep ℝ≥0 | A^i | v
    | if v then P else P | f
```

`P` represents a full program, `S` are statements (either self-contained expressions or pure bindings to variables, v), and `E` defines expressions. $\texttt{A}^i$ is used to refer to all those operations that will not advance time, e.g. the `play` operation. In defining the syntax this way we are able to abstract over the operations that do not modify virtual time.

We extend the definition given in [8] with f denoting a string used for either function definitions or function calls, and the rule for conditionals. `def` f `:` `P` means to say a function with name f is defined with program `P`; f is a function call to that definition.

Definition 1:
$[\text{-}]_v$ is an overloaded operator mapping Sonic Pi terms to $\mathbb{N} \cup \infty$. Using this we define (-)* as n* $= \infty$. Virtual time is specified for Sonic Pi using the following cases:

```
[P; v = E]_v = [P]_v + [E]_v     # Statement Sequences
       [∅]_v = 0                 # Empty Program
 [sleep t]_v = t                 # Sleep Statement
       [v]_v = 0                 # Variable
      [A^i]_v = 0                # Other Statements
```

We extend the cases here (as presented from [8]) with functions, loops and conditionals:

```
[def f : P]_v = 0                      # Function Definition
       [f]_v = [P]_v if ∃ def f: P     # Function Call

  [loop P]_v = [P]*_v                  # Loops

                                       # Conditional
[if v then P1 else P2]_v = [v]_v + ([P1] max [P2])
                         = [P1] max [P2]
```

The * operation marks a virtual time of infinite length. It is defined as an absorbing operation.

$$\infty \texttt{ + n = } \infty$$
$$\texttt{n + } \infty \texttt{ = } \infty$$

With this absorpting behaviour we may also state that:
$\texttt{[loop P; P']}_v = \texttt{[loop P]}_v$.

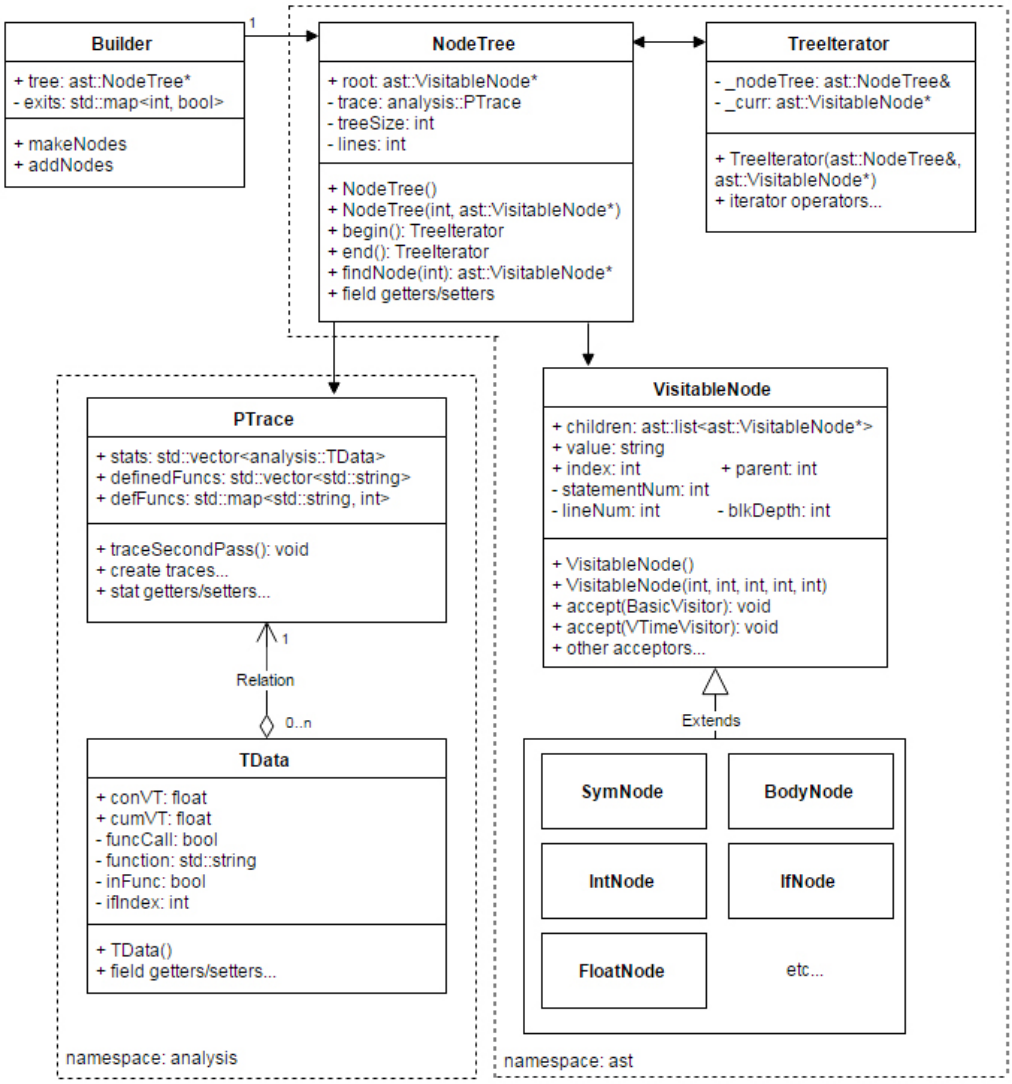


Figure 8: Tree UML

4.1.3 pTrace

In building this pTrace we are mapping our existing tree structure to a flat list structure. We could have simply stored the timing information into the nodes themselves but direct index access to the informations proves preferential when integrating data into the IDE. As shown in Figure 8, each NodeTree holds a reference to a pTrace, the data structure we use to store all of the important analysis results for the current program. The main structure of interest within this pTrace is the vector of statements allowing us to utilise the []-operator when examining specific statements.

Each index contains a struct of useful data such as the node index of the statement, contributive virtual time (referred to here as conVT and indicitive of a sleep statement or function call), cumulative virtual time (referred to as conVT and used to know what the total virtual time is at each statement), and other fields relating to function calls. When the statement is a sleep t statement, it marks that the contributing virtual time, conVT, is of length t. Most other statements, aside from function calls, are given a conVT of zero. Function calls contribute the amount of virtual time they take, the values of which are stored in a map within pTrace.

The analysis processes the tree into pTrace using the specification presented in Section 4.1.2. This happens in two 'passes' by way of interprocedural analysis. The 'first pass' refers to a full traversal of the AST, building a pTrace of the correct length and marking each sleep statement, function call and conditional where it occurs. The 'second pass' refers to an iteration over the pTrace where we use the data collected to fill in the results of function calls and process conditional brances.

Loops

To start with a simple example, Snippet 12 shows the code for the following simple trace.

```
loop do     # 1
  play 60   # 2
  sleep 2   # 3
  play 64   # 4
end
```

Snippet 12: Simple Loop

```
== Trace
[0] -
[1] conVT: 0, cumVT: 0,
    isCall: false, inFunc: false
[2] conVT: 0, cumVT: 0,
    isCall: false, inFunc: false
[3] conVT: 2, cumVT: 2,
    isCall: false, inFunc: false
[4] conVt: 0, cumVT: 2,
    isCall: false, inFunc: false
```

The information for each statement is mapped to the index of the same statement count. Index 0 is never filled as this is always assigned to the root node, and because it is natural for novice programmers to index from 1 rather than 0. Given this, we often skip handling the first index of our trace because it is always a zero slot [8]. Snippet 12 demonstrates quite neatly how simple

[8] This is based on the fact a program is normally a sequence of statements or a loop. A check has been put in place should certain constructs be processed in isolation. This is addressed specifically in Section 5.2.8.

it can be to collect the cumulative virtual time as we iterate through the program.

In Section 4.1.2 we define a loop's virtual time as being infinite. This is, however, not a useful result to return to the developer. Instead, each unconstrained loop is traversed once so that the value of one iteration is known. All the information returned to the developer is based on running one iteration of the whole source as this is a useful metric for the developer to reason against. This is valid within our specification as we have defined * as mapping natural numbers to infinity. $[\texttt{loop P}]_v$ in this case equals 2^* which equals ∞.

Functions

Snippet 13 gives a program where functions have been defined before and after their use. This is a valid ruby program and will compile and run within the Sonic Pi IDE. However, Sonic Pi's current scoping system mean that this program will fail during runtime[9]. This kind of error is outside the scope of the project to provide an error report for so we do not save the developer in this instance. Instead we handle it, as it compiles within the environment and is sent for evaluation by the underlying servers. Thankfully, our interprocedural analysis allows us to handle this case for free, so we use it here as a strong example of function calls and their temporal behaviour.

```
define :first do          # 1
  play 60                 # 2
end

first                     # 3
second                    # 4
play 60                   # 5

define :second do         # 6
  sleep 1                 # 7
end
```

Snippet 13: Surrounding Functions

```
== Trace
[0] -
[1] conVT: 0, cumVT: 0, isCall: false, inFunc: true
[2] conVT: 0, cumVT: 0, isCall: false, inFunc: true
[3] conVT: 0, cumVT: 0, isCall: true,  inFunc: false
[4] conVT: 1, cumVT: 1, isCall: true,  inFunc: false
[5] conVT: 0, cumVT: 1, isCall: false, inFunc: false
[6] conVT: 0, cumVT: 0, isCall: false, inFunc: true
[7] conVT: 1, cumVT: 1, isCall: false, inFunc: true
```

[9] At statement 4, the symbol called is not yet defined within Sonic Pi's scope so the symbol is unrecognised and the thread crashes.

The issues that Snippet 13 identifies is how to insert the correct amount of contributing virtual time into the trace when you might not have encountered the function being called. The simple way to handle this is by processing the data in two passes. On the first pass of a program we collect information such as which lines are function calls and what all of our function definitions and times are. It is not printed in these traces but one of the extra things that each statement index holds is, if the current line is a function call, which function does it call.

The trace after the first pass of this program looks as follows:

```
== Trace
[0] -
[1] conVT:  0, cumVT:  0, isCall: false, inFunc: true
[2] conVT:  0, cumVT:  0, isCall: false, inFunc: true
[3] conVT: -1, cumVT: -1, isCall: true,  inFunc: false
[4] conVT: -1, cumVT: -1, isCall: true,  inFunc: false
[5] conVT:  0, cumVT: -2, isCall: false, inFunc: false
[6] conVT:  0, cumVT:  0, isCall: false, inFunc: true
[7] conVT:  1, cumVT:  1, isCall: false, inFunc: true
```

Once this is processed it is again a simple case of adding up the cumulative virtual time as we traverse the trace. `-1` is reserved as a keycode in tandem with the `isCall` marker; if a function has `conVT = -1` it means this is a function call and will be processed in a later pass. It might seem like the keycode is useless in this case, but as the virtual time keycode is consumed during trace processing, it is useful to maintain the other boolean flag so that function calls are more easily identified afterwards.

The key idea is not to add together times from different blocks. The `inFunc` note helps with this, as we can detect when we are in a new block of the trace by what value that has. Notice that, in the above trace, index 6 has a fresh `cumVT` value when compared with index 5. The issue here is that the `inFunc` marker is still too simplistic. This handles sequential functions as each statement in `pTrace` holds information about which function it is part of when `inFunc` is set. Otherwise the model would be too simple and function times would run into each other. This is detailed more in Section 5.2.7.

Conditionals

Similar to functions, we require a two pass approach to handle conditionals in the code. Take the following program code:

```
if true then        #1
    sleep 0.5       #2
else
    sleep 1         #3
end
```

Snippet 14: Conditional

In practice it may be difficult to evaluate which branch of a conditional the program is going to take. Instead the approach is to find the maximum virtual time of both branches and take the longest one, as described in Section 4.1.2, Definition 1. In terms of timing later in the program this could potentially throw developers on what the actual virtual time of the program will be if the `sleep` lengths differ by a sizeable margin. On the other hand this is a much safer model in terms of ensuring the 'time saftey' of the program[10].

The trace for Snippet 14 is:

```
== First Pass Trace
[0] -
[1] conVT:  -2, cumVT: -2,   isCall: false, inFunc: false
[2] conVT: 0.5, cumVT: -1.5, isCall: false, inFunc: false
[3] conVT:   1, cumVT: -0.5, isCall: false, inFunc: false

== Second Pass Trace
[0] -
[1] conVT:  1, cumVT:    1, isCall: false, inFunc: false
[2] conVT: -3, cumVT: -1.5, isCall: false, inFunc: false
[3] conVT: -3, cumVT: -0.5, isCall: false, inFunc: false
```

In the first pass of the trace, we mark the location of the if with the keycode -2. After this we calculate the virtual time of each branch and store the index for the last state of each branch in the `IfNode` at the root of this expression block. Similar to how block detection works, the end of an if-branch is detected when the next index in question has the same parent index, or one of greater value (or the program has finished in the case the conditonal is at the end of the program). With this, the second pass is simply a case of, on reaching a keycode of -2, set the result of whichever branch was longest.

Dead Code

When calculating the virtual time of the program it is quite important not to take dead code into account. Dead code can easily push the trace into incorrect evaluation and make it difficult for the developer to calculate the time in their program. As well as this, being able to print when code will not be reached in a program is incredibly useful information for a developer to have in such an environment. Dead code can also be easy to locate thanks to the form of the program.

[10]The real running time of the code is guaranteed to be longer than the virtual time, as proved in [8], Section 4.1.

```
loop do
  play 60
  sleep 1
end

loop do
  play 60
  sleep 1
end
```

Snippet 15: Dead Code

We set `-3` to be the dead keycode and set all instances of `conVT` to this value when the statement is registered as dead. Here the second loop is obviously never going to be run, so we must remove this from consideration in our trace. The trace makes a note of when it has seen a loop block and sets a boolean flag for dead code on hitting another loop block. Should the trace hit a `in_thread` block or a `live_loop` block then this flag is not set as these pieces of code can run in parallel.

This is captured in our specification by the subsumption of *. As infinity plus any value is still infinity, we can consider the subsequent program to have virtual time zero, which amounts to not running it.

Variable Sleep

```
define :foo do |n|
  sleep n
end
```

Snippet 16: Variable Sleep

Snippet 16 shows a common construct in Sonic Pi; the idea of functions allowing variable sleep amounts. This is not currently supported in this iteration of the project[11], but the foundations for it are well-placed. Each `block` defines an `arg` child which lists the symbols for all arguments passed to it. Given this we can define a new keycode in our `pTrace` and use the same style of interprocedural analysis as before. In the case of Snippet 16, the function can be defined in terms of its total variable time. Then, when processing the function call itself, the numerical argument is taken and applied to the expression stored for the function[12]. A full trace of this is given in Section 5.2.7.

[11] This is owing to time constraints and feature prioritization.
[12] Argument symbols are stored as `lvar` type nodes, making expression detection and evaluation simple to manage.

4.2 Session Pi

Session Types describe communication protocols via message passing. In Sonic Pi the message originates at a `cue` statement and is 'consumed' by the `sync` statement. Given this, we elect to transform the AST used in timing analysis into a directed graph to better represent the interactions. In this chapter, we first formalise the concepts of Session Types; we also set the context for Global Types and Multi-Party Session Types through a simple example and formalise the idea of *Projection*. After this we describe the design of the graph, which details two core iterations of the graph. Finally, we relate this implementation to the theory presented in 4.2.1. We explain our approach to *Global Inference* and present our own extensions to Multi-Party Session Types: the `&&`, and `||` types.

4.2.1 Formal Session Types

Before writing the communication interactions of Sonic Pi, we provide a formalisation of Session Types. This defines a series of constructs that may be used to describe the types of communication a process may have and which other processes it may interact with. By defining and analysing these patterns we are able to verify which programs are safe from concurrency errors like deadlocks and reject those that have bad communication structure. This static verification enables developers to easily reason about the different interactions within their program.

As in Section 4.1.2, the syntax of Sonic Pi V2.0 is simple.

$$
\begin{aligned}
&\texttt{P} ::= \texttt{P; S} \mid \emptyset \\
&\texttt{S} ::= \texttt{E} \mid v \texttt{ = E} \\
&\texttt{E} ::= \texttt{sleep } \mathbb{R}_{\geqslant 0} \mid \texttt{cue } v \mid \texttt{sync } v \\
&\qquad \mid \texttt{A}^i \mid v
\end{aligned}
$$

We expand the syntax to define `cue` and `sync` which operate on some variable name, v. We maintain our definition of `sleep` in this syntax as it is a useful tool in our reasoning. $\texttt{A}_i$ is thus the set of all operations that either do not advance time or do not consist of some communication primitive. In this way we can abstract over the operations that do not contribute to communications.

This definition is lacking the syntax for function calls and conditionals as this analysis does not currently support them. These limitations are discussed in greater details towards the end of Section 4.2.

```
# P0
in_thread do
  loop do
    cue :B
    sync :A
    sleep 1
    play 63
  end
end
```

```
# P1
in_thread do
  loop do
    cue :A
    sleep 1
    sync :B
    sleep 0.5
  end
end
```

Snippet 17: Forming Types

For the following section, the base set of syntax we use to describe our session types are as follows[13]:

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

The final set of interest is that of *Processes*, ranged over by `P, Q...`. For this discussion we define the following aspects of the *Process* grammar:

```
P ::= k![ẽ]; P                    data sending
    | k?(x̃) in P                  data reception
    | def D in P                   recursion

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

We maintain this definition of recursion as it is useful to be able to apply it to the loops in our processes. Similarly to the way we defined an infinity operator in Section 4.1.2, this allows us to only consider the first iteration of a loop when handling session types. For the full performance the session types can unfold as many times as you require, but the developer only needs to see that type information once.

The central idea to binary session types 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

[13] Other sets used are *constants*, *expressions*, and *process variables* but we do not need to consider these for the language constructs discussed in this report.

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. We can consider the Sonic Pi IDE to be the session that we are using to generate channels to communicate on. Via our channels we perform the atomic operations of data sending and receiving:

```
k![e1, ..., en]; P       # Data Sending
k?(x1, ..., xn) in P     # Data Receiving
```

With binary session types defined we may now formalise the idea of a dual type. The dual relation defines the link between two endpoints of communication between two participants in binary sessions. The endpoint of communication for each participant can be thought of as the next message to consume. We have two dual relations that we are concerned with:

`!` = `dual(?)`, `?` = `dual(!)`

These are important statements as they are what we use to define the way our protocols interact with each other. In addition to these binary type relations, this project uses the multi-party session typing paradigm, which presents a slight improvement to this syntax, allowing our communication to occur between more than two *participants*.

Participants are the set ranged over by *p, q,...*

```
P ::= c!<p,e>.P              value sending
    | c?(p,x).P              value reception
```

With this we can now state which of our participants we are sending our message to and on which channel. With this we are now able to begin typing our programs.

Global Types

To begin illustrating global types by example, we present a likely scenario to occur in a piece of music. First drum, `D1`, sends a 'cue' to the melody, `M1`, to start the song in sync. In responce `M1` sends a 'cue' to both `D1` and a half-drum, `D2`. After some time has passed, `D2` may have fallen behind time slightly, so `D1` sends another message to sync these two beats together. After this `D2` can 'cue' a waiting riff, `M2`, to decorate the music. Decomposing this into several binary session types could be logically very difficult, and most likely we would lose some of the essential sequencing information in the process. Instead, let us transform this scenario into the following global type.

```
D1 → M1 . M1 → D1 . M1 → D2 .
D1 → D2 . D2 → M2
```

This example shows how simply the communication structure of a program, using multiple processes, can be described using Multi-Party Session Types. In our analysis, a message can defined a passing from process `P0` to process `P1`, written `P0->P1`, if the types in processes `P0`

and `P1` dual each other. The global types describe the various actions of a participant `p` sending either a value of a given Sort, a channel of a given Type, or some label to the partcipant `q` and then interaction continues according to the following global type `G`. The grammar we are interested in here is as follows:

```
G ::= p → q : ⟨S⟩.G          # Value Exchange
    | t | end                # Recursion/End

S ::= bool | ... | G         # Sorts
U ::= S | T                  # Exchange Types

T ::= !⟨p, S⟩.T              # Send Value
    | !⟨p, T⟩.T              # Send Channel
    | ?(p, U).T              # Receive
    | t | end                # Recursion/End
```

Projection

The global type is the description of the program as the whole. It shows which participants communication with which and what data they pass between each other. The decomposed local type describes the interaction that specific process is aware of.

> **Definition 2:** *The projection of a global type `G` onto a particiapnt `q` is defined by induction on `G`:*

$$(\texttt{p} \to \texttt{p'} : \texttt{<U>}.G')\upharpoonright \texttt{q} = \begin{cases} !\langle \texttt{p'},\texttt{U}\rangle(G' \upharpoonright \texttt{q}) & if\ \texttt{q} = \texttt{p} \\ ?(\texttt{p'},\texttt{U})(G' \upharpoonright \texttt{q}) & if\ \texttt{q} = \texttt{p'} \\ G' \upharpoonright \texttt{q} & otherwise \end{cases}$$

$$\texttt{t} \upharpoonright \texttt{q} = \texttt{t} \qquad \texttt{end} \upharpoonright \texttt{q} = \texttt{end}$$

The usual approach to multi-party session types is to construct the global type of your program and then project the individual process types from that. With this, one can then write each process, ensuring that all communication protocols remain consistent as development proceeds. Projection is a simple and decidable approach to session times.

This is not how we handle multi-party session types. This approach is not so feasible in this instance as we have already been presented with the full source. In this project we examine *Global Inference* as we are building the global type out of a set of pre-written local types. Whilst notably more difficult, this approach is also, thankfully, proveably decidable. There is no situation where any algorithmic approach to the problem will not terminate with some answer. In the cases where our program's global type is not typeable, this suggests some communication error within the provided program. This report presents a simple, but elegant, way of handling

global inference in this case. We also present two new constructs, as nothing already available in the theory of multi-party session types could describe the situation we required. These are detailed further in Section 4.2.3.

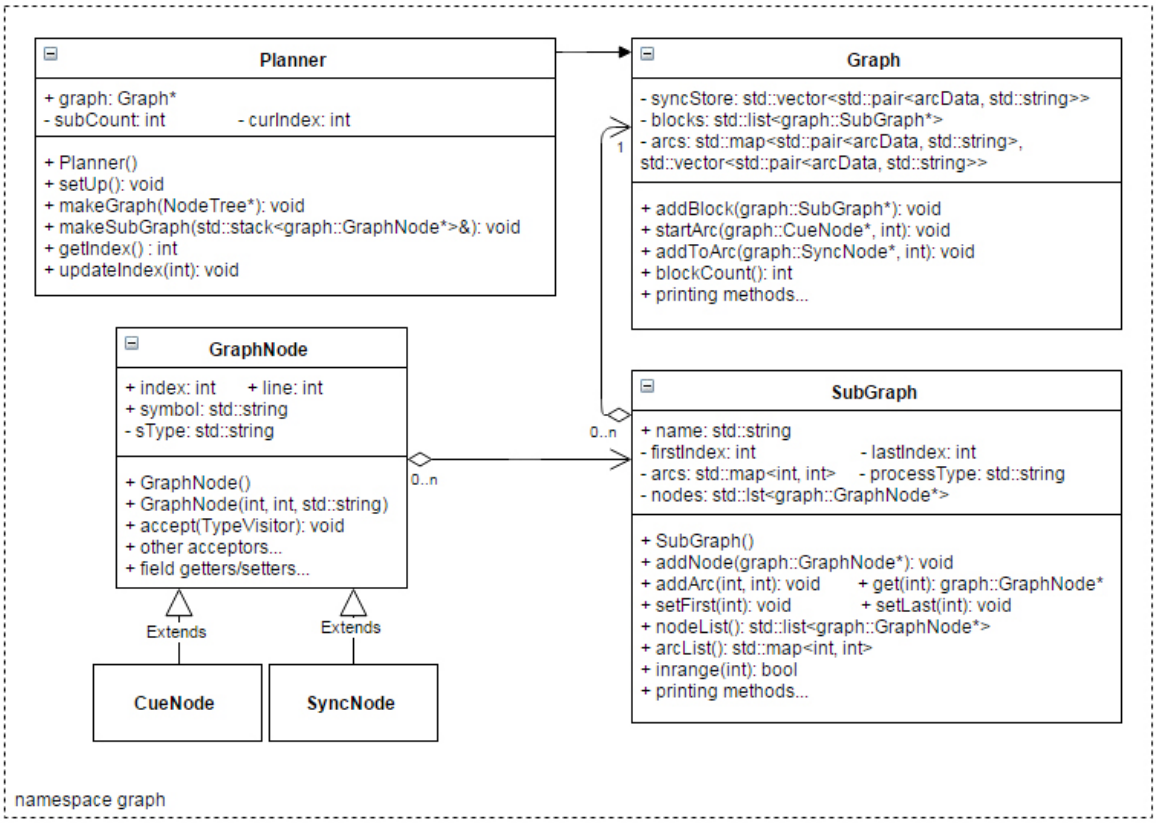


Figure 9: Graph UML

4.2.2 Planning the Graph

As shown by the Figure 9, each `Graph` is constructed as a list of `SubGraph`s where each `SubGraph` represents one process within the program. This structure is shown again more explicitly in Figure 10, which also shows the various types of arc that this analysis graph makes use of.

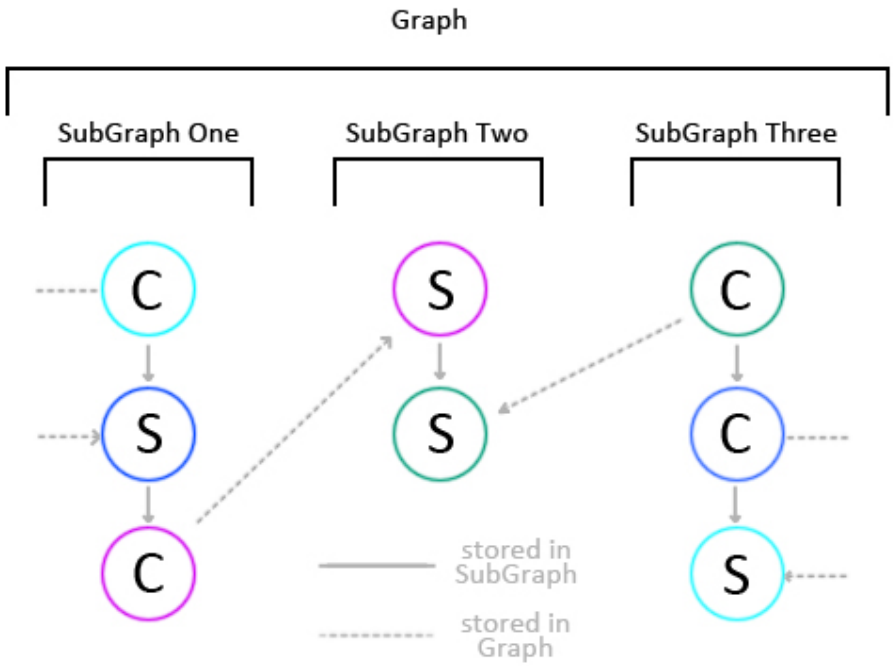


Figure 10: Graph Structure

The idea behind this is to maintain the chronology of the nodes in each process and to enable easy analysis of the program on a per-process level. In Sonic Pi there is a `cue` operation, which sends out an asynchronous message to all processes. To complement this is a `sync` operation, a blocking operation that causes the process to wait to receive a message. Immediately we draw the parallel between these actions and the sending/receiving messages that Session Types describe. Linking back to the dual relation, `cue` acts as our send (which we type with a `!` session type) and `sync` acts as our receive (which we type with a `?` session type). `SubGraph` handles the construction of local types while the `Graph` class handles the global type.

Our approach to building the global type with this graph is simple. We describe it here with pseudo code. It introduces a concept of 'tokens' in the system. We use this phrase to refer to each session type statement in the local types (e.g. in `B:(P0P1)!.A:(P1P0)?` each token is separated by . in the string, meaning this string has token length two).

```
syncStore  <- new
prevTokens <- new   # initial attempt with allowing sub-typing
```

```
while every process has tokens
|   or every non-empty process is not in syncStore:
|   currTokens <- new
|   for each process:
|   |   # sync is blocking
|   |   # so as long as a process has a token in sync
|   |   # we can't consume it
|   |   if syncStore.members.name != process.name
|   |   |   currTokens += process.top
|
|
|   check if currTokens dual each other:
|   |   build type; remove from token list
|
|   check if currTokens dual any in syncStore
|   |   build type; remove from respective store
|
|   check if currTokens dual any in prevTokens
|   |   build type; remove from respective store
|
|   syncStore += currTokens.syncTokens
```

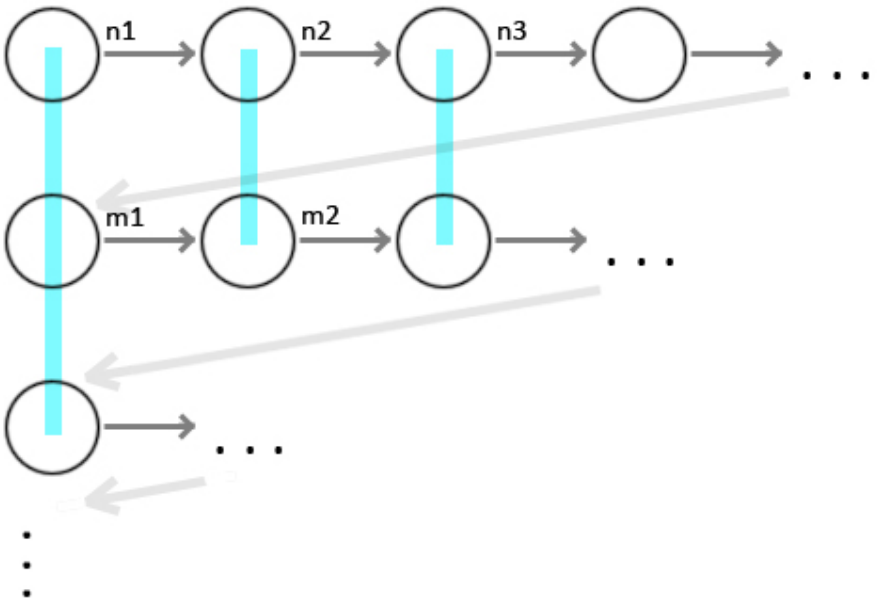


Figure 11: Node Chronology

The reason we process tokens in this way is demonstrated in Figure 11. A Sonic Pi program will

always increment its virtual time as we run through a program. `SubGraph`s are written in the order they are defined in the source by line number. Given this, within a `SubGraph` if `n1.index < n2.index`, `n1` has resolved first. We will refer to this as the 'horizontal' relationship between nodes. In two concurrent `SubGraph`s, `n.index < m.index`, and `n` will be resolved first by the interpreter, but they may have the same virtual time within the program. Given this they are compared to each other as well as those `sync` tokens processed previously. We will refer to this as the 'vertical' relationship between nodes and is demonstrated in Figure 11 by the blue bar linking the different groups together.

There is a small limitation in this design as it will allow certain untypable programs to be typed.

Take the two code snippets below. Snippet 19 is typeable and Snippet 18 is not, yet both will produce this typable sesstion type with the algorithm described so far:

```
P0 := foo:(P0P1)!.bar:(P1P0)?
P1 := bar:(P1P0)!.foo:(P0P1)?
```

In these cases `:foo` and `:bar` are both the name of the channel *and* the name of the current process. As the library is currently unable to assign user-provided names to the processes, we choose to comment in the name that the library will generate for the process. This also serves to make the distinction between channel name and process interaction clear.

The analysis falls apart with the current algorithm of focus because the basic method of iterating over each local type does not translate well when handling sub-typing.

```
live_loop :foo do     # P0
    play :e4, release: 0.5
    sleep 0.5
    sync :bar
end

live_loop :bar do     # P1
    sample :bd_haus
    sleep 1
    sync :foo
end
```

Snippet 18: Untypable

```
live_loop :foo do     # P0
    sync :bar
    play :e4, release: 0.5
    sleep 0.5
end

live_loop :bar do     # P1
    sync :foo
    sample :bd_haus
    sleep 1
end
```

Snippet 19: Typable

Sub-Typing

Under the initial algorithm, both `cue` statements in Snippet 20 would be processed and then discarded, as `cue` statements do not persist in the statement store. However, these statements all execute at the same virtual time in the program; both `cue`s will trigger both `sync`s in the Sonic Pi environment. Storing the previous round of tokens has also already been shown to be inaccurate.

```
P0 := B:(P0P1)!.A:(P1P0)?
P1 := A:(P0P1)!.B:(P0P1)?
```

In standard binary session types this is inaccurate:

```
B:(P0P1)!.A:(P1P0)?
   ≠ dual(A:(P0P1)!.B:(P0P1)?)
```

The accurate type for the second process would be `B:(P0P1)?.A:(P0P1)!`, (or one could keep `P1` unchanged and swap the terms in the other process). By introducing the concept of time progression, we are able to handle this typing.

```
# P0
in_thread do
  loop do
    cue :B
    sync :A
    play 60
    sleep 0.5
  end
end


# P1
in_thread do
  loop do
    cue :A
    sync :B
    play 64
    sleep 0.5
  end
end
```

Snippet 20: Sub-Typing `cue`

Our implementation utilises sub-typing the `cue` type.
The formal outline for this feature is:

```
A!.P <: P.A!
```

Given this we may say that:

```
A:(P0P1)!.B:(P0P1)? <: B:(P0P1)?.A:(P0P1)!
B:(P0P1)!.A:(P1P0)? = dual(B:(P0P1)?.A:(P0P1)!)

∴ B:(P0P1)!.A:(P1P0)? ≅ dual(A:(P0P1)!.B:(P0P1)?)
```

Sub-typing is a useful relation to define within session types, see [20, 26, 27], and is based on the notion that correctness will always be maintained so long as input order is not disrupted in the message queue. Here, our subtype is based on operations occuring at the same 'virtual time', so we may consider operation ordering to be maintained in this instance.

Keeping Time

To handle sub-typing effectively, we introduce a blank `GraphNode` into the graph structure described in Section 4.2.2. This node represents when virtual time has incremented in the program, helping us identify which communication operations interact with which.

This does not break the order relation described previously but it does change the grouping of the 'vertical' relationship of the nodes. Rather than mapping each column as one relation, with the total set of relationship being equal to the maximum number of types in any one `SubGraph`, each 'vertical' relation can be defined as the set of nodes between each *time* node.

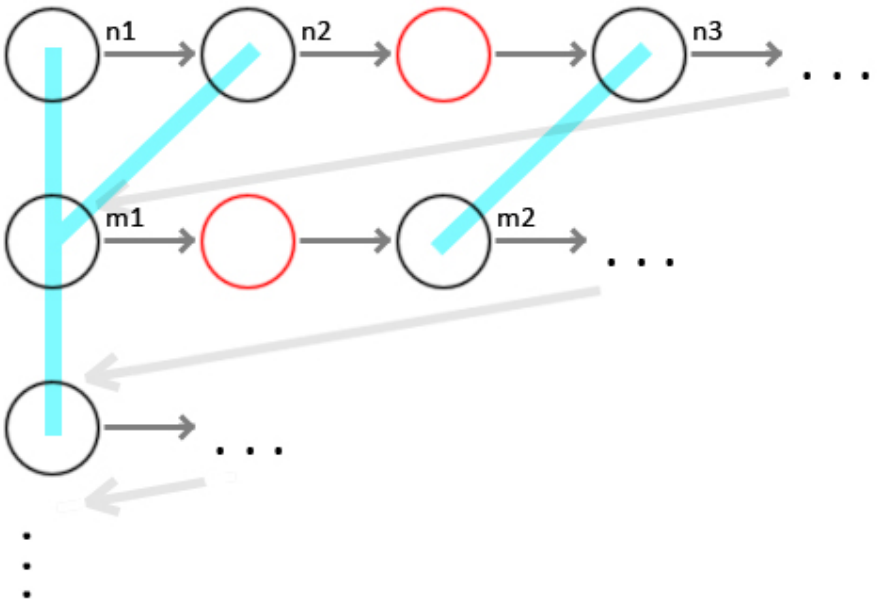


Figure 12: Fixed Node Chronology

Refering back to Snippet 19, the program graph constructed now takes the form:

```
P0 :=  foo:(P0P1)!.bar:(P1P0)?.time
P1 :=  bar:(P1P0)!.foo:(P0P1)?.time
```

More importantly the program from Snippet 18 will look as follows:

```
foo := foo:(P0P1)!.time.bar:(P1P0)?
bar := bar:(P1P0)!.time.foo:(P0P1)?
```

When analysing these local types, the only change to the pseudo code presented before is how we consume tokens at the start of each loop. Instead of only taking the top token from each

process, we keep collecting tokens until we either hit a `sync` (it is still a blocking call), or we hit `time`. Now the analysis simply processes all the tokens in hand and builds the global type according to the same typing rules as before. Figure 12 shows how the graph structure has changed visually from the previous attempt.

To demonstrate further, take the types presented for Snippet 19. With this algorithm we consume all four tokens presented in the same pass. Because of this we can sub-type a `cue` as needed and the system types correctly. In contrast, with the local types for Snippet 18, we can only consume the top token from each process on the first iteration. We skip the `time` marker and are then presented with a `sync` node at the top of each process. Here we can successfully report this to be a deadlocking error. Both participants are waiting for a message from the other one!

4.2.3 Global Inference

Here we outline how we implement the global inference initially discussed in Section 4.2.1.

Given the dual types `A:P0P1?` and `A:P0P1!` we can transform this into the global typing `P0->P1`, representing the fact that a message is being transfered from `P0` to `P1`. `A` is not represented within this global type as it acts as the name of the channel on which the message is being sent. Standard global types also print the type of message being sent; we currently omit this type information. We can consider Sonic Pi to be a series of 'null' messages being sent on each channel between each participant.

The syntax of Sonic Pi allows for some interesting situations, some of which are not technically typable in standard multi-party session types but are perfectly valid, and sometimes useful, situations to occur in Sonic Pi.

Replication Type

The first such situation we consider is when we have several `cue` signals with the potential to trigger one `sync`.

In this scenario the `sync` in `P1` may be triggered by either of the `cues` present in the other two threads. Whilst unusual, it is not unreasonable to suggest this is a valid program. It may be that the developer has set up the rest of the program code so that threads `P1` and `P2` alternate for each loop that `P1` performs. The use of this in a musical sense could be an interesting drum riff to underpin the song, or some decorative melodies. To capture this information Sonic Pi proposes an '*or*' type, denoted by `||`.

```
# P0
in_thread do
  loop do
    cue :A
    sleep 1
  end
end
```

```
# P1
in_thread do
  loop do
    sync :A
    sleep 1
  end
end
```

```
# P2
in_thread do
  loop do
    cue :A
    sleep 1
  end
end
```

Snippet 21: One `sync`, multiple `cue`

Given this notation the decomposed types[14] for this set of threads is:

```
P0 := A:(P0P1)!
P1 := A:(P0P1 || P2P1)?
P2 := A:(P2P1)!
```

In terms of processing the global type, when each of these types line up into the same 'vertical' relationship, the type constructed has the form `(P0 || P2)->P1`. The algorithm discussed previously does not need adapting to handle this new type form. The tokens are not removed from the stored pool until every node in the group has been processed. We are only required to add in some new code to handle the case where we have detected the extra information else we may actually print the types as a composition, `P0->P1.P2->P1`. This is a valid session type but is not an accurate definition for this situation.

Broadcasting

The other situation we considered is when one `cue` can trigger multiple `sync` statements. This is equally as untypable in strict session types but equally as viable from a musical point of view. The developer may have multiple melodies playing concurrently that should all sync onto the same drum beat.

[14] For the purpose of these discussions we again omit the `time` element of the type as it is not relevant.

```
# P0
in_thread do
  loop do
    sync :A
    sleep 1
  end
end
```

```
# P1
in_thread do
  loop do
    cue :A
    sleep 1
  end
end
```

```
# P2
in_thread do
  loop do
    sync :A
    sleep 1
  end
end
```

Snippet 22: One `cue`, multiple `sync`

In this situation we propose a simple '*and*' type. `cue` is an asynchronous message that propogates out to all processes currently waiting. Given this, the `cue` present in `P1` will trigger the `sync` in `P0` *and* the `sync` in `P2`. The types we print for this are as follows:

```
P0 := A:(P1P0)?
P1 := A:(P1P0 && P1P2)!
P2 := A:(P1P2)?
```

As before, the algorithm for creating global types from this type set is not affected. The `sync` statements will either be in the store from a previous iteration of the algorithm or be handled as all statements from the same 'vertical' relationship will be processed. The global type for this situation will be printed as `P1->(P0 && P2)`. In the event these statements are separated by time progression, the type will print as the usual global type involving two processes. `sync` statements will still be stored as described earlier, either to be consumed by a later `cue` of the correct dual typing or to be hit again and processed as an untypable program.

On initial consideration one might define the `||`-type as a dual of the `&&`-type, and vice versa, but this is inaccurate for the context in which they are used. The main component of these types is still the act of sending or receiving a message, so dual typings are still based off of the dual relationship of '`?`' and '`!`'. The two types we introduce here are more like a 'typing sugar' to describe situations where a process can send and received multiple messages at the same time. This is not a situation that session types usually handle. Session Types are defined by their use of message queues, sending message packets (be they single packets or multiple) in sequence along the *same* channel to *single* receivers. It does not define a situation where one can trasmit a message on one channel to multiple receivers, better known as 'broadcasting'. In this case this projects presents an interesting evolution of session types for this field. We have implemented multiple modes that our project can operate under to reflect this, so developers can choose to analyse under standard session typing or under the 'Sonic Pi' typing outlined here.

4.3 Integration

The Sonic Pi IDE has two core sections: Server and GUI. The server is written on top of the SuperCollider synthesizer and implemented in Ruby whilst the GUI is built on the Qt framework and thus written in C++.

Integrating this library comes in two stages. Firstly delving into the server to locate the area where the code is evaluated before being made into music. Sonic Pi's code evaluation is handled by a `spider` module, which holds the main evaluation method. This module also has the capacity to print straight into the IDE's output log widget, making the first step of relaying information to the screen very simple. The pseudo code for this `spider` integretion is very simple.

```
def __spider_eval(code)
    # thread setup
    # ...
    code = Preparse(code)
    analysis_output = __verify_code(code)
    __info(analysis_output)
    # ...
    # message handling and thread handling
    # ...
end
```

Secondly, to better display the timing effects information, we choose to define a new output widget in the `mainWindow` class of the gui. Qt, thankfully, simplifies the GUI process considerably, meaning it is not too difficult to produce an attractive addition in keeping with the existing theme of Sonic Pi. We are also able to mimic the `__info` method we use to print to the output log to be able to print our timing information into our new widget.

5 Evaluation

5.1 Ruby/C++11 & Performance

In evaluating our project it is necessary to question whether the tools used were in fact the right ones for the given task. We have implemented an analysis library in C++11 which successfully hooks into the existing Ruby modules. This library could easily have been written in Ruby, saving the need for the extra steps of converting the data from one type to another.

The reason for our use of C++11 is to take advantage of the speed of the language. The tool used for for the data transfer here is Ruby Rice, a type-safe and exception-safe interface between C++ and Ruby's C API[15]. Other similar tools for this are Ruby Inline and Ruby's FFI tool. Below is a table of their relative performances based on some short implementations of fibonacci, factorial and pow methods.

Table 1: Ruby Interfacing Performances [6]

	Ruby Inline	Ruby Rice	Ruby FFI
factorial	0.026175138	0.197720523	0.014882004
fibonacci	0.026792521	0.202714029	0.018928646
pow	0.03252452	0.211258897	0.023082315

Disappointingly, Ruby Rice performs the slowest of this tools in this scenario. On standard desktop environments this amount is little cause for concern as both the Sonic Pi IDE and the extention library run smoothly with no trouble. This being said, it is important to remember that these results are based on very simple mathematical methods, not something you would likely use embedded C for in a real project. The library is also able to run within the Sonic Pi IDE on its target platform with little to no noticeable difference in run time.

While these results are strong it doesn't go the full distance in answering if the library was better served implemented in Ruby or C++11. The library as it is, is able to take advantage of many useful features of C++11; notably the host of available data structures for each task. That said, whilst Ruby only has Array and Hash structures avaiable to it, Ruby does still have the concept of classes and modules meaning our current class setup is largely portable to a Ruby environment. Unfortunately there has not been time thus far to test how fast a Ruby implementation is compared to the current project state.

Another reason for the use of C++ was the hopes to make greater use of the boost graph library[16]. In implementing the graph analysis with this it would have been useful to have immediate access to such things as the iterators and many different graph algorithms (djisktra, etc...) that bgl implements. Currently the `SubGraph` structure we have does not translate well into bgl. It is possible to compile this implementation but data is tricky to initialise and access later on. In the interest of time this idea was shelved and a much simpler graph structure was

[15] As found at https://github.com/jasonroelofs/rice
[16] http://www.boost.org/doc/libs/1_58_0/libs/graph/doc/

put in place. Despite the large code dependancy bgl can introduce, it may be interesting to revisit the idea in the future, should the search-algorithms it has ready-implemented become required.

5.2 Correctness

To confirm the correctness of our approach we have built a systematic testing environment. It is important when building verification tools to have a series of programs to test results with and ensure any interesting edge cases of behaviour can be detected and dealt with. In this system we been with a series of simple programs, testing simple chords and sequencing, single function detection, etc..., before moving into more complex programs. At the later end of the suite we test on a full musical piece, taken from the samples provided in the Sonic Pi IDE. In presenting these results, we provide the source for the program, an expected trace result and then our actual analysis result.

All codes written here can be repeated with notes in MIDI format (`:C4`, for example) and will produce the same results as the integer format presented. The reason for this is that the note values are wrapped into `SymNode` types in the same way the numbers are wrapped into the `IntNode` type. This means the analysis approach for them is the same and the results are deterministic. Session analysis results are only presented when relevant to the code being discussed, as the library is quite efficient at setting empty types. Where functions are not being tested, trace information such as whether the statement is a function call or not is ommitted for brevity.

5.2.1 Chords

```
play 60
play 62
play 64
```

Chord Test Code

```
== Expected Trace
[0] -
[1] conVT: 0, cumVT: 0
[2] conVT: 0, cumVT: 0
[3] conVT: 0, cumVT: 0
```

```
== Actual Trace
[0] -
[1] conVT: 0, cumVT: 0
[2] conVT: 0, cumVT: 0
[3] conVT: 0, cumVT: 0
```

This trace correctly confirms that the given program does not advance virtual time at any points, reporting a statement length of three.

5.2.2 Sequences

```
play 60
sleep 1
play 62
sleep 1
play 64
```

Sequence Test Code

```
== Expected Trace                    == Actual Trace
[0] -                                [0] -
[1] conVT: 0, cumVT: 0               [1] conVT: 0, cumVT: 0
[2] conVT: 1, cumVT: 1               [2] conVT: 1, cumVT: 1
[3] conVT: 0, cumVT: 1               [3] conVT: 0, cumVT: 1
[4] conVT: 1, cumVT: 2               [4] conVT: 1, cumVT: 2
[5] conVT: 0, cumVT: 2               [5] conVT: 0, cumVT: 2
```

Here we correctly identify five statements, two of which advance time. The final length of virtual time is the sum of all `sleep` statements used. Here this is accurate as it is a simple sequence of operations.

5.2.3 Loops

```
loop do
  play 60
  sleep 1
end
```

Loop Test Code

```
== Expected Trace                    == Actual Trace
[0] -                                [0] -
[1] conVT: 0, cumVT: 0               [1] conVT: 0, cumVT: 0
[2] conVT: 0, cumVT: 0               [2] conVT: 0, cumVT: 0
[3] conVT: 1, cumVT: 1               [3] conVT: 1, cumVT: 1
```

The trace identifies three statements, where on is the commencement of the loop. By the end of the loop, virtual time will have advanced by one, as printed by our analysis.

```
loop do
  play 60
  sleep 1
  loop do
    play 64
    sleep 1
  end
end
```

Nested Loop Test Code

```
== Expected Trace                    == Actual Trace
[0] -                                [0] -
[1] conVT:  0, cumVT: 0              [1] conVT:  0, cumVT: 0
[2] conVT:  0, cumVT: 0              [2] conVT:  0, cumVT: 0
[3] conVT:  1, cumVT: 1              [3] conVT:  1, cumVT: 1
[4] conVT:  0, cumVT: 1              [4] conVT:  0, cumVT: 0
[5] conVT:  0, cumVT: 1              [5] conVT:  0, cumVT: 0
[6] conVT:  1, cumVT: 2              [6] conVT:  1, cumVT: 1
```

This has highlighted a limitation in the current iteration. The contributing time detection works but the cumulation does not accurately detect nested blocks. Instead it sees each new block and starts the cumulation fresh, as those they were sequential blocks. This shows the project is not currently making the most use out of the block level data that is taken during the construction of the AST. This could be fixed by testing whether a new block has a higher level than the previous token. In the event it does the trace can skip starting a fresh `cumVT` value, if it is equal or less than then the old behaviour applies.

```
5.times do
  play 60
  sleep 1
end
```

'Timed' Loop Test Code

```
== Expected Trace                    == Actual Trace
[0] -                                [0] -
[1] conVT: 5n, cumVT: 0              [1] conVT: -1, cumVT: -1
[2] conVT:  0, cumVT: 0              [2] conVT:  0, cumVT: -1
[3] conVT:  1, cumVT: 1              [3] conVT:  1, cumVT:  0
```

This loop structure is currently not handled correctly. Ideally, since this is set to run a set number of times, once the trace has processed the loop length, there should be some stored multiplier that can be used to accurately set the `cumVT` value after the loop to a much higher number (in this test case, the next `cumVT` would be `5` plus whatever the next `conVT` evaluates to).

What actually happens is `times` is registered as a function. In ruby terms, this is technically a function being sent the value 5 to act on. What we may do in future is set another keycode (-4) which tells the second pass of the trace that there is an argument to this loop that it must multiply the total loop time by and then use this value in subsequent trace calculations.

```
5.times do
  play 60
  sleep 1
  5.times do
    play 64
    sleep 1
  end
end
```

Nested 'Timed' Loop Test Code

```
== Expected Trace                    == Actual Trace
[0] -                                [0] -
[1] conVT: 5n, cumVT: 0              [1] conVT: -1, cumVT: -1
[2] conVT:  0, cumVT: 0              [2] conVT:  0, cumVT: -1
[3] conVT:  1, cumVT: 1              [3] conVT:  1, cumVT:  0
[4] conVT: 5n, cumVT: 1              [4] conVT: -1, cumVT: -1
[5] conVT:  0, cumVT: 1              [5] conVT:  0, cumVT: -1
[6] conVT:  1, cumVT: 2              [6] conVT:  1, cumVT:  0
```

This trace carries forward the limitations of both nested and timed traces shown previously.

There is also the ability to make a parameterized loop with `n.times`, where `n` has been passed to the function the loop is contained in. This test is not included here for brevity, as the results are similar those presented here. The reason for this is whilst the library can see the symbols easily, the code is currently not processing it. For this reason, parameterized code is processed as if they were unparameterized. In the case of loops, this means all loops are handled once and VT is printed as though each loop only occured once.

5.2.4 Threads

```
in_thread :foo do
  play 60
  sleep 1
end
```

Thread Test Code

```
== Expected Trace                    == Actual Trace
[0] -                                [0] -
```

```
[1] conVT: 0, cumVT: 0          [1] conVT: 0, cumVT: 0
[2] conVT: 0, cumVT: 0          [2] conVT: 0, cumVT: 0
[3] conVT: 1, cumVT: 1          [3] conVT: 1, cumVT: 1
```

Similarly to the simple loop, the analysis presents that this thread has virtual time one by the end.

5.2.5 Data Structures

Lists

```
play [60, 62, 64]
```

List Test Code

```
== Expected Trace               == Actual Trace
[0] conVT: 0, cumVT: 0          [0] conVT: 0, cumVT: 0
```

This is a single statement with a different syntax to those usually processed. The trace can accurately identify this has no contribution to virtual time despite the change in arguments.

Chords & Scales

```
chord(:E3, :minor)
```

Chord(DS) Test Code

```
scale(:E3, :minor)
```

Scale Test Code

Both of these structures have the trace results:

```
== Expected Trace               == Actual Trace
[0] conVT: 0, cumVT: 0          [0] conVT: -1, cumVT: 0
```

The cumulative times are marked as zero here because singular statements are not preceeded by a `begin` node. This means the current information is stored in a `RootNode`, which by default is not processed for information in the current iteration of the project. Running this again with the statements as part of a statement list give:

```
== First Pass Trace                    == Second Pass Trace
[0] conVT: -1, cumVT: (prev-1)         [0] conVT: 0, cumVT: 0
    isCall: true, inFunc: false            isCall: true, inFunc: false
```

This shows that `chord` and `scale` are currently being detected as functions and as they are not defined in our user-function list, they are being filled as a zero time statements. Whilst they technically are zero sleep function calls, it raises the question as to whether we should only mark user defined functions or if it is still useful to mark the statement.

Rings

```
(ring 52, 55, 59)
```

Ring V1 Test Code

```
== Expected Trace                      == Actual Trace
[0] conVT: 0, cumVT: 0                 [0] conVT: 0, cumVT: 0
[1] conVT: 0, cumVT: 0                 [1] conVT: 0, cumVT: 0
```

This function call is in a similar state to the list test run previously. The parentheses cause the AST in this case to present a root `begin` node in every case, rather than allowing a single statement as before. Again, the analysis is able to identify this has no effect on the current virtual time and is therefore correct.

```
[52, 55, 59].ring
```

Ring V2 Test Code

The results for this style of function call are the same as those just discussed for chords and scales. Other data structure functions (`range`, `bools`, `knit`, and `spread`) have similar results to these.

5.2.6 Dead Code

```
loop do
  play 60
  sleep 1
end

loop do
  play 60
  sleep 1
end
```

Dead Code V1 Test Code

```
== Expected Trace                    == Actual Trace
[0] -                                [0] -
[1] conVT:  0, cumVT: 0              [1] conVT:  0, cumVT: 0
[2] conVT:  0, cumVT: 0              [2] conVT:  0, cumVT: 0
[3] conVT:  1, cumVT: 1              [3] conVT:  1, cumVT: 1
[4] conVT: -3, cumVT: 0              [4] conVT: -3, cumVT: 0
[5] conVT: -3, cumVT: 0              [5] conVT: -3, cumVT: 0
[6] conVT: -3, cumVT: 1              [6] conVT: -3, cumVT: 1
```

`-3` is the keycode used to show deadcode. The output in the IDE can use this to print blank statements rather than numbers, helping the developer see when some code is inaccessable. This trace also shows sequential cumulation is being calculated accurately as the trace does not waste time fixing the `cumVT` values processed during the first pass.

```
loop do
  play 60
  sleep 1
  loop do
    play 64
    sleep 1
  end
  play 66
  sleep 1
end
```

Dead Code V2 Test Code

```
== Expected Trace                    == Actual Trace
[0] -                                [0] -
[1] conVT:  0, cumVT: 0              [1] conVT:  0, cumVT: 0
[2] conVT:  0, cumVT: 0              [2] conVT:  0, cumVT: 0
```

```
[3] conVT:  1, cumVT: 1
[4] conVT:  0, cumVT: 1
[5] conVT:  0, cumVT: 1
[6] conVT:  1, cumVT: 2
[7] conVT: -3, cumVT: 2
[8] conVT: -3, cumVT: 3
```

```
[3] conVT:  1, cumVT: 1
[4] conVT:  0, cumVT: 0
[5] conVT:  0, cumVT: 0
[6] conVT:  1, cumVT: 1
[7] conVT: -3, cumVT: 1
[8] conVT: -3, cumVT: 2
```

This also highlights the error found earlier in collecting time for nested loops, but the dead code is still detected properly.

The type of dead code that the trace cannot comment on is when a conditional is never going to enter a certain branch. This analysis does not ever evaluate the results of conditionals, so it could be that we mark the `ifNode` with the virtual time of a branch it will never reach. The developer will have to notice this themselves during the performance when their music always runs for a length of virtual time shorter than what the analysis reports.

5.2.7 Function Calls

Function Definition

```
define :func do
  play 55
  sleep 1
end
```

Function Definition Test Code

```
== Expected Trace
[0] -
[1] conVT:  0, cumVT: 0,
    isCall: false, inFunc: true
[2] conVT:  0, cumVT: 0
    isCall: false, inFunc: true
[3] conVT:  1, cumVT: 1
    isCall: false, inFunc: true

Function: func 1
```

```
== Actual Trace
[0] -
[1] conVT:  0, cumVT: 0
    isCall: false, inFunc: true
[2] conVT:  0, cumVT: 0
    isCall: false, inFunc: true
[3] conVT:  1, cumVT: 1
    isCall: false, inFunc: true

Function: func 1
```

The analysis will print out a list of those function definiitions found during the first iteration. Here we have correctly identified that a function, called `func` was defined at statement index 1. We can also see it has correctly identified which statements where inside this function and the virtual time data throughout is accurate.

Function Detection

```
define :func do
  play 55
  sleep 1
end

func
```

Function Detection Test Code

```
== Expected Trace
[0] -
[1] conVT:  0, cumVT: 0,
    isCall: false, inFunc: true
[2] conVT:  0, cumVT: 0
    isCall: false, inFunc: true
[3] conVT:  1, cumVT: 1
    isCall: false, inFunc: true
[4] conVT:  1, cumVT: 1
    isCall: true, inFunc: false

Function: func 1
```

```
== Actual Trace
[0] -
[1] conVT:  0, cumVT: 0
    isCall: false, inFunc: true
[2] conVT:  0, cumVT: 0
    isCall: false, inFunc: true
[3] conVT:  1, cumVT: 1
    isCall: false, inFunc: true
[4] conVT:  1, cumVT: 1
    isCall: true, inFunc: false

Function: func 1
```

Here we extend the previous test to acctually call the function defined. Index 4 is identified as the statement with a function call and the analysis correctly marks this as outside of the block of the previous function. The first pass of the trace was as follows:

```
== First Pass Trace
[0] -
[1] conVT:  0, cumVT: 0, isCall: false, inFunc:  true
[2] conVT:  0, cumVT: 0, isCall: false, inFunc:  true
[3] conVT:  1, cumVT: 1, isCall: false, inFunc:  true
[4] conVT: -1, cumVT: 0, isCall:  true, inFunc: false
```

This is provided to show the analysis correctly marked index 4 as being a function call and reset the `cumVT` as it had left a function block from the previous line.

```
define :foo do
  play 55
  sleep 1
end

play 60
foo
bar

define :bar do
  play 75
  sleep 2
end
```

Multiple Function Detection Test Code

```
== Expected Trace
[0] -
[1] conVT: 0, cumVT: 0,
    isCall: false, inFunc:  true
[2] conVT: 0, cumVT: 0
    isCall: false, inFunc:  true
[3] conVT: 1, cumVT: 1
    isCall: false, inFunc:  true
[4] conVT: 0, cumVT: 0
    isCall: false, inFunc: false
[5] conVT: 1, cumVT: 1
    isCall:  true, inFunc: false
[6] conVT: 2, cumVT: 3
    isCall:  true, inFunc: false
[7] conVT: 0, cumVT: 0
    isCall: false, inFunc:  true
[8] conVT: 0, cumVT: 0
    isCall: false, inFunc:  true
[9] conVT: 2, cumVT: 2
    isCall: false, inFunc:  true

Function: foo 1
          bar 7
```

```
== Actual Trace
[0] -
[1] conVT: 0, cumVT: 0
    isCall: false, inFunc:  true
[2] conVT: 0, cumVT: 0
    isCall: false, inFunc:  true
[3] conVT: 1, cumVT: 1
    isCall: false, inFunc:  true
[4] conVT: 0, cumVT: 0
    isCall: false, inFunc: false
[5] conVT: 1, cumVT: 1
    isCall:  true, inFunc: false
[6] conVT: 2, cumVT: 3
    isCall:  true, inFunc: false
[7] conVT: 0, cumVT: 0
    isCall: false, inFunc:  true
[8] conVT: 0, cumVT: 0
    isCall: false, inFunc:  true
[9] conVT: 2, cumVT: 2
    isCall: false, inFunc:  true

Function: foo 1
          bar 7
```

This demonstrate the accuracy of the function detection, as the trace is able to correctly fill the virtual times whether the function is defined before or after its uses.

```
== First Pass Trace (Differences)
[5] conVT: -1, cumVT: -1, isCall: true, inFunc: false
[6] conVT: -1, cumVT: -2, isCall: true, inFunc: false
```

Nested Functions

```
define :bottom do
  sleep 10
end

define :top do
  sleep 2
  bottom
end

top
bottom
```

Multiple Function Detection Test Code

```
== Expected Trace
[0] -
[1] conVT:  0, cumVT:  0,
    isCall: false, inFunc:  true
[2] conVT: 10, cumVT: 10
    isCall: false, inFunc:  true
[3] conVT:  0, cumVT:  0
    isCall: false, inFunc:  true
[4] conVT:  2, cumVT:  2
    isCall: false, inFunc:  true
[5] conVT: 10, cumVT: 12
    isCall:  true, inFunc:  true
[6] conVT: 12, cumVT: 12
    isCall:  true, inFunc: false
[7] conVT: 10, cumVT: 22
    isCall:  true, inFunc: false

Function: bottom 1
          top 3
```

```
== Actual Trace
[0] -
[1] conVT:  0, cumVT:  0
    isCall: false, inFunc:  true
[2] conVT: 10, cumVT: 10
    isCall: false, inFunc:  true
[3] conVT:  0, cumVT:  0
    isCall: false, inFunc:  true
[4] conVT:  2, cumVT:  2
    isCall: false, inFunc:  true
[5] conVT: 10, cumVT: 12
    isCall:  true, inFunc:  true
[6] conVT: 12, cumVT: 12
    isCall:  true, inFunc: false
[7] conVT: 10, cumVT: 22
    isCall:  true, inFunc: false

   Function: top 1
       bottom 3
```

This demonstrates that both sequential functions and nested functions are handled as required. Given how Sonic Pi's scoping operates, functions will always be defined before they can be called, meaning a nested function can always be given a time when calculating the time of the upper function.

Parameterized Functions

```
define :func do |n|
  play 60
  sleep n
end
```

Parametre Test Code

This feature is not implemented correctly. To explain the flaws in the current iteration we will present only the first pass information and explain why the second pass cannot produce the correct result.

```
== First Pass Trace
[1] conVT:  0, cumVT:  0, isCall: false, inFunc: false
[2] conVT: -1, cumVT: -1, isCall: false, inFunc:  true
[3] conVT:  0, cumVT: -1, isCall: false, inFunc:  true
[4] conVT: -1, cumVT: -2, isCall: false, inFunc:  true
```

In this syntax the first line is processed as having a function call (for the keyword `do`), keycode `-1`. This identifies a problem with the way the analysis marks statements. `-1` is overloaded as the default symbol recognition and a marker as a function call. To be clear when something is unrecognised the default keycode should be changed. In the second pass, index 2 is zero'd and index 4 remains the same. This is due to the fact that some code exists which resets the values for the first method call found. This is useful code for sequences of method calls where the incorrect starting times would sometimes garble the results, but this test highlights two things: it should not actually be run if the function in question doesn't exist (so that the line remains unchanged for clarity after the fact), and it does not reset properly between sequences of functions.

In the future the analysis will need to recognise when the `block` node's `arg` child has children of it's own. This marks when something has been written with parametres and what those parametre symbols are. They can then be stored in the scope of that block similar to how functions are stored at the global level of the trace.

5.2.8 Conditionals

```
if cond then
  sleep 1
else
  sleep 0.5
end
```

Conditional Test Code

```
== Expected Trace                  == Actual Trace
[0] conVT:  1, cumVT:    1         [0] conVT:  1, cumVT:    1
[1] conVT: -3, cumVT:   -1         [1] conVT: -3, cumVT:   -1
[2] conVT: -3, cumVT: -1.5         [2] conVT: -3, cumVT: -1.5
```

Conditionals presented an interesting case while collecting these test results. The `if` keyword actually works similar to the `block` keyword in that they both mark the beginning of some statement list. This means they will always come before a `begin` keyword in the tree. For the most part during a program this does not cause much issue as the processing is still the same. However, in this case, where we have only the conditional in our workspace, the `if` gets taken in as the `RootNode` in our AST. Previously `RootNode` was not involved in the processing as it was assumed there would never be anything but a `begin` or a `block` keyword there. Because of this, the second pass could not accurately clean the trace and our results look like this:

```
== Trace
[0] conVT:  -2, cumVT:   -2
[1] conVT:   1, cumVT:   -1
[2] conVT: 0.5, cumVT: -0.5
```

This has since been fixed, hence the corrected trace shown initially.

Given the nature of our implementation (as described in Section 4), this has the same output regardless of the conditional's expression.

5.2.9 ; Separation

```
play 60 ; sleep 1 ; play 66 ; sleep 0.5
```

Snippet 23: Multiple Statements on Single Line Test Code

```
== Expected Trace                  == Actual Trace
[0] -                              [0] -
[1] conVT:   0, cumVT:   0         [1] conVT:   0, cumVT:   0
[2] conVT:   1, cumVT:   1         [2] conVT:   1, cumVT:   1
[3] conVT:   0, cumVT:   1         [3] conVT:   0, cumVT:   1
[4] conVT: 0.5, cumVT: 1.5         [4] conVT: 0.5, cumVT: 1.5
```

To further demonstrate the ability to process multiple statements on one line, and that the analysis will work whether statements are comma separated or not. The analysis recognises floats and successfully as ints and time advancement is summed correctly throughout the whole trace.

5.2.10 Communications

```
live_loop :foo do
  cue :A
  sleep 1
end
```

Snippet 24: Cue Detection Test Code

```
== Expected Trace
[0] -
[1] conVT: 0, cumVT: 0
[2] conVT: 0, cumVT: 0
[3] conVT: 1, cumVT: 1

== Expected Arc Results
Cue 0 Block:P0 foo
  |_> points at
Cue 1 Block:P0 A
  |_> points at

'' Unclaimed Syncs ''
```

```
== Actual Trace
[0] -
[1] conVT: 0, cumVT: 0
[2] conVT: 0, cumVT: 0
[3] conVT: 1, cumVT: 1

 == Actual Arc Results
 Cue 0 Block:P0 foo
   |_> points at
 Cue 1 Block:P0 A
   |_> points at

 '' Unclaimed Syncs ''
```

`cue` is accurately recorded in the session graph whilst virtual time is also still recorded correctly. `live_loop`s also send out a `cue` operation and this is also present.

```
live_loop :foo do
  sync :A
  sleep 1
end
```

Snippet 25: Sync Detection Test Code

```
== Expected Trace
[0] -
[1] conVT: 0, cumVT: 0
[2] conVT: 0, cumVT: 0
[3] conVT: 1, cumVT: 1

== Expected Arc Results
Cue 0 Block:P0 foo
  |_> points at

'' Unclaimed Syncs ''
= 0 A
```

```
== Actual Trace
[0] -
[1] conVT: 0, cumVT: 0
[2] conVT: 0, cumVT: 0
[3] conVT: 1, cumVT: 1

 == Actual Arc Results
 Cue 0 Block:P0 foo
   |_> points at

 '' Unclaimed Syncs ''
 = 0 A
```

`sync` is recorded properly alongside the correct virtual time results. `sync` is recorded into a different data structure when not pared with a `cue` as it is blocking and will disrupt your performance. The separate data structure is easier to print as an error result if desired.

```
in_thread do
  loop do
    sync :A
    cue :B
    sleep 1
    play 63
  end
end
```

```
in_thread do
  loop do
    cue :A
    sync :B
    play 60
    sleep 0.5
  end
end
```

Snippet 26: Global Type Test Code

```
== Expected Trace
[0] -
[1]  conVT:   0, cumVT:   0
[2]  conVT:   0, cumVT:   0
[3]  conVT:   0, cumVT:   0
[4]  conVT:   0, cumVT:   0
[5]  conVT:   1, cumVT:   1
[6]  conVT:   0, cumVT:   1
[7]  conVT:   0, cumVT:   0
[8]  conVT:   0, cumVT:   0
[9]  conVT:   0, cumVT:   0
[10] conVT:   0, cumVT:   0
[11] conVT:   0, cumVT:   0
[12] conVT: 0.5, cumVT: 0.5

 == Expected Arc Results
 Cue 1 Block:P0 B
    |_> points at
        → P1: 4 B
 Cue 3 Block:P1 A
    |_> points at
        → P0: 0 A

 '' Unclaimed Syncs ''

 == Expected Local Types
 SubGraph P0
   A:(P1P0)?.B:(P0P1)!.time
 SubGraph P1
   A:(P1P0)!.B:(P0P1)?.time

 Global Type: P1→P0.P0→P1
```

```
== Actual Trace
[0] -
[1]  conVT:   0, cumVT:   0
[2]  conVT:   0, cumVT:   0
[3]  conVT:   0, cumVT:   0
[4]  conVT:   0, cumVT:   0
[5]  conVT:   1, cumVT:   1
[6]  conVT:   0, cumVT:   1
[7]  conVT:   0, cumVT:   0
[8]  conVT:   0, cumVT:   0
[9]  conVT:   0, cumVT:   0
[10] conVT:   0, cumVT:   0
[11] conVT:   0, cumVT:   0
[12] conVT: 0.5, cumVT: 0.5

  == Actual Arc Results
  Cue 1 Block:P0 B
     |_> points at
         → P1: 4 B
  Cue 3 Block:P1 A
     |_> points at
         → P0: 0 A

  '' Unclaimed Syncs ''

  == Actual Local Types
  SubGraph P0
    A:(P1P0)?.B:(P0P1)!.time
  SubGraph P1
    A:(P1P0)!.B:(P0P1)?.time

  Global Type: P1→P0.P0→P1
```

This trace shows the virtual time kept during the program, the `cue` operations by index and which `sync` operations they link with. The trace accurately reports no unclaimed `syncs`. After this are the local types of each thread, as named by our analysis, and the global type of the system. All traces shows that the analysis information is correct; no results disagree with one another.

```
in_thread do
  loop do
    sync :A
    cue :B
    sleep 1
    play 63
  end
end
```

```
in_thread do
  loop do
    sync :B
    cue :A
    play 60
    sleep 0.5
  end
end
```

Snippet 27: Deadlock Detection Test Code

```
== Expected Trace
[0] -
[1]  conVT:   0, cumVT:   0
[2]  conVT:   0, cumVT:   0
[3]  conVT:   0, cumVT:   0
[4]  conVT:   0, cumVT:   0
[5]  conVT:   1, cumVT:   1
[6]  conVT:   0, cumVT:   1
[7]  conVT:   0, cumVT:   0
[8]  conVT:   0, cumVT:   0
[9]  conVT:   0, cumVT:   0
[10] conVT:   0, cumVT:   0
[11] conVT:   0, cumVT:   0
[12] conVT: 0.5, cumVT: 0.5

 == Expected Arc Results
 Cue 1 Block:P0 B
    |_> points at
        → P1: 3 B
 Cue 3 Block:P1 A
    |_> points at
        → P0: 0 A

 '' Unclaimed Syncs ''

 == Expected Local Types
 SubGraph P0
   A:(P1P0)?.B:(P0P1)!.time
 SubGraph P1
   B:(P0P1)?.A:(P1P0)!.time
```

```
== Actual Trace
[0] -
[1]  conVT:   0, cumVT:   0
[2]  conVT:   0, cumVT:   0
[3]  conVT:   0, cumVT:   0
[4]  conVT:   0, cumVT:   0
[5]  conVT:   1, cumVT:   1
[6]  conVT:   0, cumVT:   1
[7]  conVT:   0, cumVT:   0
[8]  conVT:   0, cumVT:   0
[9]  conVT:   0, cumVT:   0
[10] conVT:   0, cumVT:   0
[11] conVT:   0, cumVT:   0
[12] conVT: 0.5, cumVT: 0.5

  == Actual Arc Results
  Cue 1 Block:P0 B
     |_> points at
         → P1: 3 B
  Cue 3 Block:P1 A
     |_> points at
         → P0: 0 A

  '' Unclaimed Syncs ''

  == Actual Local Types
  SubGraph P0
    A:(P1P0)?.B:(P0P1)!.time
  SubGraph P1
    B:(P0P1)?.A:(P1P0)!.time
```

```
    Global Type: -                    Global Type: -
```

Similar to previously, this trace shows that the analysis can correctly identify a deadlock situation as the global type of the system as not been printed.

5.2.11 Musical Score

```
load_samples [:drum_heavy_kick, :drum_snare_soft]

define :drums do
  cue :slow_drums
  6.times do
    sample :drum_heavy_kick, rate: 0.8
    sleep 0.5
  end
  cue :fast_drums
  8.times do
    sample :drum_heavy_kick, rate: 0.8
    sleep 0.125
  end
end

define :snare do
  cue :snare
  sample :drum_snare_soft
  sleep 1
end

define :synths do
  puts "how␣does␣it␣feel?"
  use_synth :mod_saw
  use_synth_defaults amp: 0.5, attack: 0, sustain: 1,
                     release: 0.25, cutoff: 90, mod_range: 12,
                     mod_phase: 0.5, mod_invert_wave: 1
  notes = [:F, :C, :D, :D, :G, :C, :D, :D]
  notes.each do |n|
    play note(n, octave: 1)
    play note(n, octave: 2)
    sleep 1
  end
end
```

```
in_thread(name: :synths) do
  sleep 6
  loop{synths}
end

in_thread(name: :drums) do
  loop{drums}
end

in_thread(name: :snare) do
  sleep 12.5
  loop{snare}
end
```

Monday Blues - Coded by Sam Aaron, as found in Sonic Pi IDE Samples

This test result is lengthy but is included here in full, with full source, as it is the best example of the current state of this project. We do not present the second pass trace in full; instead we inline it with the relevant discussion points. This is an accurate example of some piece that may be written for a real live coding performance and contains both complicated nesting structures and multiple function definitions. It does not make full use of the communication primitives, using only the `cue` operation in this test. This seems illogical, as it has nothing to work with, but `cue` does print to the output box of the Sonic Pi IDE, making it another useful performance resource, and a situation the analysis should be able to handle.

```
== First Pass Trace
[0] -
[1]  conVT:    -1, cumVT:     -1, isCall:  true, inFunc: false
[2]  conVT:    -1, cumVT:     -2, isCall:  true, inFunc: false
[3]  conVT:     0, cumVT:      0, isCall: false, inFunc:  true
[4]  conVT:     0, cumVT:      0, isCall: false, inFunc:  true
[5]  conVT:    -1, cumVT:     -1, isCall:  true, inFunc:  true
[6]  conVT:     0, cumVT:     -1, isCall: false, inFunc: false
[7]  conVT:   0.5, cumVT:   -1.5, isCall: false, inFunc:  true
[8]  conVT:     0, cumVT:   -1.5, isCall: false, inFunc: false
[9]  conVT:     0, cumVT:   -1.5, isCall:  true, inFunc: false
[10] conVT:     0, cumVT:   -1.5, isCall:  true, inFunc: false
[11] conVT: 0.125, cumVT: -1.375, isCall: false, inFunc: false
[12] conVT:     0, cumVT:      0, isCall: false, inFunc:  true
[13] conVT:     0, cumVT:      0, isCall: false, inFunc:  true
[14] conVT:     0, cumVT:      0, isCall: false, inFunc:  true
[15] conVT:     1, cumVT:      1, isCall: false, inFunc: false
[16] conVT:     0, cumVT:      0, isCall: false, inFunc:  true
[17] conVT:    -1, cumVT:     -1, isCall:  true, inFunc:  true
[18] conVT:    -1, cumVT:     -2, isCall:  true, inFunc:  true
[19] conVT:    -1, cumVT:     -3, isCall:  true, inFunc:  true
```

```
[20] conVT:    -1, cumVT:     -4, isCall:  true, inFunc:  true
[21] conVT:     0, cumVT:      0, isCall: false, inFunc:  true
[22] conVT:     0, cumVT:      0, isCall:  true, inFunc:  true
[23] conVT:     0, cumVT:      0, isCall: false, inFunc:  true
[24] conVT:     0, cumVT:      0, isCall:  true, inFunc:  true
[25] conVT:     1, cumVT:      1, isCall: false, inFunc:  true
[26] conVT:    -1, cumVT:      0, isCall:  true, inFunc:  true
[27] conVT:     6, cumVT:     -6, isCall: false, inFunc:  true
[28] conVT:     0, cumVT:      0, isCall: false, inFunc:  true
[29] conVT:    -1, cumVT:     -1, isCall:  true, inFunc:  true
[30] conVT:    -1, cumVT:     -2, isCall:  true, inFunc:  true
[31] conVT:     0, cumVT:      0, isCall: false, inFunc:  true
[32] conVT:    -1, cumVT:     -1, isCall:  true, inFunc:  true
[33] conVT:    -1, cumVT:     -2, isCall:  true, inFunc:  true
[34] conVT:  12.5, cumVT:   10.5, isCall: false, inFunc:  true
[35] conVT:     0, cumVT:      0, isCall: false, inFunc:  true
[36] conVT:    -1, cumVT:     -1, isCall:  true, inFunc:  true

Function: drums 3
          snare 12
          synths 16

  == Arc Results
  Cue 0 Block:P0 slow_drums
     |_> points at
  Cue 1 Block:P1 fast_drums
     |_> points at
  Cue 3 Block:P2 snare
     |_> points at

  '' Unclaimed Syncs ''

  == Local Types
  SubGraph P0
    slow_drums:()!
  SubGraph P1
    time.fast_drums:()!
  SubGraph P2
    time.snare:()!.time
  SubGraph P3
    time


  Global Type: -
```

The main failing with this piece is the number of syntax structures that were not previously considered when first working with the program. The analysis marks many statements as functions (`puts`, `sample`, `use_synth`) but has no record of them in the function definitions list.

Given this in the second pass it updates most of these records with the incorrect virtual time contributions and can't calculate the accumulated amounts correctly. The analysis does detect each individual use of `sleep` correctly but total running time of this program would not be possible until the issues involing parameterized loops and functions were solved.

In communication terms, the analysis can detect the use of `cue` and `sync` but has highlighted a problem with their use in functions. The graph is not splitting across the blocks correctly, suggesting nested loops do not build correctly, and the graph does not distinguish between block types. This highlights that the session type analysis should also build a function list in the same way as the timing effects system did. Then, when the function is used in a process the nodes can be assigned to the correct process `SubGraph`.

5.3 Visual Adaptation

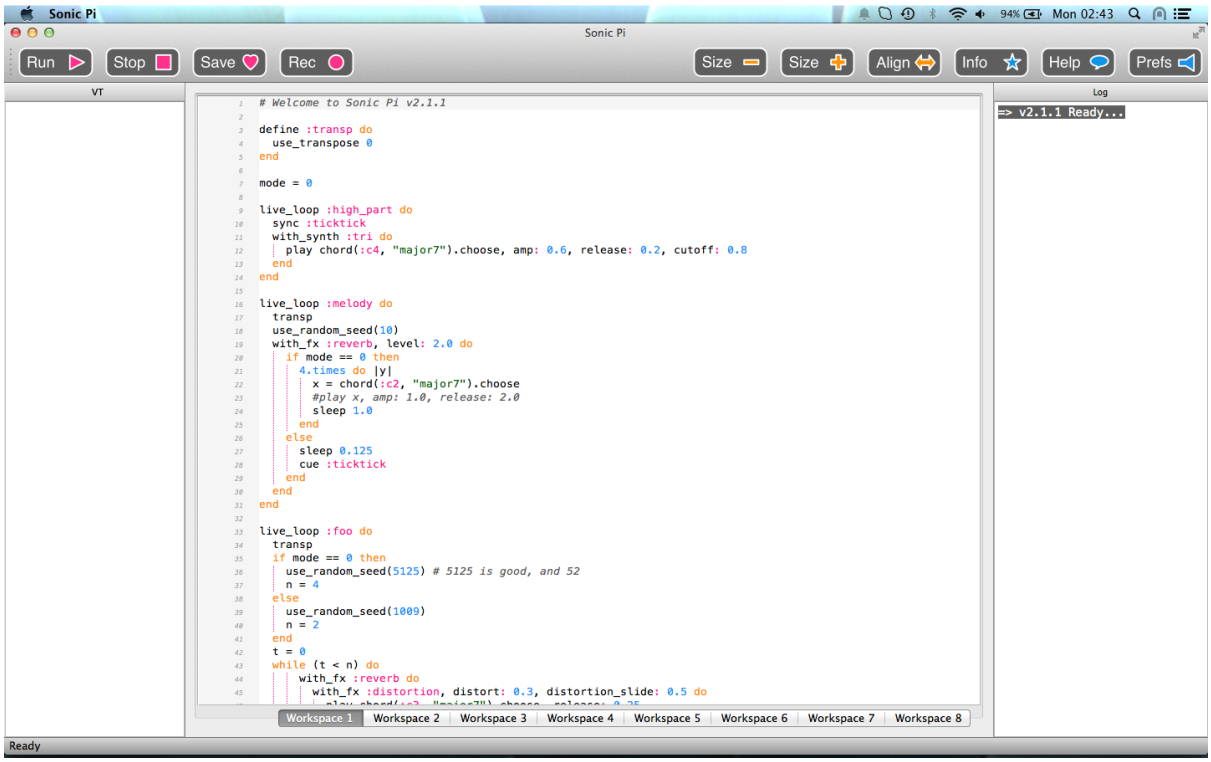


Figure 13: Sonic Pi V2.1.1 IDE Extension - Apple Mac View

The visual differences to the IDE are quite minimal. One notable feature that is currently missing is the ability not to load this library whilst using the IDE, a useful switch to have as not all users will want to see this information all of the time.

For the most part test users relayed that they found the virtual time bar to the left to be a handy little tool. Some noted that the printed session types in the output log were well placed but did not have enough experience with the material to be able to get the most use out of the information.

One interesting piece of feedback was the lack of ability to see the virtual time of your program before running it for the first time. At the moment most of the analysis only runs when playing the piece, so there is no trace to display until that point. It may be that in the future the integration with the library is improved such that, similarly to how a `live_loop` can update the synth mid-performance, the IDE can detect a change in the workspace and run the analysis on the new source. This would require a couple of improvements to the analysis itself; currently the analysis runs afresh each time the code is processed. In the future it would be preferable to hold a trace object for each workspace and to dynamically update this during the lifetime of the IDE window.

6 Conclusion

In this chapter we bring the project to a close by first discussing various improvements for the project, and other features that we might implement in the future, before concluding with a summary of achievements.

6.1 Future Work

Function Parametres

In its current state, the project is able to detect the use of functions and accurately update the current program state to show the current virtual time at a given point. That said, it is not able to do so with variable amounts of sleep. When a function is called with an argument that affects the amount of time the process must sleep for, the project cannot detect the argument passed into the function and apply this where necessary.

Function Calls for Sessions

As noted in Section 4.2.1, the Session Type analysis of this project is unable to handle function calls within processes. Currently this limitation extends to the fact the analysis will consider function definitions as their own processes. In the future this project will benefit from employing similar techniques when building the graph structure as when building the timing trace. One option to explore is the idea that the graph collect lists of nodes as defined in the functions and insert these into the process graph when the functions are called.

Branching Session Types

This iteration of the project cannot accurately handle branching statements within a program. There is some interesting theory to be considered in this improvement as labels will not be passed between processes as standard Session Types may expect them to be. Instead it is more likely that a given if-statement will be considered as an unlabelled branching type and its condition can be marked as the dual selection type repesenting all possible label choices. The subsequent global type for this statement appears as an internal message from the process to itself; this is not an accurate application of branching types but suits the given situation neatly.

Detailed Time Nodes

In the current iteration of the session graph, the passage of time is simply denoted by an empty `GraphNode` with no extra information. In most cases this works well but in the event that there are two processes, `P0` and `P1`, it may be that `P0` processes two passages of time of equal length two one passage in `P1`. In this case the current node of `P1` will be testing itself against the wrong position in `P0` leading to an inaccurate type analysis. In the future we could investigate incorporating more of the information calculated in the timing affects portion of the program into the construction of the session graph.

Minecraft API

Since starting this project, the Sonic Pi IDE has added a section detailing how to use Sonic Pi alongside the Minecraft API. Our analysis is currently untested against this API and it may prove interesting in the future to adapt the library to be able to run alongside this API.

6.2 Achievement Summary

We have produced a lightweight shared library that can statically analyse a Sonic Pi program. We have made a tool that can assist the developer with both temporal and concurrent reasoning of their Sonic Pi programs. The tool will output information on both the state of virtual time throughout the source, and also output information about the communication structure of the program. It can successfully report when the program is deadlocked. We also contribute two new typing ideas for multi-party session types with our `&&` and `||` types, which handle replication type and broadcasting situations within Sonic Pi.

This library can integrate successfully with the existing Sonic Pi IDE and we have also provided a small systematic testing environment within the project to prove the correctness of our analysis and demonstrate the current limitations of the project. With the current set of features implemented, the library is able to accurately run during live perforamnces with very simple musical pieces. Working with fuller pieces, like those presented in the Evaluation, is also very close to being a reality.

References

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

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

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

[4] *http://news.microsoft.com/apac/2015/03/23/three-out-of-four-students-in-asia-pacific-want-coding-as-a-core-subject-in-school-reveals-microsoft-study/*

[5] *http://www.curse.com/mc-mods/minecraft?filter-project-game-version=* (Visited on 14/6/2015)

[6] *https://www.amberbit.com/blog/2014/6/12/calling-c-cpp-from-ruby/*

[7] 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)

[8] 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)

[9] 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)

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

[11] 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)

[12] 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)

[13] 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)

[14] 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)

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

[16] Department for Education, *National curriculum in England: computing programmes of*

study (key stages 1 - 4), (2013)

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

[18] 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)

[19] 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)

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

[21] 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)

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

[23] 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)

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

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

[26] Mostrous, D., Yoshida, N., *Session typing and asynchronous subtyping for the higher-order pi-calculus*, INFORMATION AND COMPUTATION, Vol: 241, Pages: 227-263, (2015)

[27] Mostrous, D., Yoshida, N., Honda, K., *Global principal typing in partially commutative asynchronous sessions*, in: ESOP09, volume 5502 of LNCS, Springer-Verlag, pp. 316332, (2009)

[28] 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)

[29] Lee, I., Davidson, S., and Wolfe, V., *Motivating Time as a First Class Entity*, Technical Reports (CIS), pp. 288, (1987)

[30] Short, D., *Teaching Scientic Conceptsusing a Virtual World - Minecraft*, Teaching Science, Volume 58, No. 3, pp. 55-58, (2012)

[31] Smeaton, D., *Minecraft As A Teaching Tool - A Statistical Study of Teachers' Experience Using Minecraft In The Classroom*, 7267EDN Research Methods in Education, Griffith

University, Austrailia, (2012)

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

[33] Thomasian, A., *Two-phase Locking Performance and Its Thrashing Behaviour*, Performance of Concurrency Control Mechanisms in Centralized Database Systems Prentice-Hall, Inc., Upper Saddle River, NJ, USA, pp. 166-214, (1995)

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

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

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

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