UNIVERSITY OF CALIFORNIA

Santa Barbara

Vessel:

A Platform for Computer Music Composition,

Interleaving Sample-Accurate Synthesis and Control

A Thesis submitted in partial satisfaction of the requirements for the degree

Master of Arts

in

Media Arts and Technology

by

Graham David Wakefield

Committee in charge:

Professor Curtis Roads, Chair

Professor JoAnn Kuchera-Morin

Professor Marcos Novak

June 2007

The thesis of Graham David Wakefield is approved.

JoAnn Kuchera-Morin

Marcos Novak

Curtis Roads, Committee Chair

June 2007

Vessel:

A Platform for Computer Music Composition,

Interleaving Sample-Accurate Synthesis and Control

by

Graham David Wakefield

ACKNOWLEDGEMENTS

I would like to thank the members of my committee for their support, patience and the many ways in which they have inspired me.

I would also like to thank:

Lance Putnam, for both the sharing of and support for his Synz audio library, and synthesis programming in general,

Wesley Smith for inspiration and support both creatively and technically regarding 3-D graphics, and for the OpenGL bindings to Lua,

The MAT GLV, CREATE CSL and IGERT Mint student researchers (including Lance Putnam, Wesley Smith, Eric Newman, Rama Holtzein, Alex Norman, and the author) for resources in the in the windowing/GUI implementation,

Stephen Pope and Xavier Amatriain for instruction in software synthesis and event scheduling,

My friends, family and loved ones for endless support, encouragement and patience.

Partial support provided by NSF IGERT Grant #DGE-0221713.

ABSTRACT

Vessel: A Platform for Computer Music Composition, Interleaving Sample-Accurate Synthesis and Control

Graham David Wakefield

The rich new terrains offered by computer music invite the exploration of new techniques to compose within them. The computational nature of the medium has suggested algorithmic approaches to composition in the form of generative musical structure at the note level and above, and audio signal processing at the level of individual samples. In the region between these levels, the domain of microsound, we may wish to investigate the musical potential of sonic particles that interrelate both signal processing and generative structure. In this thesis I present a software platform ('Vessel') for the exploration of such potential. In particular, a solution to the efficient scheduling of interleaved sound synthesis and algorithmic control with sample accuracy is expounded. The formal foundations, design and implementation are described, the project is contrasted with existing work, and avenues for musical application and future exploration are proposed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

The rich new terrains offered by computer music invite the exploration of new techniques to compose within them. The computational nature of the medium has suggested algorithmic approaches to composition in the form of generative musical structure at the note level and above, and audio signal processing at the level of individual samples. The region between these levels, the domain of microsound [43], holds special interest due to the potential of sonic events to finely interrelate both signal processing and generative structure.

This thesis defends the position that algorithmic exploration of microsound calls for the dynamic yet deterministic interleaving of both signal processing and structural control with up to sample accuracy. Satisfying this demand poses a challenge for both the (outside-time) representation and (in-time) rendering of computer music compositions. This thesis presents a software solution ('Vessel') to this challenge. For representation, it comprises an interpreted music programming language with extensions for event, control and synthesis articulation, while for rendering, it comprises a deterministic, dynamic, lazy scheduling algorithm for both concurrent control logic and signal processing graphs.

Chapter 2 comparatively places the work in relation to existing languages and frameworks. Many key observations related to the thesis question, and the subsequent implementation, are drawn.

Chapter 3 summarizes the requirements of the thesis project, and the implementation of the scheduler and language extensions are expounded in detail. The conceptual model can be summarized as follows: A composition represented as a script file may be evaluated in real-time into hierarchies of dynamically interleaved concurrent processes and relatively outside-time structures. The processes themselves are iteratively interpreted over discrete time, producing an in-time performance or form as digitally produced sound.

Chapter 4 describes the Vessel language and applications in detail, including a number of example scripts for evaluating its ability to satisfy the requirements.

Chapter 5 reviews the conclusions of the thesis and outlines directions for future work.

1.1 Motivations & Significance

> "In the 1950s, certain composers began to turn their attention toward the composition of sound material itself. In effect, they extended what had always been true at the phrase level down to the sound object level. Just as every phrase and macro form can be unique, each sound event can have an individual morphology. This creates a greater degree of diversity – of heterogeneity in

sound material – without necessarily losing continuity to other objects… we can extend the concept of heterogeneity even further, down to the level of microsound, where each sound particle may be unique. The microstructure of any sound can be decomposed and rearranged, turning it into a unique sound object." [43].

Throughout millennia we have invented tools to overcome our physical and mental limitations, and the domain of music production is no exception. The digital computer in particular has redefined what a musical tool can be, offering computational potential with tremendous generality applicable to many different musical activities. Of special interest to the composer are the use of computational facilities for the synthesis of otherwise inaccessible sounds and the articulation of complex musical structures.

The synthesis of music according to a set of rules is often referred to as *algorithmic music*, though strictly speaking the term *generative music* should be applied if the rules are expressed in a form that can be processed by the computer [42]. One might object that, at a certain level, all music produced by computers is algorithmic, since all computer activity is rule-based; to be useful, the distinction of algorithmic computer music indicates that the rules are stipulated and/or meaningfully applied by the composer, rather than the system architect.

Algorithms and rules may be used to stipulate musical structure with different relationships to time. Synthesis and signal-processing algorithms may be specified

by the composer to generate sonic textures at the level of discrete samples or sample-streams, while musical form and structure across larger temporal ranges may be derived according to mathematical functions, categorizations and relationships of set theories, operations of flavors of formal logic, procedural instructions, and so on. In addition, the composer may require algorithms to be dynamic over the duration of the composition, whether according to pre-set configurations, or by deriving them generatively at run-time.

The primary motivation therefore for enabling fine algorithmic control of synthesis and musical structure is to augment the vocabulary and enrich the nuances of computer music composition. Trevor Wishart for example highlights the need for "precise sound-compositional control of the multi-dimensional space" in order to achieve "a subtly articulated and possibly progressively time-varying 'playing' of the sound space." [54]. Curtis Roads meanwhile emphasizes that the exploration of the microsonic time-scale presents exciting creative opportunities for the computer musician: "Microsonic particles remained invisible for centuries. Recent technological advances let us probe and explore the beauties of this formerly unseen world. Microsonic techniques dissolve the rigid blocks of music architecture – the notes – into a more fluid and supple medium." [43]. The fine interleaving of control and synthesis in musical expression is also a key component of the author's own artistic endeavors [47].

Today both signal processing [42] and algorithmic form [31] benefit from extensive research and musical use, however as we shall see in Chapter 2, many

contemporary tools for computer music composition limit the combined exploration of these techniques dynamically and at micro time-scales. This thesis describes the development of software tools to specifically support such compositional explorations.

1.2 Some suggested Applications

Some specific examples of potential avenues of musical exploration enabled by this thesis project are described below. It should be noted that any and all of these techniques can be combined with each other and with more established approaches computer music composition. Since what is possible at the micro-level may be extrapolated to larger time-scales, much of the potential can be extended to computer music composition in a more general sense.

1.2.1 Granular synthesis

There are many different forms of granular synthesis [43], however in general all use some combination of event scheduling for multiple voices, each of which has a small signal processing routine. Geiger notes that granular synthesis "needs a high level of dynamic instantiation with a exact time granularity and speed," which cannot be transparently supported by many computer music systems, thus "Most of the systems therefore integrate granular synthesis as a separate unit generator" [15]. The model described in this thesis overcomes these issues (as described in later

Chapters) and thus is not only capable of many known of granular synthesis both in terms of signal processing and event distribution, but is also ideal for the investigation of new granular approaches.

1.2.2 Synthesis with timbral complexity

Xenakis compared the stationary steady state of synthesized sounds to the tiny variations and fluctuations evident in acoustic sources, and called for "new theories of approach, using another functional basis and harmonic analysis on a higher level, e.g., stochastic processes, Markov chains, correlated or auto-correlated relations, or theses of pattern and form" [55]. Finely interleaving control logic and synthesis introduces scope for quite non-linear methods of micro-variation within sound timbre. Applied to FOF synthesis for example [42], one could embed the indeterminacies and fluctuating behavior of the vocal tract into the generative algorithm rather than applying it as a global control.

1.2.3 Musical micro-agents

Algorithmic composition based upon parallel procedural flow can support techniques based upon Finite State Automata (FSA) on a per-particle basis. Such techniques might include Markov chains, formal grammars, L-systems, cellular automata, artificial life, flocking and any agent-based algorithmic models [31].

Structured rather than stochastic determination of acoustic grain properties is an exciting field to explore within music composition.

1.2.4 Atomic physical modeling

Physical modeling is an established technique for sound synthesis [42] as well as high-level musical control [19]. The extension of physical modeling techniques to micro-sonic particle simulations could include interacting sonic atoms with inter-particle and surface collisions, under the action attractive and/or repulsive fields. One can imagine a model of a corpuscular scattering of an impulse, as a granular approximation of reverberation for example. What differentiates such an approach principally from typical stochastic approaches to granular synthesis is the insertion of memory and pattern coherence.

1.2.5 Strategy variation and top-down articulation

Roads notes that signal processing suggests a parametrical model that need not be exclusively adhered to: "Alternatively, the compositional strategy itself may be the subject of variations... Juxtaposition refreshes the brain, breaking the cycle of closed permutations and combinations." [43]

Besides enriching synthesis from the bottom-up, a high-level musical programming language can support top-down determination of musical structure. For these purposes, generic language features such as math and text processing can

be used to generate, model and evaluate semantic grammars, narrative system dynamics, and fuzzy and temporal logic. Temporal scripting is particularly well suited to responsive real-time systems and simulation systems (for further discussion see the Tempo language of Dami et al. [9].) Goal-oriented coroutine programming could be used to dynamically evaluate pattern logics, constraint systems or elaboration graphs [29].

1.3 Key concepts & terms

Before proceeding, it is prudent to clarify some of the key concepts and terms used in this thesis.

1.3.1 Formal representations of computer music composition

Computer music composition encompasses humans interfacing with computers in order to write compositions in such a way that objective sonic output (in the form of digital samples) becomes possible. To stipulate musical structure, it must be represented in a form that can be expressively written and read by the human while also precisely parsed by the computer. It is a role of the software developer to provide a good bridge between human users and computer-parsable formal representations, and to maximize the capabilities of these formal representations by finding an appropriate substrate in which to work while minimizing the tradeoffs of expressive flexibility against efficiency.

Unfortunately, any definition of artistic ideas must necessarily be open-ended, and the domain of music composition is fraught with complexity and ambiguity [11]. In summary, a desired musical structure may be a tangled conceptual hierarchy that interleaves with variable dependencies. It may be described in sequential and parallel terms, include complex, dynamic, homogeneous or heterogeneous strata of containment, behavior and relationships. A good formal representation of computer music should support all of these possibilities.

Varèse described music as 'organized sounds' [51]. The noun term 'sounds' suggests that music results in objective forms, whilst the past-tense term 'organized' indicates that music is the result of a process of organization. We can thus identify two extremes in the representation of musical output, the form of the musical output of a composition, and the process by which a composition is constructed. Likewise, Roads here echoes observations by Xenakis: "Musicologists have long argued whether, for example, a fugue is a template (form) or a process of variation. This debate echoes an ancient philosophical discourse pitting form against flux, dating back as far as the Greek philosopher Heraclitus. Ultimately, the dichotomy between form and process is an illusion, a failure of language to bind two aspects of the same concept into a unit… A form is constructed according to a set of relationships. A set of relationships implies a process of evaluation that results in a form." [43]. Similarly, we can identify the formal representations of compositions (the composition file, script, project files etc.), as well as the process by which the output is generated (the algorithms specified in the composition and the software substrate

in which it is rendered). The author therefore suggests that a formalized model of musical process to be a suitable basic grounding for algorithmic computer music composition.

1.3.2 Unit generators and block rate processing

At the lowest level, computer music involves the procedural determination of a series of samples to be rendered as sound by a digital to audio converter. Audio signal processing comprises mathematical functions used to synthesize output sample values in response to input samples or the passage of time. The unit generator model is archetypal: "One of the most significant developments in the design of digital sound synthesis languages was the concept of unit generators. Unit generators are signal processing modules … which can be interconnected to form synthesis instruments or patches that generate sound signals." [42]. Unit generators encapsulate mathematical functions and context variables into processing nodes, which can be connected via arcs into directed graphs suitable for real-time signal processing (DSP graphs). Unit generators offer flexibility due to their modularity: the same unit generator types can be reconfigured into different graph structures, resulting in wholly different sonic output. The flexibility and generality of the unit generator model is ideal for exploratory computer music composition.

1.3.3 Block rate and control rate

Audio signal processing involves mathematical operations on large quantities of data, and is therefore expensive computationally. A solution to improve efficiency, supported by most unit generator models, is to operate upon 'blocks' of audio data at a time. The loading of contextual data for a unit generator operation does not need to occur at each sample (improving cache performance), and operations upon chunks of data can make use of SIMD instructions in the CPU. Block sizes are typically between 32 and 256 samples, corresponding to durations between 7.25 and 58 milliseconds respectively. Control (rather than signal) parameters to unit generators are updated at this block rate to avoid disrupting the block-processing efficiency, and likewise structural changes to the synthesis graph must occur at the boundaries of the block, resulting in the notion of a 'control rate'.

However, in order to freely explore regions of the micro time-scale, control and graph changes may be required between these block boundaries, perhaps as finely delimited as a single sample. The block-rate quantized control also limits the capability to relate synthesis and control logic with temporal accuracy at the micro scale. It should be noted that control-rate has no musical significance, but is made apparent to the user in order to manage computational efficiency.

2 Related Work & Observations

The contemporary computer music composer can choose from a plethora of software tools, however for the purposes of this thesis the author will consider four of the most prevalently used and relevant to the thesis question: the Max family (Max/MSP, PD, jMax), CSound, SuperCollider, and ChucK. The capabilities of each of these systems regarding the dynamic interleaving of both signal processing and structural control are evaluated, and in each case, observations applicable to the design and development of the Vessel system are made clear.

2.1 Max

For the purposes of this thesis, Max refers to Max/MSP [56], PD [38] and related software. Max is a popular choice for composing interactive digital media works because of the approachable graphical interface, extensive bindings to media processes and protocols, and the open-ended philosophy. The Max family implements a Data Flow Architecture [28] for both synthesis and message scheduling, defined in a visual a patching interface in which audio is processed in stream flow and other data types are processed in event flow.

Puckette emphasizes that "Max is fundamentally a system for scheduling real-time tasks and managing communication among them." [39], and as such can be ideal for the complex interleaving of synthesis and control. In addition, the Max

interface may ease the learning curve required to use it and thus support exploratory composition, as McCartney notes: “Max … provides an interesting set of abstractions that enable many people to use it without realizing they are programming at all. “ [30].

For synthesis, Max provides an extensive library of unit generator modules that can be patched together quite freely by the composer. Max uses a control-rate for efficiency, though the user has some control over its relationship to the message processing and priorities. Subsections of signal graphs can operate with different control rates using the *poly~* container, though the interface can be cumbersome.

2.1.1 Graphical and textual representations for composition

While a graphical patching interface may facilitate rapid sketching through an intuitive representation, it also carries some inherent limitations for algorithmic composition of microsound. Since each processing unit must be visually represented, the process graph becomes somewhat static and struggles to represent large numbers of processors, especially with minor variations. It is difficult to dynamically change the process graph during performance, particularly with accuracy regarding timing. Expressive data structures, variable scoping and in particular procedural control flow can be difficult to express visually.

Many of these limitations do not apply to textual interfaces. For example, Puckette notes that procedural approaches to control are "better undertaken within

the tasks than by building networks of existing tasks. This can be done by writing "externs" in C, or by importing entire interpreters…" [39]. McCartney notes that Max's visual representation "is also limited in its ability to treat its own objects as data, which makes for a static object structure." [30].

A general comparison of the advantages and disadvantages of graphical and textual interfaces are presented in Table 1.

	Graphical user interface		Textual language interface
+	User-input may be constrained to logically valid operations	-	Steeper learning curve of syntax and vocabulary
+	Easier to view and input quantitatively rich data such as control envelopes	-	Tiresome to specify by data-entry when precision is not required
+	Common tasks can be immediately and intuitively represented	-	Simple tasks may require detailed code
+	Interaction can be more rapid	-	Interaction can be time-consuming, particularly if text must be compiled
-	Interfaces tend to be more specific	+	Interface is highly generic
-	Complex data-structures, if made visible, can be visually overwhelming	+	Compact description of complex data-structures
-	Precise qualitative specification can be difficult at fine granularity	+	High degree of precision & control
-	Visual representations usually demand rigid models	+	Textual elements may more easily refer to or embed each other

Table 1: Relative merits of graphical and textual representations of computer music.

2.2 CSound

CSound is one of the better-known textual interfaces for computer music composition. CSound was originally written by Barry Vercoe at MIT in 1985, based upon earlier languages of the Music-N family, and continues to be developed today (advancing to version 5.0 in February of 2005). At its core, CSound is "designed around the notion that the composer creates a synthesis orchestra and a score that references the orchestra." [42]. The orchestra and score are specified textually using distinct syntaxes (Figure 1).

```
<CsoundSynthesizer>;

  <CsOptions>
    csound -W -d -o tone.wav
  </CsOptions>

  <CsInstruments>
    sr     = 44100           ; Sample rate.
    kr     = 4410            ; Control signal rate.
    ksmps  = 10              ; Samples pr. control signal.
    nchnls = 1               ; Number of output channels.

    instr 1
    a1     oscil p4, p5, 1   ; Simple oscillator.
           out a1            ; Output.
    endin
  </CsInstruments>

  <CsScore>
    f1 0 8192 10 1           ; Table containing a sine wave.
    i1 0 1 20000 1000        ; Play one second of one kHz tone.
    e
  </CsScore>

</CsoundSynthesizer>
```

Figure 1: Basic Csound XML file with orchestra and score sections.

Csound files were originally processed in non real-time to render sonic output, in a "process referred to as 'sound rendering' as analogous to the process of 'image rendering' in the world of computer graphics." [6]. Csound instruments are defined in the orchestra file as directed graphs of unit generator types (called 'opcodes' in CSound). Flexible sound routing can be achieved using control and audio busses via the Zak objects. Control rate is evident in CSound through the *a-rate* and *k-rate* notations.

2.2.1 Distinction of synthesis and temporal form

Since 1990, Csound has provided real-time rendering [52], and today various implementations support interaction input such as graphical interfaces, VST controls, MIDI, OpenSoundControl etc. Nevertheless, the set of instruments must be defined in advance of performance, thus any generative structures desired must be imposed externally according to this vocabulary.

The strong separation of synthesis and temporal event definition imposes a strict limitation on the scope for algorithmic composition: new synthesis processes cannot be defined in response to temporal events, and new temporal events cannot occur in response to the synthesis output. "Csound is very powerful for certain tasks (sound synthesis) while not particularly suited to others (data management and manipulation, etc.)." [5].

2.2.2 Declarative and procedural languages

The Csound score and orchestra languages are essentially declarative series of statements, with almost no provision for procedural control (such as expressions and control flow). Roads identifies two key benefits of procedural representations of musical flow: "First, the compositional logic is made explicit, creating a system with a degree of formal consistency. Second, rather than abdicating decision-making to the computer, composers can use procedures to extend control over many more processes than they could manage with manual techniques." [42].

The CSound orchestra language does support rudimentary procedural control flow using *goto/label*, Boolean conditions with *if/goto*, and temporal pseudo-subroutines via *reinit/rireturn/timout* and *ihold/turnoff*, however the use of such features for algorithmic composition is not trivial. For example, Eugenio Giordani uses the *timout* function in order to generate individual grain events within a granular synthesis instrument definition, yet it is clear in the implementation that this was far from straightforward to achieve [5]. The Csound score language does not support any kind of programmatic control flow suitable for algorithmic composition[1].

In contrast to declarative languages, procedural languages "generate musical events by stipulated procedures or rules… Procedural composition languages go beyond the representation of traditional scores to support the unique possibilities of computer music. These languages let composers specify music algorithmically."

---

[1] The carry and tempo stretching operators are for pre-processing only.

[42]. There are many features in procedural languages with potential applications to algorithmic music that are unavailable to declarative languages, as summarized in Table 2. Generic procedural programs can be written in C and executed using Cscore to create score files generatively.

Procedural language feature	Applications in algorithmic music
Flexible data-description (Variable data-types, homogeneous and heterogeneous containment, object hierarchies)	Musical signal representation, categorization and set theoretic operations, behavioral encapsulation
Mathematical functions & logical functions	Mathematically and logically specified rules
Procedural control flow	Branching, looping, parallelism and nesting of functional activity, compact representation through code re-use
Extensibility (static and dynamic binding)	Generically connect to other processes, e.g. scientific library routines or graphical rendering

Table 2: Features of procedural languages with applications in algorithmic music.

2.2.3 High-level interpreted languages

Writing generative programs in C requires low-level programming skills not necessarily appropriate for the computer music composer. In contrast, high-level interpreted programming languages such as Lua, Python, Ruby and Scheme are increasingly popular due to more approachable syntaxes. In addition, interpreted music languages can be modified and executed immediately while the program is

running, without needing to go through a distracting compilation stage. The composer engaged in musical experimentation may appreciate a shorter programming-testing loop in a real-time system. Interpreted languages can also support advanced techniques such as run-time code generation, which may offer unique potential for algorithmic composition.

Recently CSound has added bindings to Python, a high-level interpreted programming language. Scripts in Python can generate Csound orchestra/score files and instantiate Csound renderers to interpret them into sound, while utilizing the powerful data description, control flow and extension library capabilities that such high-level languages provide. There remains however a functional and temporal separation between the generation of orchestra/score and the rendering thereof. Conversely, Csound can interpret Python code embedded within an orchestra file, supporting more powerful generative synthesis techniques. This Python interpreter can also be used in real-time rendering, however the Python opcodes are limited to control rate execution, and Python is not optimized for high-priority real-time execution. Though it is possible to have sample-accurate Python calls within the synthesis rendering by setting the control-rate equal to the sample-rate, the CPU cost is likely to be prohibitive for most real-time applications.

2.3 SuperCollider

SuperCollider [30] is a high-level interpreted programming music language designed specifically for dynamic and generative structures and synthesis of computer music. It can be generally applied to many different approaches to composition and improvisation rather than any particular preconceived model. It features an application-specific high-level programming language SCLang (drawing inspiration from C++ and Smalltalk) with extensive data-description and functional programming capabilities, and support functions for common musical needs. SuperCollider also features an extensive library of unit generators for signal processing. Sample-rate and control-rate distinctions are made explicit via the *.ar* and *.kr* notation. A key distinction from CSound is that code can be evaluated in real-time as the program runs.

SuperCollider is ideal for the exploration of algorithmic composition. Since version 3.0 (the currently available version), graphs of unit generators are defined textually and compiled at run-time into dynamic libraries (‘*SynthDefs’*) to be loaded as instruments (‘*synths’*) by the synthesis engine (‘*SCServer’*), all under control of the language. The language and synthesis engine run as different processes or applications that communicate using socket messaging.

2.3.1 Latency in the procedural control of synthesis

The separation of language and synthesis into distinct processes in version 3.0 introduces compilation and performance optimizations, but also implies limitations in the degree of temporal control: “Because instruments are compiled into code, it is not possible to generate patches programmatically at the time of the event as one could in SC2... In SC2, an audio trigger could suspend the signal processing, run some composition code, and then resume signal processing. In SC Server, messaging between the engines causes a certain amount of latency.” [30]. While for most purposes this latency is not noticeable, in the micro time-domain it can be devastating.

An additional consequence of the separation is that the expressive functional language of SCLang is not available within synthesis instrument definitions. SuperCollider 3.0 therefore represents a slight return to the CSound model of orchestra and score, in which however the score is procedural rather than declarative.

2.3.2 Concurrency and musical flow

One of the motivations behind the design of SuperCollider was the support for the representation of musical structure using high-level data descriptions of concurrent musical flow [30]. Compositions with sequential and parallel symmetries can be represented more succinctly and structurally in functional terms than as flat

lists of elements[2]. Herein lies a benefit: increasing apparent functional structure also increases expressive affordances for transformation, including algorithmic composition. Smoliar for example considers a procedural representation of musical flow as an interaction of multiple processes in order to develop a language (‘*Euterpe’*) for the algorithmic analysis of musical structure [48]. We might also observe that real performers do not react linearly to each elementary advancement of time but rather multi-task: they maintain meter, prepare for imminent gestures, scan ahead in the score, and so on.

Software development encounters a similar problem: in the limit, computation according to the finite state machine model[3] is in fact a singular procedural progression, but this is not a natural way to think about the design of interactive software. High level programming languages present interfaces with greater affordances by making use of apparently concurrent constructs such as subroutines, threads and coroutines. Many digital composition tools also embed parallel structure as multiple timelines (e.g. Adobe Flash). SCLang provides excellent support for concurrent processes and musical flow using for example the *Routine* and *Task* data types.

---

[2] Just as redundant information can be compacted by Huffman coding.

[3] For the purposes of this thesis, we disregard parallel CPU architectures; the justification will become clear later in the section describing the scheduling implementation and threading.

2.4 ChucK

ChucK [53] represents one of the only contemporary options that avoids latency in the procedural control of synthesis. ChucK is a concurrent, dynamic programming language designed for run-time programming in mind. It also provides a library of unit generators (largely based on the STK library) to be freely instantiated and connected into graphs within ChucK scripts. The authors refer to ChucK as 'strongly timed', which can be defined as follows:

- Supports sample accurate events,
- Defines no-control-rate (or supports dynamically arbitrary control-rates),
- Supports concurrent functional logic,
- Control logic can be placed at any granularity relative to synthesis,
- Supports run-time interaction and script execution.

Like SuperCollider's SCLang, the ChucK language was written especially for the ChucK software. It is a high-level interpreted programming language, which is strongly typed. It focuses on a 'massively overloaded' operator => which is used for variable assignment, unit generator patching and text stream processing for example. Also like SCLang, ChucK provides support for concurrency using the *Shred* data type, a kind of deterministic coroutine. Code in ChucK does not advance in a block of code unless the programmer explicitly advances it, by assigning durations to the *now* object.

2.4.1 Strongly-timed: avoiding block control-rates

ChucK's concurrent shreds and explicit control of timing within the same language as synthesis graph specification supports complete, sample-accurate control of synthesis structure. Being able to specify signal graphs dynamically in response to control events opens up the scope for algorithmic music composition in which control and synthesis evaluations are finely interleaved. Generative algorithms frequently involve some kind of input based upon feedback from previous output, thus for example a granular synthesis technique can be supported in which each grain's properties is calculated upon the demise of the previous one, along with other properties of the local context. The requirements of this thesis clearly include the 'strongly timed' classification.

2.4.2 Application-specific or generic programming language?

Procedural music languages may be written specifically for an application, or be domain-specific extensions of an existing general purpose programming languages. Amatriain notes that "Offering a completely new programming environment based on a new language is a titanic effort that needs of a very large development team. On the other hand, the language has to offer very unique and outstanding features in order to convince new users that the effort of learning it is worth the while." [4]. McCartney, author of the SuperCollider language, also wonders whether an application-specific language is worthwhile: "Is a specialized computer music

language even necessary? In theory at least, I think not. The set of abstractions available in computer languages today are sufficient to build frameworks for conveniently expressing computer music." [30].

Making use of a powerful existing programming language benefits from the proficient work of many skilled software developers, and implies additional advantages as outlined in Table 3.

Generic language feature	Benefit
Existing facilities of data description, function and control flow	Formally verified in the computer community
Existing documentation, may also be familiar to some users	Easing learning curve
Existing development, debugging and profiling tools	Minimizing user error and improving user experience
Undergone extensive revision	Removal of developer bugs and increase of efficiency
Potentially numerous extension libraries	Many additional capabilities available for scope of exploration
Formal generality	Future scalability & portability Code written in the language can be re-used in many applications.

Table 3: Benefits of using an existing programming language.

Given these advantages, why would a developer choose to write a new language? The tight efficiency, tiny time-scales and large data-processing demands of the computer music domain may often drive developers to create new languages for synthesis control. McCartney for example bemoans the lack of garbage collection

appropriate for real-time and inflexibility of syntax of most generic programming languages as the main obstacles to use for computer music.

An intermediate solution may to use a generic programming language that is designed for application extension, and which offers an open enough programming interface to be optimized to real-time demands.

3 Design & Implementation

In this section I outline the design and implementation of Vessel: how the control language was chosen and extended, how dynamic synthesis was supported, how the sample-accurate interleaving of control and synthesis was achieved and how the constraints due to the substrate were minimized. Before proceeding, let us summarize the key observations made so far.

3.1 Summary of requirements

Software for computer music composition must formally represent musical structures in a form parsable by the computer, but also humanly readable. For algorithmic composition, a process-based representation supporting generic logic and mathematical relationships is ideal. The software must also provide the means to evaluate such a description into active structural and synthesis processes to produce audio output in the form of digital samples, without compromising efficiency. Synthesis specification is generally well modeled using the unit generator model, and both serial and parallel musical flow can be well modeled using concurrent timeline constructs. It is essential for the exploration of microsound to avoid block-rate quantization of control and structural changes. Nevertheless, efficiency is a key demand.

A graphical representation may be intuitive, but a textual representation is preferred for exploration of algorithmic composition due to generality, precision and scalability[4]. Synthesis and temporal structure should be combined in the same language, with no latency of interaction between them ('strongly timed'). Procedural languages are more suitable than declarative languages for algorithmic approaches, since they directly expose the algorithms to the user, and high-level interpreted languages offer specific benefits such as interactivity over compiled languages for real-time purposes. A generic language is more portable and better supported than an application-specific language.

The implementation requirements for the software may now be summarized:

- A domain-specific (computer music composition) extension of an interpreted procedural programming language supporting generic programming and concurrency,
- Feature a vocabulary of unit generators that can be variously and dynamically connected into signal processing graphs,
- Incorporate a real-time sample-accurate scheduler to simultaneously render dynamic unit generator graphs and concurrent process timelines,
- Be efficient.

---

[4] The graphical paradigm does have advantages however, and thus this thesis project is also presented as a library extension (‘extern’) for Max/MSP, to be described in a later section.

Each of these requirements will be described in turn through the remainder of this section.

3.2 Representation language

The design model calls for a domain-specific extension of an existing high-level interpreted procedural language (supporting concurrent processing) for computer music composition. Interpreted programming languages exhibit higher-level interfaces to programming more suited to quick prototyping and testing. However, real-time audio processing involves large quantities of numeric operations, and efficiency is the primary concern when choosing a programming language for audio synthesis. The compiled languages C and C++ are established as the standards in this field, due to their efficiency, flexibility and active support in the wider programming community. Interpreted languages can be embedded within compiled languages such as C and C++ such that CPU-intensive operations can take place outside of the interpreted context, thus the interpreted language penalty can be constrained to a minimum.

3.2.1 Choice of language

The interpreted languages considered for this project included Ruby, Python, Scheme, IO, JavaScript and Lua. Though IO [12] had an appealing cleanliness to its

syntax, it was judged to be not yet mature enough for the project. Likewise, JavaScript was soon dismissed due to excessive CPU and memory overhead [3].

Scheme is a dialect of LISP, which has long been a popular choice for algorithmic computer music languages [11]. However, many users suffer with the unusual syntax of LISP variants, which is often described as both cumbersome and error-prone. Scheme is however a very powerful language, with high-level functional programming and lexical scoping features.

Ruby is an interpreted object-oriented language with a large and growing community, plenty of support and extension libraries, and has been particularly successful for web programming. It is however regarded as difficult to embed.

Python is a very popular interpreted language for application extension. It is object-oriented and incorporates features such as modules and exception handling. The syntax is clean, and Python benefits from an incredibly large selection of extension libraries, tools, supporting documentation and active community. Embedding Python is nontrivial however, and its support for concurrency is good but not excellent.

Lua is an interpreted programming language specifically designed for application extension, featuring the high level functional and concurrent programming features of LISP/Scheme with a more familiar infix syntax along the lines of Python and Ruby [20]. Lua is perhaps best regarded for its small size and efficiency, and thus is most highly regarded in the game developer community [22]. A rough comparison of efficiency of these languages is given in Table 4, based upon benchmarks at [3].

For the curious, these benchmarks show that Lua is around 10-30x slower than C, using around 2-3x more memory.

Language	**CPU usage:**	**Memory usage:**
JavaScript (SpiderMonkey)	5x	50x!
Lua	1x	1x
Python	5x	2-3x
Ruby	10x	2-3x
Scheme (MzScheme)	1.5x	4x

Table 4: Rough CPU and memory usage comparisons of Lua, JavaScript, Python, Ruby and Scheme, with Lua as the reference.

Ruby was discounted as being not sufficiently distinct to Python yet less efficient, while Scheme was discarded as offering similar benefits to Lua but with a less approachable syntax. The decision was close between Python and Lua; Python's extensive libraries and community support (including use in CSound) were appealing, but the portability, ease of embedding, formal completeness, concurrency support and overall efficiency of Lua was deemed more valuable. As Brandtsegg notes: "One could argue that Python is not the most CPU-effective language available, but it seems it's speed will be sufficient for compositional algorithmic control, as these processes do normally evolve at a relatively slow pace compared to

e.g. audio processing tasks. The exception being compositional algorithms that works directly on the audio signal." [7]

3.2.2 Lua

Lua's authors describe Lua as an extension language [20] specifically designed to be embedded within host programs and extended by domain-specific APIs. McCartney also states that an abstractly extensible language allows the programmer to "focus only on the problem and less on satisfying the constraints and limitations of the language's abstractions and the computer's hardware." [30]

Lua meets the needs of an extension language by providing good data description facilities (associative tables), clear and simple syntax, and flexible semantics. Lua is a full-fledged programming language, supporting higher-level features found in languages such as Scheme such as first-class functions and coroutines. As in Scheme, a variable in Lua never contains a structured value, only a reference to one. Lua incorporates an incremental garbage collector suitable for real-time use. Lua is frequently used for game logic programming (e.g. World of Warcraft [44]) and application extension (e.g. Adobe Lightroom [20]).

3.2.3 Concurrency

The design model calls for the support of hierarchies of interacting serial and parallel timelines to deterministically represent an algorithmic musical process.

Lua provides excellent support for deterministic concurrency in the form of coroutines, or more fully, asymmetric collaborative multi-tasking [32]. Coroutines were originally introduced by Conway in the early 1960s, and described as subroutines that act as the master program [8]. A coroutine in Lua represents an independent thread of execution, a parallel virtual machine, for deterministic scheduling. It is constructed from a function defined in Lua code. A coroutine has its own stack, its own local variables (persistent between calls), and its own instruction pointer (it resumes from the same code point at which it last yielded); but shares non-local variables with other coroutines. Lua coroutines are first-class objects: variables can point to coroutines, and coroutines can be passed into and returned from functions. Lua coroutines are asymmetric, based on the primitives *yield()* and *resume(),* and since calls from Lua to Lua are 'stackless', the algorithm by which coroutines are resumed can be determined very flexibly (in fact, the 'main process' of the interpreter is itself a coroutine). Coroutines have helped Lua to gain popularity in the game development community. For a more detailed description of coroutines and their use in Lua, see [32].

Each concurrent musical timeline in Vessel is represented as Lua coroutine along with metadata such as the sample-clock time it should next continue processing. The body of a timeline-coroutine is a Lua function, and can include the full range of dynamic control structures that the Lua languages offers, along with a small number of additional functions to interact with the scheduler. In this model, a timeline may represent the entire composition, or a single grain, and each timeline responds

distinctly to the passage of time according to internal determinations. In functional programming parlance, coroutines are continuations: they are objects that model 'everything that remains to be done' at certain points in the functional structure. Thus the evaluation of a composer's script is implemented as a variation of the continuation-based enactment design pattern [27].

The flexible nature of Lua coroutines and the transparent C API permits the developer to specify with great freedom the manner in which they are resumed, and this will be described in detail in the Scheduling section below. The manner in which the unit generator graphs are specified through Lua will also be described in a later section, however it roughly follows the game development paradigm, which calls for efficiency and flexibility: "many games are coded in (at least) two languages, one for scripting and the other for coding the engine" [22].

Finally, to summarize the representation model in the language extension: a computer music composition is evaluated in real-time into hierarchies of dynamically interleaved concurrent processes and relatively outside-time structures. The processes themselves are iteratively interpreted over discrete time, producing temporal form as digitally produced sound. The author does not suggest that all composition fit the representation model described above, or that the formalized models that the computer can provide are necessary for music. However the model does provide a working hypothesis upon which to develop an implementation that may be evaluated in practice.

3.2.4 Existing Lua / Audio bindings

A web search uncovers few Lua bindings of audio libraries, predominantly comprising simple sound-mixer additions to gaming software. CSound 5.0 includes a Lua binding which is mostly limited to loading and rendering orchestra and score files and thus not of interest to this thesis.

The closest relation found is the ALUA project, a part of Günter Geiger's doctoral thesis [15]. The software was not available for testing, so the comparison here is purely theoretical. A general observation however is that ALUA is "a research system only" and "although not my main focus, there is still a lot of work to be done until ALUA is a fully usable language for computer music" [15]. Geiger chose Lua for the flexible and expressive high-level syntax, and ALUA supports unit generator constructors and explicit control of scheduled time in a similar but reduced manner to Vessel. It is not clear whether it supports concurrency. The ALUA language extensions are not as developed as Vessel; for example, operators are not overloaded for unit generators (`Add(Sine(), Sine())` rather than `Sine() + Sine()`). Overall, it appears that ALUA may no longer be a supported project.

3.3 Software synthesis

One can distinguish between software synthesis applications, such as the project outlined in this thesis, and software synthesis libraries. Synthesis libraries provide DSP functions or unit generator modules written in an efficient programming

language, with application programming interfaces (APIs). Libraries offer atomic access to elementary DSP functions and make minimal assumptions as to how they may be used. The developer of a synthesis application can take advantage of many existing libraries for DSP to provide tested and recognized functionality.

This section describes the various synthesis libraries considered and implemented in Vessel. In accordance with the design requirements, any synthesis considered library for Vessel must meet the following requirements:

- C/C++ API in order to provide bindings to Lua
- Single-sample or variable block-size processing for microsound
- Efficiency of performance and minimal opcode setup/removal cost
- Minimal dependencies for portability
- Open-source distribution for portability

3.3.1 Synz

Synz [40] is a C++ library for common signal processing tasks, providing a set of efficient opcodes based on a set of low-level stateless operator functions and generic data structures. The evasion of preconceived use-cases allowed the author to very easily bind this library to Vessel. Synz neither assumes nor prevents block processing, it provides a low-level but consistent C++ API, and is distributed as open source.

3.3.2 STK

STK [46] is suitable to be embedded within Vessel, since it provides a C++ API and supports single-sample opcode evaluation with a generic *tick()* method, and is distributed as public domain source code. STK is familiar to many computer music researchers, and would be a valuable asset due to its particular support for physical modeling synthesis. A binding of STK in Vessel is therefore planned as future work.

3.3.3 CSL

The CREATE Signal Library [37] is an object-oriented C++ library of synthesis unit generators. CSL is inherently block-processed and embeds the signal graph representation within unit generators themselves; for these reasons CSL could not be efficiently utilized within Vessel.

3.3.4 CLAM

CLAM [4] is a framework for building audio applications, both in C++ and through a graphical editor application (the CLAM Network Editor). CLAM implements data-flow architecture for processing, distinguishing between synchronous data flow and asynchronous control flow. In contrast to the Max family, it is the control flow that is constrained to numeric types, while data flow

may include signals, spectra, and complex data structures. A CLAM network, expressed as an XML file, can be converted into a standalone application with a graphical interface designed using the QT GUI toolkit, and CLAM is distributed under an open source license.

CLAM processing nodes can support dynamic block sizes (up to a maximum size) within the audio thread, through reconfiguration of processing nodes cannot occur in the audio thread. The architecture of unit generators and source code is very similar to the approach taken in the Vessel system, though it remains to be evaluated whether CLAM could be used within the Vessel system.

3.3.5 SndObj

SndObj (Sound Object Library [24]) is an open-source C++ generic audio processing library incorporating many opcodes and utilities. SndObj can use different block sizes per opcode instance, however it remains to be evaluated whether these can be efficiently modified dynamically (using `SndObj::SetVectorSize`) in order to be incorporated within the Vessel scheduler[5].

[5] The documentation at http://music.nuim.ie//musictec/SndObj/SndObj_Manual-2.6.1.pdf suggests that this will not be the case.

3.3.6 Csound opcodes

Csound has over 450 opcodes for audio signal processing, probably the most complete of any software. Csound is distributed as open-source, and provides APIs in C and C++ in order to be embedded within other languages and applications. In fact, a binding of Csound for Lua already exists in the Csound distribution, however this binding is high level (supporting the loading and rendering of Csound files) rather than offering low-level access to the synthesis opcodes themselves. Examining the source code reveals that opcodes may involve strong dependencies on the CSound host, making it unlikely that they can be generally used within other scheduling environments.

3.4 Scheduling

> "This moment which I live, this thought which crosses my mind, this movement which I accomplish, this time which I beat: before it and after it lies eternity; it's a non-retrogradable rhythm." Olivier Messiaen, in [45]

The design model calls for a real-time sample-accurate scheduler to simultaneously render dynamic unit generator graphs and concurrent process

timelines, in order to finely interleave algorithmic musical structure and signal processing. This section describes how this is achieved.

3.4.1 Unit generator graph traversal

Signal processing of unit generator directed graphs must be executed in deterministic orders such that a node's inputs have been determined before the node can output. A naïve tree-search algorithm may quite effectively achieve this. It can also be viewed as a formulation of the producer-consumer problem, and thus both push (leaf to root) and pull (root to leaf) models may be used for the traversal. Static scheduling pre-determines the graph before executing, while dynamic scheduling evaluates the graph at runtime. Dynamic scheduling can therefore handle changes to the DSP graph at run-time.

Normally the unit generator graph is viewed as an indivisible process, such that each node processes equally sized tokens per iteration (typically matching the block-rate). We have seen however that this model is insufficient when control or graph changes are required more finely than the block-rate.

The Vessel scheduler algorithm attains state changes not quantized to the block rate by allowing arbitrary sub-divisions of the block duration. The cost incurred is that graph traversal is derived dynamically at each state change. Traversing only those portions of the graph hierarchy upon which the state change is deterministically dependent can minimize this cost (lazy dynamic scheduling). The

graph manager in Vessel traverses only the deterministic input[6] dependencies of unit generators within the signal-processing graph, only up to the current state-change time-stamp, and thus implements just-in-time sample-accurate graph dynamics. Between state changes, the signal processing proceeds in sample blocks, taking advantage of block processing efficiency whenever possible.

3.4.2 Threads considered harmful

Vessel has the responsibility to maintain sufficient potential in the system for free action of arbitrary and independent state change of the synthesis graph during real-time performance. Synthesis processing and structural control are usually separated into distinct operating system threads for efficiency, however as noted by Dannenberg & Bencina [10]:

> "The simple timing approach, which is something like
>
> A(); sleep(5); B(); sleep(3); C(); sleep(7); ...
>
> will accumulate error due to finite computation speed and system latencies."

[6] Within a directed dynamic graph, it is not possible to schedule with sample accuracy for input nodes that are also downstream of the current processing context (i.e. cycles), but the latency will be automatically minimized to block-rate.

The standard solution to achieve scheduling determinism is to provide an event buffer with an acceptable latency and schedule accurately time-stamped events ahead within this buffer. Effectively, the buffer conceals the indeterminacy (jitter) of synchronization between the system timer, the message thread and the audio thread sample clock. Early implementations of Vessel took this approach. Unfortunately, this solution incurs indeterminacy if a scheduled event is micro-temporally dependent on another event's output. To achieve sample accuracy of state change in response to audio events, it became apparent the composition script must execute in the same system thread as the synthesis processing and manage event scheduling and execution directly with the audio sample clock.

The cost of interpreted Lua code in the high-priority audio thread is minimized by maintaining the expensive signal processing and graph management/scheduling entirely within C++ code, only calling into Lua for the relatively cheap coroutine evaluations. Furthermore, expensive workarounds to threading indeterminism (locking, semaphores etc.) are entirely avoided within the Vessel interpreter [26]. The Vessel language can thus support thousands of concurrent coroutines with deterministic behavior and shared memory, rather than hundreds of concurrent threads with unpredictable behavior and buffered/locked memory[7].

[7] On the other hand, pre-emptive concurrency can be achieved if desired (e.g. for file loading) using the Lanes extension described in section 4.

3.4.3 Scheduling dynamic graphs and coroutines

The scheduler algorithm manages the proper execution of both the unit generator graph and the list of active coroutines. The scheduler 'wakes up' each coroutine timeline when its sample-clock time is due, and the coroutine proceeds through its virtual machine instructions until it completes or it yields to reschedule itself at a future sample-clock time. While it proceeds, the composition time is effectively frozen. If at any point during the coroutine's execution, it triggers a state change in a unit generator or the signal processing graph, the affected portion of the signal processing graph will be traversed and processed up to the current composition timestamp. Once all active timelines are complete for the current audio block, the synthesis graph is once more traversed from the root node, to calculate any remaining indeterminate samples. Composition time can now advance to the end of the block.

Since Lua includes dynamic and indeterminate control structures, the effective control rate can be arbitrarily specified or even indeterminately derived with sample accuracy as the performance proceeds.

3.5 Efficiency

Besides the optimizations made in the scheduling algorithm, the efficiency of the synthesis library functions and the performance of Lua, the principal issue for Vessel is memory management. The high priority real-time audio thread demands that

processes to occur in bounded time [10], but memory allocation cannot always satisfy this constraint. Dynamic graphs therefore require memory management techniques to achieve bounded time performance.

Vessel implements up-front allocated free-list memory pools for the most dynamic elements: audio buffers, coroutines and unit generators. Pre-allocated memory is recycled as the program executes, and pools grow if needed, utilizing a memory allocator optimized for real-time use [25]. The same real-time allocator easily replaced *malloc* and *free* for all Lua calls using the Lua API. The reader is referred to the Real Time Memory Management patterns in [10] for a fuller description of these techniques.

Since version 5.1 the Lua language incorporates an incremental garbage collector. Lua 5.1 saw the introduction of an incremental collector adaptable for real-time use (provided in response to requests from game developers), avoiding potentially long pauses during garbage collection [22].

4 Description & Examples

4.1 Single language, multiple applications

The language extensions to Lua to support sample-accurate synthesis and control constitute a software library that can be embedded or dynamically loaded within other applications. The three existing applications described in this section should be taken as indicative of its scalability. Additional future targets could include audio plug-ins (VST, JACK, AU etc), cross-platform support for Windows and Linux, and embedded devices such as PDAs and gaming consoles.

4.1.1 Vessel command line

A command-line implementation of Vessel is available, which can be used for testing, or controlling from other applications (Figure 2). Input arguments specify the main Lua script file to execute, an optional maximum duration, and an optional file in which to record audio output.

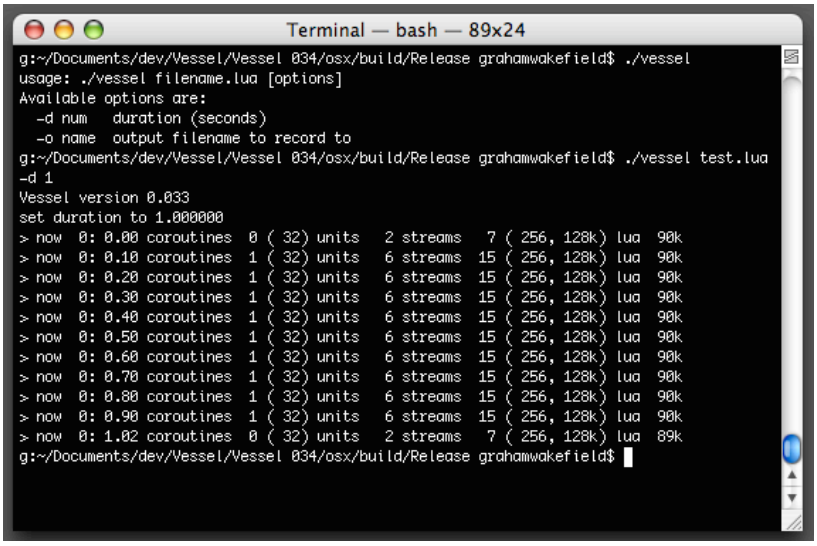

Figure 2: The Vessel command line tool in use.

4.1.2 Vessel application

Vessel exists as a standalone application (presented for OSX but scalable to other platforms) incorporating the Vessel language and synthesis scheduler along with an audio scope and status window, and a Lua code editor window (Figure 3). The code editor is based upon the Cocoa MDI (multiple document interface) development pattern, and features syntax highlighting for both Lua and Vessel reserved words. Run, Merge, and Stop buttons restart, mix and terminate Lua scripts respectively.

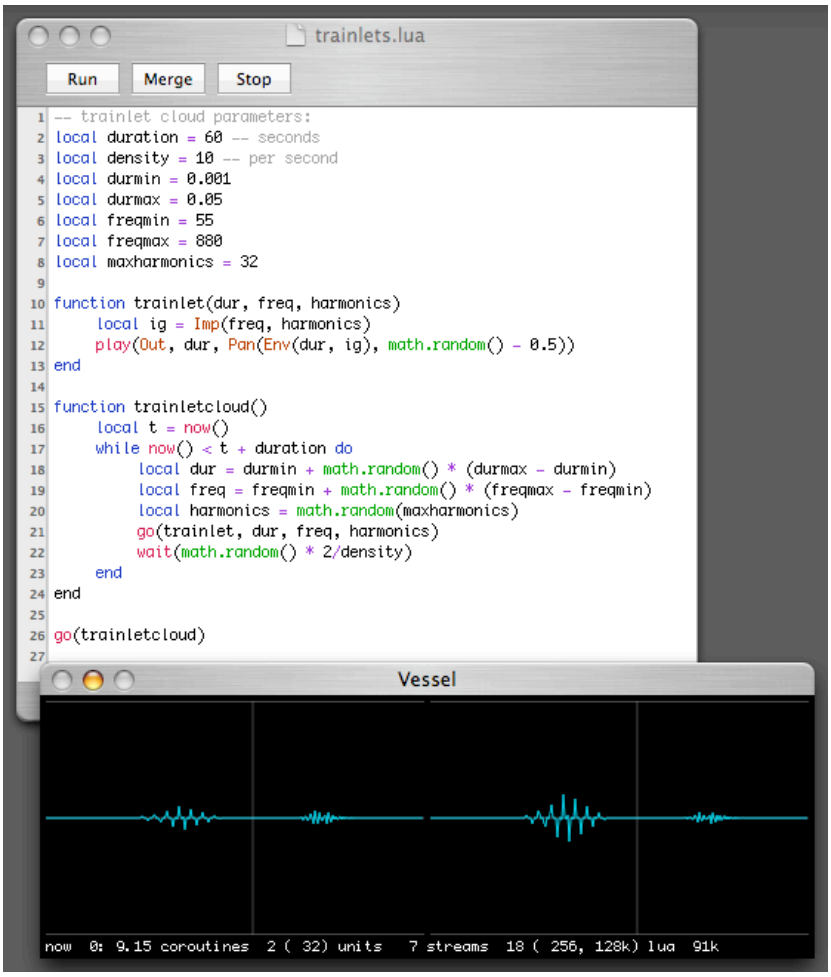

Figure 3: Screenshot of the Vessel standalone application on OSX.

4.1.3 Vessel in Max/MSP (the lua~ external)

Lua~ is an extension (external) for the Max/MSP environment containing the Vessel language and synthesis scheduler along with bindings for relevant Max/MSP components (Figure 4).

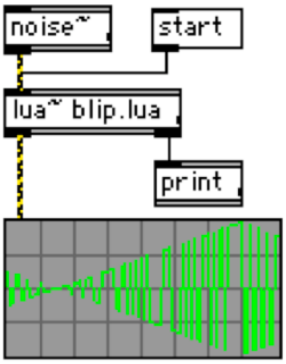

Figure 4: Screenshot of the lua~ object within a Max/MSP patch.

A *lua~* object in a Max patch can load and interpret Lua scripts and receive, transform and produce MSP signals and Max messages accordingly. The `In` and `Out` busses represent the lua~ object inlets and outlets respectively. Messages set to the lua~ external are interpreted as function calls with arguments[8], and the function `outlet()` is used to send Max messages out of the *lua~* external. Lua~ provides the unified integration of text-based and graphical meta-mechanisms for audiovisual composition within a single application.

[8] E.g. the Max message "`print hello 5`" results in the Lua call `print("hello", 5)`.

4.2 The language

This section presents an overview of the language extensions to Lua that constitute Vessel.

4.2.1 Scheduler functions

`now()`

Returns the number of seconds since the containing coroutine was created. With an optional argument (any), returns the root scheduler timestamp.

`go([`*`delay`*`], `*`func`*`, [`*`args`*`…])`

Adds a new coroutine to the scheduler queue, and returns the coroutine. The coroutine will begin after *delay* seconds (or immediately if not specified), based upon the function *func*, which will be passed all values in *args*.

`wait([`*`delay`*`])`

Pauses the containing coroutine for *delay* seconds. The `wait()` function in Vessel is deterministically sample-accurate; it effectively causes the currently executing coroutine to yield and reschedule itself, and permit the next scheduled coroutine or synthesis process to resume.

`play(`*`bus`*`, `*`dur`*`, `*`unit`*`)`

Adds the unit generator *unit* as an input to *bus*, pauses the containing coroutine for *dur* seconds, then removes the unit generator from *bus*. Equivalent to `bus:add(unit); wait(dur); bus:remove(unit).`

`abort([`*`coro`*`])`

Aborts the coroutine coro immediately, removing it from the scheduler and freeing any memory resources unique to the coroutine (including other coroutines launched by coro and not referenced elsewhere). Note: this may lead to 'stuck notes' if the coroutine had added unit generators to external busses. An alternative strategy is being investigated to avoid this issue.

4.2.2 Units

Units are Lua objects that encapsulate C/C++ unit generator DSP code. Units flexibly handle constant numbers or other Units for most input parameters. Units may provide Lua methods to determine instantaneous state changes. Units can expand to multi-channel upon demand, and individual channels can be indexed with the `unit[n]` notation, where n is an integer starting from 1. Units can be composed into graphs via their inputs, by using Busses (see below), or by using math operators (`+, -, *, /, %, ^`).

Vessel currently incorporates a minimal set of Units for development purposes, outlined below. This list will grow rapidly in the near future.

Noise generators:	`Noise(), Pink()`
Oscillators:	`Sine([freq]), Square([freq]),` `Tri([freq]), Saw([freq]),` `Imp([freq, harmonics, mode]),` `Dsf([freq, fratio, aratio, harmonics])`
Generators:	`Decay([t60]),` `Curve([dur, curve, start, end])`
Filters:	`Smooth(input, [factor]),` `Biquad(input, [freq, resonance, mode])`
Shapers:	`Env(duration, input, [shape])`
Spatializers:	`Pan(input, [pan]),` `Reverb({parameters})`
Math:	`Round(input), Floor(input), Ceil(input),` `Abs(input), Min(input, operand),` `Max(input, operand), Mean(input,` `operand),` `Gt(input, operand), Lt(input, operand),` `Clip(input, a, b), ClipB(input, a, b),` `Wrap(input, a, b), Fold(input, a, b),` `+, -, *, /, %, ^`

4.2.3 Busses

Busses are a particular kind of Unit into which other Units can write. Busses therefore allow arbitrary signal mixing, efficient effects chains, and graph cycles.

Busses add the `bus:add(unit)` and `bus:remove(unit)` methods to add or remove Unit writers from a Bus.

Two special global Busses, named `Out` and `In`, represent the output and input channels of Vessel respectively. The number of channels matches the number of channels of the sound card (standalone) or number of inlets/outlets (lua~).

4.2.4 Distributed interaction (OSC, MIDI)

Vessel supports MIDI and OSC for input and output. Ports are created from constructors taking textual or numeric qualifiers (e.g. `MidiIn(1)` or `OscOut("localhost", 7400)`. Messages are read using the `:read()` method, and sent using `OscOut:send(…) or MidiOut:noteon(note, vel, chan), MidiOut:control(cc, val, chan)` etc.

4.2.5 Lua libraries

The entire Lua core libraries are available for use in the script, including standard math and string functions. Additional functions are defined by Vessel for common musical tasks, such as `miditofreq()`. Lua itself is an extensible language, and any libraries written for stock Lua can be dynamically imported and used within a Vessel Lua script, to provide scientific math functions, networking capabilities, etc.

4.2.6 Lanes

The Lua Lanes project [22] is a special library incorporated into Vessel to enable the sharing of simple data types between Lua states in distinct operating threads, via named FIFO message queues. Lanes can also be utilized to create pre-emptive tasks in distinct system threads, which may be useful to compute expensive non-real-time operations.

4.3 Examples

In this section I present simple demonstrations of the capabilities of the presented software.

4.3.1 Minimal example: note list player

The following code fragment defines a coroutine process to progressively iterate a note list table and interpret its data as a sequence of notes to synthesize using a Sine oscillator:

```
-- a simple sequence player:
local player = function(notelist)
 for i = 1, #notelist do
     local event = notelist[i]
     play(Out, event.dur, Sine(event.freq))
 end
end

-- a minimal sequence:
local triplet = {
```

```
 { freq = 440, dur = 0.5 },
 { freq = 880, dur = 0.25 },
 { freq = 660, dur = 0.25 }
}

-- play the sequence concurrently:
go(player, triplet)
```

This is minimally equivalent to the orchestra-score model of Csound et al., yet can be endlessly extended with functional and concurrent programming. For example, the table of event parameter sets could just as easily contain functions or other coroutines in place of numbers. A library of complex and generative pattern streams can be designed using tables, functions and coroutines, according to the composer or programmer's discretion.

4.3.2 Microsound Synthesis

Curtis Roads describes a diverse catalogue of rich synthesis techniques dependent upon the micro time scale [43]. Vessel has been designed to support microsound synthesis in general terms, but for the purposes of this thesis we will consider trainlet synthesis as a specific indicative example: "A trainlet is an acoustic particle consisting of a brief train of impulses. Like other particles, trainlets usually last between 1 to 100 ms." [43]

Trainlet clouds incorporate at least three levels of hierarchy: the trainlet cloud, each stream of trainlets, and the impulses within each trainlet. The following code sample demonstrates a basic specification of a trainlet cloud:

```
-- trainlet cloud parameters:
local duration = 40 -- seconds
local density = 80 -- per second
local durmin = 0.001
local durmax = 0.1
local freqmin = 220
local freqmax = 880
local maxharmonics = 40

function trainlet(dur, freq, harmonics)
 local ig = Imp(freq, harmonics)
 play(Out, dur, Pan(ig * Decay(dur), math.random() - 0.5))
end

function trainletcloud()
 local t = now()
 while now() < t + duration do
     local dur = durmin + math.random() * (durmax - durmin)
     local freq = freqmin + math.random() * (freqmax - freqmin)
     local harmonics = math.random(maxharmonics)
     go(trainlet, dur, freq, harmonics)
     wait(math.random() * 2/density)
 end
end

go(trainletcloud)
```

4.3.3 Concurrent processes

This simple example demonstrates the layering of concurrent processes. Note that both processes are instantiated from the same function template, but with distinct arguments:

```
-- simple percussive repeater as coroutine template:
function pattern(stepdur, freq, p)
 while true do
         -- create a DSP graph:
         local f = Sine(freq, 0)
         local ugen = Pan(Sine(freq, 0) * Decay(0.2) * 0.5, p)
         -- play for one step, then pause for one step:
     play(Out, stepdur, ugen)
     print(f:freq():current())
         wait(stepdur)
 end
end

-- launch coroutine immediately,
-- at 1/6s step size, 440Hz, pan right:
go(pattern, 1/4, 440, 0.5)

-- launch coroutine after 2 seconds,
-- at 1/4s step size, 330Hz, pan left:
go(2, pattern, 1/6, 330, -0.5)
```

4.3.4 Sample-accurate dynamic graphs

In Four Criteria of Electronic Music, Karlheinz Stockhausen described a technique to produce synthetic tones that demonstrated the continuum between pitch and rhythm (and by extension, timbre and polyrhythm). The following code example demonstrates that graph can be created and destroyed at any control rate, slowly moving from rhythm through to timbre. The period between each new instantiation gradually reduces from one second to a single sample:

```
local dur = 1

local function player()

 -- launch child activity to change tempo:
```

```
 local t = go(function()
     while dur * samplerate > 1 do
         dur = dur * 0.9
         print("dur", dur * samplerate, "samples")
         wait(0.1)
     end
 end)

 -- loop until down to a single sample duration:
 while dur * samplerate > 1 do
     local s = Sine(55 * math.random(8))
     play(Out, dur, Env(dur, s))
 end
end

go(player)
```

4.3.5 Generative signal graphs

In the following code sample, a series of events are produced in which each has a signal graph that is determined randomly. The *node()* function selects from the template table of functions to generate each part of the graph. Note that the functions themselves recursively call *node(),* until the proper depth is reached.

```
-- mixer with reverb
local mix = Out:add(Bus())
local verb = Out:add(Reverb())
verb:add(mix * 0.1)

-- a set of templates to generate ugen nodes
local node -- forward declaration
local templates = {
 function(a) return Sine(node(a) * (2^math.random(10)),
     math.random()) end,
 function(a) return Tri(node(a) * (2^math.random(10)),
     math.random()) end,
 function(a) return Smooth(node(a), math.random(10)) end,
 function(a) return Decay(math.random() * 4) end,
```

```
 function(a) return Env(math.random(), node(a),
     math.random(7)) end,
 function(a) return node(a) * node(a) end,
 function(a) return Fold(node(a), node(a), node(a)) end,
 function(a) return Mean(node(a), node(a)) end,
 function(a) return Abs(node(a)) end,
}

-- recursive graph node generator:
function node(depth)
 if depth > 0 then
     -- call a function from the template
     local t = templates[math.random(#templates)]
     return t(depth - 1)
 else
     -- just use a constant:
     return math.random()
 end
end

-- event generator:
function note()
 -- make a new duration
 local d = 0.1 + math.random() * 3
 -- generate a graph
 local graph = node(3)
 -- scale by an envelope
 graph = graph * Smooth(Decay(d), 100) * (math.random() - 0.5)
 -- DC block
 graph = Biquad(graph, 1, 1, 1)
 -- panning graph (pan at sample rate)
 local panner = node(2)
 -- clip to reasonable range
 panner = Clip(panner, -0.5, 0.5)
 -- schedule it!
 play(mix, d, Pan(graph, panner))
end

-- produce some graphs in sequence:
while true do
 local d = 0.1 + math.random()*0.1
 go(note)
 wait(d)
end
```

5 Conclusion

The aim to finely interleave signal generation, event handling and composition structure within a unified composition language has been achieved. Embedding functional logic on a per-event basis can support both sequenced and algorithmic composition in tangled hierarchies of parallel and serial time flows, and the composer's script itself becmes the source of synthesis complexity.

This section returns to some points that bear further elaboration, evaluates some of the limitations of Vessel, and highlights areas for future development.

5.1 Extensible for musical structures

Many computer music composition systems incorporate abstractions and behaviors appropriate for common musical structures. This can be very helpful, however musical structures should not obstruct the free exploration of new ideas. Many higher-level problems of musical signal representation are therefore not directly addressed by Vessel. In the design of Vessel the author strove to avoid limiting its use to certain ways of thinking about music, instead providing lower-level general 'meta-mechanisms' (to follow the acknowledged philosophy of both Max [56] and Lua [21]) with which musical ideas can be constructed. The use of these mechanisms within real musical composition remains to be evaluated.

By minimizing hard-coded distinctions and instead providing meta-mechanisms, Vessel grants the composer greater responsibility towards the creative output (for better or for worse). Truax states: “Ultimately, a computer music composition reflects the musical knowledge embodied in both the software system that produced it and the mind of the composer who guided its realization. The interaction between these two bodies of knowledge is the essence of the creative musical process” [50] An open-ended system may begin with little or no musical knowledge, however if this knowledge is provided by the composer, the system may provide tools with which to represent and then make use of this knowledge, and by consequence of organizing complexity, afford new points of view otherwise obscured.

5.2 Avoiding an application-specific language

The decision to make use of an existing language rather than write a new one not only simplified the implementation incredibly, but also suggests a wide scope for experimentation in the future. The value of using an embedded language for composition software is apparent in the many extensions to Csound derived from Python.

I was fortunate to find a language as efficient, portable and well defined as Lua, particularly with respect to the ability to extend coroutines into a sample-accurate time domain. Throughout its ten-year history, Lua has been designed to be a simple, portable, efficient extensible extension language.

Since, beyond minor syntactical differences, this is the principal contrast with the ChucK language, a slightly more detailed consideration is appropriate. Unlike Vessel, ChucK cannot benefit from existing code, documentation or extension libraries written for general programming languages. The following list gives an indication of some of the Lua extension libraries that could be used within Vessel:

- Lbc, Numeric Lua (extended math)
- LPeg, Lrexlib (textual pattern matching)
- LuaExpat (XML)
- Pluto, lper (persistence & serialization)
- LuaCairo, LuaPDF (2-D graphics for printing)
- LuaFileSystem, Lposix, LuaZip, lgzip (file access & compression)
- LuaSocket, LuaCURL, CGILua (networking & web)
- LuaSQL, luasqlite (databases)
- Luaunit, Lunit (unit testing)
- Lua-eSpeak (speech synthesis)
- LuaJava, LuaObjCBridge (interfacing other languages & libraries)

A particular concern of the author is future portability: compositions written in application-specific languages become unrealizable as soon as the application is no longer supported. While applications such as Max and particularly CSound have enjoyed longevity, the history of computer music composition is littered with now defunct composition tools. While it is true that established programming languages

may suffer the same fate, a formally defined generic programming language should be easier to 'port' into future representations than an application-specific language.

5.3 Drawbacks

Despite the optimizations made, there is clearly a cost incurred by using an interpreted language in audio thread. It remains to be tested how detrimental this effect may become; however the author has been quite satisfied with performance so far.

Although the abstraction level of the unit generator is at a level typical for computer composition environments, certain activities may require access to direct sample-producing functionality. The author is considering extending Vessel to include a secondary level offering direct function calls upon sample buffers accordingly.

5.4 Vessel in use

A component of the project was presented at the UCDarNet symposium in January 2007, and a paper describing the Max/MSP implementation (along with a workshop and performance) has been accepted for the Digital Arts Week conference at ETH Zurich for July 2007. Another paper documenting the project has been accepted to the ICMC 2007, Copenhagen. The software has been used already in a

composition by the author and Wesley Smith [47], performed a number of times in public.

5.5 Future Work

The software described in this thesis is planned for beta release in the summer of 2007. The author believes that the real test of a design concept is the implementation and evaluation in practice, and (beyond bugs!) no doubt many new ideas for extension will arise from such activity. However, the author in this section will outline some of areas of development already identified.

5.5.1 Extended set of unit generators

The palette of unit generators is thus far minimal, but should indicate to the reader that the extension to a more complete is feasible. Certain unit generators may call for a different approach to timing or signal representation however; a case in point would be FFT and IFFT processors. The design of such an interface is planned as future work.

5.5.2 Notifications & audio triggers

A valuable addition to the scheduler/language would be a notification mechanism for the coroutine yield `wait()` call, as an optional alternative to

durations. Resuming a coroutine yielded in this way might be triggered by message events (such as `wait(midinote)`), or more interestingly, events due by audio analysis. For example, dynamic processing of Wavesets [54] might involve yielding a coroutine until a zero crossing occurs on a specified input. It should be noted however that any situations involving cyclical dependencies must cause a certain buffer of latency in response.

5.5.3 Runtime specification of unit generators

Thus far, signal processing may be specified using unit-generator graphs for micro-temporal durations, however like many other environments, Vessel is limited to the vocabulary of unit generators provided, and unit generators themselves remain opaque for the sake of efficiency. The standard solution to this is to provide a software development kit (SDK), which developers may use to write new unit generators in C or C++. New unit generators must be built in C++ and compiled prior to use. At some level there will always be a trade-off between design time (coding, compiling, loading) and execution time (efficiency).

A more novel approach may be supported by Faust [34], in which a high-level functional language is used to specify unit generator algorithms, which can in turn be automatically compiled into generated C/C++ code for many different composition environments. The code generated by Faust may not be as efficient as hand-written code, but it a) allows users with no C/C++ experience to create unit generators, b)

creates representations of unit generators that are not specific to any environment, good for portability and longevity, and c) may benefit from a shorter implementation-test loop with runtime compiling & loading.

A third option for Vessel is to provide utilities to write signal-processing code directly in Lua using basic primitives, and machine code compilation with the LuaJIT compiler [35]. This will probably be less efficient than Faust, but can provide an implementation-test loop so short that signal processing code itself could be the result of a generative algorithm at runtime!

5.5.4 Graphics

The integration of algorithmic audio and graphics has long been a goal of the author. Fortuitously, Wesley Smith, a fellow graduate student at MAT, had been simultaneously developing a 3D graphics toolkit based upon the Lua language and the OpenGL standard, named Abelian. Vessel and Abelian will communicate and share data through serialized message buffers, and may share code. In addition, user interface components can be created using the GLV OpenGL user interface library [33], developed by the authors and other researchers at MAT.

The potential uses include graphical interfaces and visual instruments for real-time performance and installation, visual music composition, audio-visual software art and scientific or pedagogical visualizations. Conjoining a 3D graphics and user-

interface toolkit with audio synthesis for a generalized digital media composition environment is perhaps the most exciting future direction of research for this project.

6 References

[1] H. Abelson, G. J., Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Massachusetts, USA, (1996).

[2] P. Ackerman, *Object-Oriented Time Synchronization of Audio-Visual Data in a Multimedia Application Framework*, PhD dissertation, University of Zurich, Switzerland, (1995).

[3] Alioth, *Computer Language Benchmarks Game*, retrieved April 2007: http://shootout.alioth.debian.org/gp4/benchmark.php?test=all&lang=lua&lang2=javascript.

[4] X. Amatriain, *An Object-Oriented Metamodel for Digital Signal Processing, with a focus on Audio and Music*, Doctoral Thesis, Departament de Tecnología, Universitat Pompeu Fabra, Barcelona, Spain (2004).

[5] R. Bianchini, A. Cipriani, *Virtual Sound: Sound Synthesis and Signal Processing – Theory and Practice with Csound*, Contempo s.a.s, Rome, Italy, (2001).

[6] R. Boulanger, *The Csound Book*, MIT Press, Cambridge USA, (2000).

[7] Ø. Brandtsegg , "A Sound Server Approach to Programming in Csound: Building a modular system for realtime algorithmic composition and improvisation", *Csound Journal,* Vol. 1, Issue 1 (Fall), (2005).

[8] M. Conway, *Design of a separable transition-diagram compiler*, in Communications of the ACM 6, 7 (July), 396–408, (1963).

[9] L. Dami, E. Fiume, O. Nierstrasz, D. Tsichritzis, *Temporal Scripting using TEMPO*, Centre Universitaire d'Informatique, Université de Genève, (1994).

[10] R. Dannenberg, R. Bencina, *Design Patterns for Real-Time Computer Music Systems*, Septermber (2005); http://www.cs.cmu.edu/~rbd/doc/icmc2005workshop/real-time-systems-concepts-design-patterns.pdf

[11] R. Dannenberg, P. Desain, H. Honing, *"Programming Language Design for Music"* in *Musical Signal Processing*, Swets & Zeitlinger, Netherlands, (1997).

[12] S. Dekorte, *io*, retrieved June 2007: http://www.iolanguage.com/about/

[13] B. Garton, *Maxlisp v0.8*, July (2004); http://www.music.columbia.edu/~brad/maxlisp/

[14] B. Garton, D. Topper, *RTcmix – using CMIX in real time*, In Proceedings of the International Computer Music Conference. International Computer Music Association, (1997).

[15] G. Geiger, *Abstraction in Computer Music Software Systems*, doctoral thesis, Department of Technology, Universitat Pompeu Fabra, Barcelona, Spain (2005).

[16] T. Grill, *Py/Pyext*, retrieved April (2007); http://grrrr.org/ext/py/.

[17] E. Giordani, "GSC4: A Csound Program for Granular Synthesis", in *Virtual Sound: Sound Synthesis and Signal Processing – Theory and Practice with Csound*, Contempo s.a.s, Rome, Italy, (2001).

[18] F. Guerra, *LuaGL*, retrieved April (2007); http://luagl.wikidot.com/.

[19] C. Henry, A. Momeni, "Dynamic Independent Mapping Layers for Concurrent Control of Audio and Video Synthesis", *Computer Music Journal*, 30:1, pp.49–66, Spring (2006).

[20] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, "Lua - an extensible extension language", in *Software: Practice & Experience* 26 #6, 635-652, (1996).

[21] R. Ierusalimschy, *Programming in Lua (2nd ed.)* PUC-Rio, Rio de Janeiro, (2006).

[22] R. Ierusalimschy, L. H. de Figueirdo, W. Celes, *The Evolution of Lua*, to appear in ACM HOPL III, (2007).

[23] A. Kauppi, "Lua Lanes – multithreading in Lua", April (2007); http://kotisivu.dnainternet.net/askok/bin/lanes.html.

[24] V. Lazzarini, *Sound Processing with the SndObj Library: An Overview,* Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-01), Limerick, Ireland, December 6-8, (2001)

[25] D. Lea, *A Memory Allocator*, April (2000): http://gee.cs.oswego.edu/dl/html/malloc.html

[26] E. Lee, *The Problem with Threads*, Technical Report No. UCB/EECS-2006-1, Electrical Engineering and Computer Sciences University of California at Berkeley, (2006).

[27] D. Manolescu*, Workflow Enactment with Continuation and Future Objects*, in OOPLSA'02, Seattle, WA, (2002).

[28] D. Manolescu, *A Dataflow Pattern Language* in Proceedings of the 4th Pattern Languages of Programming Conference, (1997).

[29] Marsden, "Representing Melodic Patterns as Networks of Elaborations," in *Computers and the Humanities*, Vol 35, 37-54, (2001).

[30] J. McCartney, "Rethinking the Computer Music Language: SuperCollider," *Computer Music Journal* 26, 4, 61–68 (2002).

[31] E. Miranda, *Composing Music with Computers*, Focal Press, USA, (2001).

[32] L. de Moura and R. Ierusalimschy, *Revisiting Coroutines*, Computer Science Department – PUC-Rio, PUC-RioInf.MCC15/04 June, (2004).

[33] E. Newman, L. Putnam, W. Smith, G. Wakefield, "GLV – OpenGL Application Building Kit," December (2006); http://glv.mat.ucsb.edu/.

[34] Y. Orlarey, G. Albert, S. Kersten, *DSP Programming with Faust, Q and SuperCollider*, Proceedings of the 4th International Linux Audio Conference, Karlsruhe, Germany (2006).

[35] M. Pall, *LuaJIT*, retrieved April (2007); http://luajit.luaforge.net/.

[36] T. M. Parks, *Bounded Schedule of Process Networks*, PhD thesis, University of California at Berkeley, USA, (1995).

[37] S. Pope, X. Amatriain, L. Putnam, J. Castellanos, and R. Avery, *Metamodels and Design Patterns in CSL4* in Proceedings of the 2006 International Music Conference, Computer Music Association, (2006).

[38] M. Puckette, *Pure Data*, in Proceedings of the 1997 International Music Conference, Computer Music Association, pp. 224–227, (1997).

[39] M. Puckette, "Max at Seventeen," *Computer Music Journal* 26, 4, 31-43 (2002).

[40] L. Putnam, *Synz*, retrieved April (2007); http://www.uweb.ucsb.edu/~ljputnam/synz.html.

[41] D. Pyon, *Csound~*, retrieved April (2007); http://www.davixology.com/csound~.html.

[42] C. Roads, *The Computer Music Tutorial* MIT Press, Cambridge, MA, USA, (1996).

[43] C. Roads, *Microsound*, MIT Press, Cambridge, MA, USA, (2001).

[44] Rustak, *WoWWiki, The Warcraft wiki*, retrieved April (2007); http://www.wowwiki.com/UI_Beginners_Guide.

[45] C. Samuel, *Olivier Messiaen, Music and Color: Conversations with Claude Samuel*, Hal Leonard Corporation, (2003).

[46] G. P. Scavone, P. R. Cook, *RTMIDI, RTAUDIO, and a Synthesis ToolKit (STK) Update* In Proceedings of the 2005 International Computer Music Conference, Barcelona, Spain, (2005).

[47] W. Smith, G. Wakefield, *Synecdoche*, January (2007); http://www.mat.ucsb.edu/~whsmith/Synecdoche/.

[48] S. Smoliar, *A Parallel Processing Model of Musical Structures*, report, Department of Mathematics, MIT, Cambridge, USA, (1971).

[49] H. K. Taube, *Notes from the Metalevel: Introduction to Algorithmic Music Composition,* Swets & Zeitlinger, (2005).

[50] B. Truax, "Computer Music Language Design and the Composing Process.", in Emerson S. (ed.) *The Language of Electroacoustic Music,* Macmillan, London, (1986).

[51] E. Varèse, C. Wen-Chung, "The Liberation of Sound", *Perspectives of New Music*, Vol. 5, No. 1, (1966).

[52] Vercoe, D. Ellis, *Real-time CSOUND: Software synthesis with sensing and control.* Proceedings of the ICMC Glasgow, 209-211, (1990).

[53] G. Wang, P. Cook, *ChucK: A Concurrent, On-the-fly Audio Programming Language*, in Proceedings of the International Computer Music Conference, (2003).

[54] T. Wishart, *Audible Design*, Orpheus, UK, (1994).

[55] Xennakis, *Formalized Music*, Pendragon Press, New York, USA, (1992).

[56] Zicarelli, "How I Learned to Love a Program that Does Nothing" *Computer Music Journal* 26, 4, 44-51 (2002).