

LC: A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC
PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS
FOR MICROSOUND SYNTHESIS

HIROKI NISHINO

(Master of Media and Governance), KEIO UNIVERSITY

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

NUS GRADUATE SCHOOL FOR INTEGRATIVE SCIENCES AND ENGINEERING
NATIONAL UNIVERSITY OF SINGAPORE

2014

1. Reviewer: Professor Brad Garton

2. Reviewer: Professor Roger Zimmerman

3. Reviewer: Professor Eric Lyon


Day of the defense: 08th/May/2014

Signature from head of PhD committee:

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

A handwritten signature in black ink, appearing to read 'Hiroki Nishino', written in a cursive style.

Hiroki NISHINO

This thesis is dedicated to my father, the late Hirotoshi Nishino, and my
mother, the late Miyoko Nishino.

Acknowledgements

This thesis would not have been possible without help and support from numerous people.

First and foremost, I am deeply grateful to my supervisor Professor Ryohsei Nakatsu, who has been helping me with his mentorship and invaluable insights on creativity and technology through my Ph.D. study. The encouragement that I have received from his supervision was not just limited to academic research but also for personal issues regarding to life. It was his guidance that led my interest in the creative applications of computer technology to the more academic investigation to contribute to computer music research.

I am so truly thankful to the Thesis Advisory Committee Chair, Professor Steven Miller, especially for his insightful perspective as a professional computer music composer and for the encouragement towards a larger achievement. Having a computer music composer with mastery as one of my advisers has helped me deepen the insight into the history of computer music.

I wish to express my sincerest gratitude to Thesis Advisory Committee member, Professor Naotoshi Osaka for his advices. His proficiency in both musical creation and engineering research encouraged me to the further investigation of any questions in creative practices that could result in a contribution to engineering research. The synergy between creativity and technology seen in his computer music research and compositions has been always a great source of motivation.

I feel incredibly fortunate to have insightful thesis examiners, Professor Brad Garton, Professor Eric Lyon, and Professor Roger Zimmermann.

I am grateful to a number of people who helped me, both for academic matters and personal issues at National University of Singapore, I would

like to thank Professor Philip Moore, Professor Tang Bor Luen, Professor Lawrence Wong, Ms. Wendy Lee, and Mr. Tan Boon Chye for their continuous support during my study at National University of Singapore.

It has also been a great pleasure to work with the people at Interactive and Digital Media Institute, not just for the research inspiration but also for their friendship. I would like to show my gratitude to the Cute Centre director, Professor Ellen Yu-Luen Do, and the researchers, Dr. Koh Sueda, Dr. Yuichiro Katsumoto, Dr. Masaaki Sato, Dr. Kentaro Yasu, Dr. Kelvin Cheng for their insights as researchers in related fields.

I would also like to thank my fellow Ph.D students, Zhu Kening, Wang Xuan, Jeffrey Koh, Roshan Peiris, Kasun Karunanayaka, Nimesha Ranasinghe, Weiquan Lu, Wei Jun, Elham Saadatian, Yoonsoon Choi, Ron Huang, Stefano Fasciani, Srikumar Karaikudi Subramanian and Suranga Nanayakkara. In addition, I am grateful to the people outside Interactive Digital and Media Institute who motivated and inspired me on the subject of art and/or technology during this Ph.D study. I would like to thank Professor Masahiko Inami, Professor Tsuyoshi Natsuno, Professor Annie On Ni Wan, Professor Eunsu Kang, Professor Satoru Tokuhisa, Dr. Hugo Solis, Ms. Karolina Sobecka, Mr. Rodeny Berry, Mr. Kentaro Fujinuma, Mr. Hisashi Ishihara, and Mr. Yuta Nakayama.

Last but not least, as this thesis is also highly motivated by my own interests in artistic creation, I am deeply grateful to my mentors in Gagaku music, Master Sukeyasu Shiba and Ms. Naoko Miyamaru, for deepening my understanding of music and cultural tradition.

Contents

List of Tables	ix
List of Figures	xi
1 Introduction	1
1.1 Problem statement	1
1.2 Contribution	6
1.3 Roadmap	9
2 Background and Motivation: Three Problems in Today’s Computer Music Programming Language Design	11
2.1 The insufficient support for dynamic modification of a computer music program	13
2.1.1 Rapid-prototyping	13
2.1.2 Live-coding	14
2.1.3 The problems in the existing computer music programming languages	17
2.2 The insufficient support for precise timing behaviour and other features with respect to time	19
2.2.1 Precise timing behaviour in non real-time computer music languages and systems	19
2.2.2 Precise timing behaviour in the era of the hybrid computer music systems	19
2.2.3 Precise timing behaviour in the era of stand-alone real-time computer music systems	23

CONTENTS

2.2.3.1	The necessity for precise timing behaviour with sample-rate accuracy	23
2.2.3.2	Timing behaviour in sound synthesis libraries and frameworks	24
2.2.3.3	The use of coroutines in a sound synthesis framework	31
2.2.4	Strongly-timed programming	33
2.2.4.1	Synchronous programming	33
2.2.4.2	ChuckK, a strongly-timed programming language	34
2.2.4.3	Discrete event simulation in FORMULA, coroutines in LuaAV and strongly-timed programming in ChuckK	34
2.2.4.4	Visual computer music programming languages	36
2.2.5	The problems in the existing computer music programming languages	38
2.3	The difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion	41
2.3.1	The unit-generator concept and microsound synthesis techniques	42
2.3.1.1	The unit-generator concept	42
2.3.1.2	Microsound synthesis techniques	44
2.3.2	Abstraction inversion in microsound synthesis programming	45
2.3.2.1	Abstraction inversion	45
2.3.2.2	The microsound synthesis examples in SuperCollider and ChuckK	46
2.3.2.3	The microsound synthesis examples in visual programming languages	54
2.3.2.4	The lack of objects and manipulations for microsound synthesis in the sound synthesis software frameworks	55
2.3.3	The problems in the existing computer music programming languages	64
2.4	The problems as design opportunities	66
3	Design: LC, a Mostly-strongly-timed Prototype-based Computer Music Programming Language that Integrates Objects and Manipulations for Microsound Synthesis	69
3.1	The basic language features	70

3.1.1	The grammar	70
3.1.2	Operators and primitive types	70
3.1.3	Typing and variable scope	74
3.1.4	Control structure	74
3.1.5	Lexical closure	75
3.1.6	Exception handling	77
3.1.7	Tail call optimization	78
3.1.8	Strongly-timed programming	82
3.1.9	Lightweight concurrency and multitasking	82
3.2	The Core Language Features	86
3.2.1	Prototype-based programming	86
3.2.1.1	Prototype-based programming at the level of compositional algorithms	87
3.2.1.2	Prototype-based programming at the level of sound synthesis	94
3.2.2	Mostly-strongly-timed programming and other features with respect to time	103
3.2.2.1	Mostly-strongly-timed programming	106
3.2.2.2	Timed-tagged message communication	107
3.2.2.3	Timing constraints	115
3.2.3	The Integration of objects and manipulations for microsound synthesis	118
3.2.3.1	Objects and manipulations for microsound synthesis	122
3.2.3.2	Microsound synthesis in LC	126
3.2.3.3	The collaboration between microsounds and unit-generators	144
4	Discussion: the Necessity for the Development of LC as a New Language and the Benefits of Its Language Design	167
4.1	The justification of the development of LC as a new computer music programming language	167
4.1.1	The necessity to provide more suitable syntaxes for frequently performed tasks	168
4.1.2	Execution time constraints	168

CONTENTS

4.1.3	Mostly-strongly-timed programming cannot be implemented as library functions	173
4.1.4	The necessity for LC's own compiler and virtual machine	174
4.2	Comparing LC with the existing computer music languages	177
4.2.1	The support for dynamic modification of a computer music system at runtime	177
4.2.1.1	Dynamic modification of a computer music system in the existing computer music languages	178
4.2.1.2	The benefits of LC's language design for dynamic modification of a computer music system	191
4.2.2	The support for precise timing behaviour and other features with respect to time	194
4.2.2.1	Timing behaviour in the existing computer music languages	195
4.2.2.2	Other features with respect to time in the existing computer music languages	196
4.2.2.3	The benefits of LC's language design	202
4.2.3	The difficulty in programming microsound synthesis techniques	206
4.2.3.1	Abstraction inversion in the unit-generator languages	206
4.2.3.2	When black-box abstractions do not benefit	207
4.2.3.3	Microsound objects and manipulations in the existing computer music languages	216
4.2.3.4	The benefits of LC's language design	223
5	Conclusion and Future Work	235
5.1	Conclusion	235
5.1.1	Problems	235
5.1.2	Contribution	237
5.1.3	Conclusion	238
5.2	Future Work	239
5.2.1	Language features.	239
5.2.2	Performance efficiency.	239
5.2.3	Garbage collection	242

References	245
6 Appendix I: Related Publications	257
7 Appendix II: A Brief History of Computer Music Languages and Systems - the Synergy between Technology and Creativity	263
7.1 Early computer music programming languages and systems	264
7.1.1 MUSIC-N languages	264
7.1.2 Other notable early computer music programming languages and systems	271
7.1.2.1 Other Music-N descendant and non Music-N descendant languages	271
7.1.2.2 Computer music programming languages and systems for algorithmic compositions	271
7.2 Real-time computer music programming languages and systems	273
7.2.1 Early live computer music systems (before real-time digital sound synthesis)	273
7.2.2 The emergence of variable-function digital signal processors . . .	274
7.2.3 MIDI-based interactive computer music systems	281
7.2.4 The development of standalone real-time computer music programming languages	283
7.2.5 Software libraries for digital sound synthesis	284
7.2.6 New exploration in computer music programming language design	285
7.2.7 The emergence of mobile platforms	291
7.3 The synergy between technology and creativity	291
8 Appendix III: the Implementation of the Proof-of-concept Prototype of LC	295
8.1 System architecture	295
8.2 LC Editor	295
8.3 LC Virtual Machine	296
8.4 Latency issues	297
8.5 The issues related to the performance efficiency	301
8.5.1 Audio vectors	301

CONTENTS

8.5.2	Parallelism	301
9	Appendix IV: Additional Discussion	303
9.1	The definition of ‘abstraction inversion’ referred in this thesis	303
9.2	The HCI related issues	308
9.2.1	The expected users	308
9.2.2	The conceptual gap between the unit-generator concept and mi- crosound synthesis techniques	309
9.2.3	User interface design	313
9.3	Other miscellaneous issues	314
9.3.1	Popularization	314
9.3.2	Musical practices that LC may be suitable for and may not be suitable for	317

Abstract

Through the design of LC, a new computer music programming language, this thesis contributes to solutions to three problems in today's computer music language design: (1) the insufficient support for dynamic modification, (2) the insufficient support for precise timing behaviour and other desirable features with respect to time, and (3) the difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion.

As the creation process of computer music composition can be highly exploratory in that musicians normally experiment with different compositional and sound synthesis algorithms, better support for rapid-prototyping is considered important. At the same time, recent computer music practices can even involve dynamic modification of a program at runtime, on-the-fly on stage, at both levels of compositional algorithms and sound synthesis. Nevertheless, even the latest computer music languages do not provide a terse and consistent programming model with a sufficient degree of support for dynamic modification, especially at the sound synthesis level. This thesis contributes to this issue by the adoption of prototype-based programming, which is highly dynamic in its nature, at both levels of compositional algorithms and sound synthesis in the language design.

The insufficient support for precise timing behaviour and other desirable features with respect to time is another significant problem in many computer music languages. While the strongly-timed programming concept can achieve precise timing behaviour with sample-rate accuracy by the explicit control of the advance of logical time, a time-consuming task that

hinders the advance of logical time can easily cause the temporary suspension of real-time DSP. This thesis proposes the concept of mostly-strongly-timed programming, which extends strongly-timed programming with explicit switching to asynchronous context, in which a thread can be pre-empted regardless of the synchronization with the advance of logical time; thus, a mostly- strongly-timed program can avoid temporary suspension of real-time DSP by executing time-consuming tasks in the asynchronous pre-emptive context, while maintaining sample-rate accurate timing behaviour of strongly-timed programming. This thesis also discusses the benefits for integrating other desirable features with respect to time, such as timing constraints and time-tagged message communication.

Microsound synthesis programs written in unit-generator languages often involve certain programming patterns, which complicate the implementation to compensate imprecise timing behaviour and the lack of the consideration on microsound synthesis in the abstraction of its underlying sound synthesis framework. Such a symptom can be assessed as abstraction inversion, an anti-pattern that occurs when high-level abstractions must be combined to express a lower-level abstraction. This thesis proposes a novel abstraction for microsound synthesis that integrates objects and manipulations for microsounds in the design, which can collaborate with the traditional unit-generator concept in a complementary style. Together with precise timing behaviour supported by mostly-strongly-timed programming, the abstraction makes it possible to describe microsound synthesis techniques more tersely without involving abstraction inversion.

As above, this thesis contributes to three issues that computer music language research faces today, through the design of LC, a mostly-strongly-timed prototype-based programming language that integrates objects and manipulations for microsounds.

List of Tables

2.1	Discrete-Event Simulation Approach: Simple Telephone Call Centre Simulation (taken from (245, p.16)).	21
2.2	Waveset transformations in the Composer's Desktop Project software (242, p.207).	48
2.3	Nine time scales of music by Roads (242, p.3).	57
2.4	The typical abstraction hierarchy in sound synthesis framework design	58
2.5	Three type constructors in Chronic (56, p.8).	62
3.1	The grammar of LC.	71
3.1	The grammar of LC in EBNF (continued).	72
3.2	The relative precedence levels of operators in LC.	73
3.3	The data types available in LC.	73
3.4	The constructor definition of Exception object in LC	77
3.5	Arithmetic operations on time and duration in LC.	83
3.6	The list of library functions for Table objects.	90
3.7	The list of Patch object's methods	97
3.8	The list of unit-generator's methods.	98
3.9	Samples object.	123
3.9	Samples object (continued).	124
3.10	SampleBuffer object.	126
3.11	The list of library functions related to microsound synthesis in the prototype version of LC.	128
3.12	The list of library functions related to microsound synthesis in the prototype version of LC (continued).	129

LIST OF TABLES

3.13	Waveset transformations in the Composer's Desktop Project software (242, p.207), reproduced from Table 2.2.	140
3.14	The list of UGen's methods for the collaboration between microsounds and unit-generators in LC.	153
3.15	The list of Patch's methods for the collaboration between microsounds and unit-generators in LC.	154
4.1	The six required features for high-level real-time programming as Lee et al. discuss in (179)	198
4.1	The six required features for high-level real-time programming as Lee et al. discuss in (179) (continued).	199

List of Figures

1.1	A waveset harmonic distortion example in SuperCollider.	7
1.1	A waveset harmonic distortion example in SuperCollider (continued). . .	8
1.2	A bitwise operation (bitwise-and) example in Lua.	8
2.1	A picture of live-coding performance	15
2.2	A photo of ReacTable	16
2.3	A Just-in-Time programming example in SuperCollider (320, p.209). . .	17
2.4	The components of a computer music performance system	20
2.5	An example of real-time sound synthesis in STK (rtsine.cpp from STK tutorial program).	26
2.5	An example of real-time sound synthesis in STK (rtsine.cpp from STK tutorial program) (continued).	27
2.6	Two different tasks to be performed concurrently.	27
2.7	An example that combines two tasks (task A and task B).	28
2.8	An example with a scheduler function.	29
2.9	An example of the infite loop with the 1.0 second sleep inside.	29
2.10	A temporal recursion example.	30
2.11	A coroutine example in LuaA (150, p.73).	32
2.12	A LuaAV example (307).	32
2.13	An Esterel example and its specification (37).	33
2.14	A ChuckK program to generate a sine wave, changing its frequency of oscillation every 100 milliseconds. (312, p.43).	35
2.15	A simple ChuckK program, which performs the equivalent task as the Figure 2.12 example in LuaAV.	35
2.16	An Instrument with attack, decay, and vibrato	43

LIST OF FIGURES

2.17	A mutex example with abstraction inversion in Ada (28).	46
2.18	A bitwise operation (bitwise-and) example in Lua (reproduced from Figure 1.2).	46
2.19	A pictorial representation of waveset harmonic distortion.	47
2.20	A waveset harmonic distortion example in SuperCollider (reproduced from 1.1).	49
2.20	A waveset harmonic distortion example in SuperCollider (reproduced from 1.1) (continued).	50
2.21	A pictorial representation of synchronous granular synthesis.	51
2.22	A synchronous granular synthesis example in ChuckK.	52
2.23	Another synchronous granular synthesis example in ChuckK with less memory-leak.	53
2.24	A pictorial representation of waveset inversion	54
2.25	A granular synthesis patch by Richard Dudas (the whole patch)	55
2.26	A granular synthesis patch by Richard Dudas (the inside <i>rgrain2</i> ~ <i>sub-patch</i>)	56
2.27	Object-oriented granulator structure	60
2.28	Event sequence when samples are requested from a granulator	60
2.29	Event sequence when it is time to activate a new grain	61
2.30	A pictorial representation of temporal type constructor examples given by Brandt	62
2.31	A score example with sine beeps in Chronic (56, p.26).	63
3.1	Local variables and global variables in LC.	74
3.2	An example of dynamic-typing and strong-typing in LC.	75
3.3	An example of control structures in LC(left) and its output(right).	76
3.4	The examples of lexical closure in LC.	76
3.5	An example of default parameters/keyword arguments in LC.	77
3.6	The built-in exception hierarchy example in LC (the prototype version).	78
3.7	An example of exception handling in LC.	79
3.7	An example of exception handling in LC (continued).	80
3.7	An example of exception handling in LC (continued).	81
3.8	A simple tail call example in LC.	82

LIST OF FIGURES

3.9	A strongly-timed programming example in LC.	83
3.10	Directly computing output samples without unit-generators in LC. . . .	84
3.11	A simple multi-threading example in LC.	85
3.12	Table object examples in LC(1).	87
3.12	Table object examples in LC(1) (continued).	88
3.13	A Table example in LC(2).	89
3.14	A delegation example in LC.	92
3.15	An object-cloning example in LC.	93
3.15	An object-cloning example in LC (continued).	94
3.16	A duck-typing example in LC.	95
3.17	A simple sine wave oscillator example in LC.	99
3.18	Another sine wave oscillator example in LC.	100
3.19	A patch expression example in LC.	101
3.20	A subpatch example in LC.	102
3.21	A patch-cloning example in LC.	103
3.22	A duck-typing example (for Patch object) in LC.	104
3.22	A duck-typing example (for Patch object) in LC (continued).	105
3.23	A mostly-strongly-programming example in LC.	108
3.23	A mostly-strongly-programming example in LC (continued).	109
3.24	A timed interthread messaging example in LC(1).	112
3.25	A timed interthread messaging example in LC(2).	113
3.25	A timed interthread messaging example in LC(2) (continued).	114
3.26	A thread start-time constraint example in LC.	116
3.27	A patch start-time constraint example in LC.	117
3.28	A timeout example in LC(1).	119
3.29	A timeout example in LC(2).	120
3.30	A timeout example in LC(3).	121
3.31	A timeout example in LC(4).	121
3.32	A <i>Samples</i> object creation example.	122
3.33	An indexed-access to <i>Samples</i> object example.	125
3.34	A <i>SampleBuf</i> object example.	127
3.35	A ReadADC/WriteDAC example.	131
3.36	A synchronous granular synthesis example.	132

LIST OF FIGURES

3.37 A quasi-synchronous granular synthesis example.	133
3.38 An asynchronous granular synthesis example.	133
3.39 A granular sampling example (pitch-shifting).	135
3.39 A granular sampling example (pitch-shifting) (continued).	136
3.40 A granular sampling example (time-stretching).	137
3.40 A granular sampling example (time-stretching) (continued).	138
3.41 A waveset example to reproduce the original sound in LC.	141
3.42 A waveset inversion example in LC.	141
3.43 A waveset distortion example in LC.	142
3.44 A waveset transposition example in LC.	143
3.45 A waveset substitution example in LC.	144
3.46 A waveset harmonic distortion example in LC.	145
3.47 A waveset inversion + waveset transposition example (real-time sound input) in LC.	146
3.48 A waveset harmonic distortion example (real-time sound input) in LC. .	147
3.49 A FFT/IFFT example (cross synthesis).	148
3.50 A PFFT/PIFFT example (cross synthesis).	149
3.51 A PFFT/PIFFT example (time-stretching).	150
3.52 A PFFT/PIFFT example (time-stretching real-time input).	151
3.53 A PFFT/PIFFT example (time-stretching real-time input, with a buffer).152	
3.54 An example of creating Sample objects from the unit-generator's output samples (1)	157
3.55 An example of creating Sample objects from the unit-generator's output samples (2).	158
3.56 An example to create Sample objects from the patch's output samples. .	159
3.57 A granular synthesis example with the pregenerated grains.	160
3.58 A granular synthesis example with on-demand generation of the grains. .	161
3.59 A duck-typing example of 'pread'.	162
3.60 A reverberation example (1).	163
3.61 A reverberation example (2).	164
3.62 A reverberation example (3).	165

LIST OF FIGURES

3.63	A duck-typing example to apply an envelope (by a unit-generator) and an envelope + reverberation (by a patch) to the output of waveset harmonic distortion.	166
4.1	A simple sine wave oscillator example in LC (reproduced from Figure 3.17).	169
4.2	A timed interthread messaging example in LC(1) (reproduced from Figure 3.24).	170
4.3	Another sine wave oscillator example in LC (reproduced from 3.18). . .	171
4.4	A timed interthread messaging example in LC.	172
4.5	An example of context switching between synchronous/non-preemptive context and asynchronous/preemptive context with an execution time constraint.	175
4.6	An example of context switching by library function calls between the synchronous/non-preemptive context and the asynchronous/preemptive context with an execution time constraint.	176
4.7	A prototype-based programming example by Dictionary and Event in SuperCollider.	179
4.8	A prototype-based programming example with chucklib in SuperCollider (320, p.600).	180
4.9	Playing Synth objects in SuperCollider.	182
4.10	Just-in-Time programming example in SuperCollider (320, pp.208-210). .	183
4.11	Creating a proxy object explicitly and changing its source (from (320, p.215)).	184
4.12	Refactoring a synthesis graph at runtime (from (320, p.212)).	184
4.13	Parameter mapping and setting (from (320, p.216)).	185
4.14	A simple example to connect/disconnect the connections in a synthesis graph in ChuckK.	187
4.15	A typing issue in dynamic modification at the sound synthesis level in ChuckK.	188
4.16	An oscillator with amplitude modulation: synthesis graph (left), and equivalent abstract syntax tree (right)	190
4.17	An execution-time constraint example in Impromptu (276).	202

LIST OF FIGURES

4.18	A bitwise operation (bitwise-and) example in Lua (reproduced from 2.18).	208
4.19	A waveset harmonic distortion example in SuperCollider (reproduced from Figure 2.20).	209
4.19	A waveset harmonic distortion example in SuperCollider (continued) (reproduced from Figure 2.20).	210
4.20	Another synchronous granular synthesis example in ChuckK with less memory-leak (reproduced from 2.23).	211
4.21	A synchronous granular synthesis example with a triangle envelope applied to the entire sound output in SuperCollider.	212
4.21	A synchronous granular synthesis example with a triangle envelope applied to the entire sound output in SuperCollider (continued).	213
4.22	Another synchronous granular synthesis example in ChuckK with a triangle envelope applied to the entire sound output.	214
4.23	A waveset harmonic distortion example in Nyquist's SAL programming language.	216
4.24	A macro definition of <i>secrep</i> in Nyquist (open source distribution) - Copyright (c) 2000-2002, by Roger B. Dannenberg).	217
4.25	A simple FFT-based cross synthesizer example.	220
4.26	A simple FFT-based cross synthesizer example (using the different hop sizes for the source a and source b).	221
4.26	A simple FFT-based cross synthesizer example (using the different hop sizes for the source a and source b) (continued).	222
4.27	Underlying pipeline of a generic hybrid synthesis/analysis system in ChuckK audio programming language	223
4.28	A sound synthesis and lowpass filter example in Matlab (205).	224
4.29	A waveset harmonic distortion example in LC (reproduced from Figure 3.46).	226
4.30	A synchronous granular synthesis example (reproduced from Figure 3.36).	227
4.31	A reverberation example (2) (reproduced from Figure 3.61).	228
4.32	A waveset harmonic distortion example in LC (equivalent to Figure 4.23 Nyquist example).	229
4.33	Another cross synthesis example in LC (with the variable hopsizes).	231

LIST OF FIGURES

4.34	An example to create Sample objects from the unit-generator's output samples (1) (reproduced from Figure 3.54).	232
4.35	A duck-typing example to apply an envelope (by a unit-generator) and an envelope + reverberation (by a patch) to the output of waveset harmonic distortion (reproduced from Figure 3.63).	233
7.1	An IBM 704 computer	265
7.2	An instrument with attack, decay, and vibrato	268
7.3	An instrument with attack, decay, and vibrato	268
7.4	VAX 11/750	269
7.5	DEC VT-100 Terminal	270
7.6	A commnd line example in CARL (213).	271
7.7	A MINC program example (taken from STRUM1.sco, which is a part of the RTcmix 4.0 package released under GPL license).	272
7.8	An IRCAM/Sogitec Real-Time Digital Signal Processor 4X's circuit boards	276
7.9	An example of a Patcher program	278
7.10	An IRCAM Signal Processing Workstation (left) and an i860 board (right)	279
7.11	A Max/FTS patch example	280
7.12	A screenshot of AUTOBUSK	282
7.13	A screenshot of SuperCollider	287
7.14	A screenshot of Impromptu	288
7.15	A screenshot of miniAudcle	289
8.1	The overall system architecture of LC (the proof-of-concept prototype) .	296
8.2	A screenshot of LC Editor (the proof-of-concept prototype)	297
8.3	The implementation to handle everything within the audio callback function	299
8.4	How the audio computation is triggered	300
8.5	The implementation to perform DSP in a real-time thread	300
9.1	Classic synthesis techniques classified according to their principles of realization	310
9.2	The examples of the descriptions on frequency-domain synthesis techniques in Csound book (52).	311

LIST OF FIGURES

9.3 The examples of the descriptions on microsound synthesis techniques in Microsound (242).	311
---	-----

1

Introduction

1.1 Problem statement

Even since the earliest era in the history of computer music, programming languages tailored for computer music have been playing a significant role in both academic research and artistic creation. Computer musicians and researchers still show considerable interest in computer music languages as primary tools for both research and creation; research on computer music languages is still very active as one of the central topics in the computer music community even today.

Throughout the history of computer music, computer music languages have been continuously evolved by researchers and engineers in the field, making the best use of the available computer technology of the time and being influenced from achievements in programming language research.

For instance, the development of faster processors made it possible to compute digital signals fast enough for real-time use, and computer music languages began to take real-time sound synthesis into consideration as one of the most important design criteria. The popularization of object-oriented programming also had a significant influence on computer music language design. Textual computer music languages are often designed with the object-oriented programming paradigm and even visual computer music programming languages adopt the concept in language designs today; the unit-generator concept is also extended with the object-oriented programming paradigm and the features of object-oriented programming, such as the encapsulation of the data and

1. INTRODUCTION

methods to control behaviour, are quite common today.

In addition to the influence of the advance of computer technology and programming language research, computer music language design has been significantly influenced by needs arising among artists, which may be specific to computer music. For example, the demands for a computer music language that is user-friendly, even to computer music composers without expertise in programming, led to the development and prosperity of visual computer music languages such as Max/MSP and PureData. For another example, as the creative exploration by composers and sonic artists in interactive music composition and interactive installations increased, computer music languages began to take the capability of interaction beyond the traditional score-orchestra model into consideration as one of the essential language features.

Thus, both the advancement in computer technology and programming language research and the domain-specific needs in computer music have motivated the design and development of new computer music languages throughout the history of computer music.

This Ph.D thesis addresses three problems that computer music programming language design faces in our decade: (1) the insufficient support for dynamic modification of a computer music program, (2) the insufficient support for precise timing behaviour and other desirable features with respect to time, and (3) the difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion; these issues are of significant importance when considering the emergence of new sound synthesis techniques and novel creative practices in computer music in the last several decades. The development of a new computer music programming language that overcomes such issues can benefit the further research in computer music programming language design and the creative explorations by computer musicians of our time.

The insufficient support for dynamic modification of a computer music program. The creation process of a computer music composition can be quite similar to rapid-prototyping, as “audio programming, in both computational acoustics research and in music composition and performance, is necessarily an experimental and empirical process; it requires rapid experimentation, verification/rejection/workshopping of ideas

and approaches, and takes the forms of both short-term and sustained prototyping” (312, p.3). Composers usually experiment with various different sound synthesis and compositional algorithms during the creation process, similar to the way programmers do with their programs in the process of rapid-prototyping.

On the other hand, recent computer music practices often call for dynamic modification of a computer music program that is already being executed. For instance, in live-coding performance (43)(62)(77)(212)(216), performers write and modify computer music programs on-the-fly on stage. Similarly, in the performance that involve ‘dynamic-patching’ (160)(162), an instrument is built and modified while it is playing, by connecting and disconnecting sound synthesis modules.

Broadly speaking, the degree of the support for such dynamic modification of a computer program can be significantly limited by the language design. Significant demands exist for more dynamic computer music languages in today’s computer music research and practices.

The insufficient support for precise timing behaviour and other desirable features with respect to time. As computer music is a time-based art form, precise timing behaviour can be quite important in computer music programming. Compared to visual presentation, in which 60 frames-per-second is good enough for human visual perception, human auditory perception is far more sensitive to timing and a high degree of precision is required both at the rhythmic level and the audio level.

At the rhythmic level, Lyon discusses that “even when the amount of deviation from sample accuracy is not clearly noticeable at a rhythmic level, it may still have an undesirable musical effect. For example, a pulsation may feel not quite right when there are a few 10s of milliseconds of inaccuracy in the timing from beat to beat” and “smaller inaccuracies, though rhythmically acceptable, can still cause problems when sequencing sounds with sharp transients, since changes in alignment on the order of a couple of milliseconds will create different comb filtering effects as the transients slightly realign on successive attacks” in (194). On the other hand, sample-rate accuracy in timing behaviour is necessary to accurately perform some sound synthesis techniques; as discussed in the next section, many microsound synthesis techniques require precise timing behaviour with sample-rate accuracy to render the sound output. Imprecise

1. INTRODUCTION

timing behaviour can result not just in theoretically inaccurate output, but often in clearly a noticeable difference to human auditory perception.

Furthermore, many computer music programming languages still lack desirable features with respect to time, such as timing constraints and timed communications. As such features are almost essential to real-time programming languages, computer music programming languages of our time should be equipped with such features with respect to time, together with sample-rate accurate precise timing behaviour.

The difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion. Microsound synthesis techniques were rapidly popularized in computer music practices in the last several decades. Generally speaking, in microsound synthesis techniques, short sound particles (or microsounds) that overlap-add onto others constitute the entire sound. Normally, microsound synthesis techniques require sample-rate accuracy in the scheduling of microsounds to render the sound output as theoretically expected. While most computer music languages provide various unit-generators for microsound synthesis techniques as built-in objects, such a strategy to encapsulate the algorithms of microsound synthesis within the unit-generators significantly limits what users can explore in the domain of microsounds, as the users can not experiment beyond the functionalities and interfaces of the unit-generators; the extension and modification of built-in unit-generators require a certain level of expertise in programming skill, which hardly can be expected of the end users.

Yet, while it is still possible to write microsound synthesis programs within the existing computer music languages, creative exploration in microsound synthesis still may be hindered by some obstacles. Because of the lack of direct counterpart objects to microsounds in the unit-generator concept, each microsound must be normally modelled as a note-level object. To make matters worse, to compensate imprecise timing behaviour in many languages, it is required to take a special care in scheduling, such as the use of a library function so to schedule note-level events ahead of the actual timing. Figure 1.1 is an example of waveset harmonic distortion in SuperCollider (320) and describes a typical example of such a programming pattern. While we describe waveset harmonic distortion and this example in more detail, it should be noted that

each microsound is modelled as a note-level object (defined between line 06-13) and library objects (*Pbind* and *Ppar*) are involved in this implementation; programming microsound synthesis techniques within a unit-generator programming language often involves such cumbersome programming patterns with library functions/objects, even when the synthesis techniques to be performed are conceptually very simple. This would make creative exploration by computer musicians harder in the domain of microsound synthesis.

Such a situation can be considered an anti-pattern called *abstraction inversion*, which “occurs when a programmer is forced to use a combination of higher-level abstractions to express a lower-level abstraction” (28). For instance, the lack of the bitwise operators in the early versions of the Lua is one of the widely-known examples of abstraction inversion. While the bitwise-and operation can be performed just by using ‘&’ in C or C++ (e.g., by the expression such as ‘ $a \ \& \ b$ ’), one has to perform the same operation by writing the code in the early versions of Lua, because of the lack of bitwise operators. Figure 1.2 describes a bitwise-and example in Lua.

In a traditional unit-generator-based sound synthesis framework, the lack of objects and functions that can directly represent microsounds and related manipulations can lead to a similar problem. In a unit-generator language, as shown in the SuperCollider example in Figure 1.1, the combination of higher-level abstractions is a certain programming pattern described above, which involves note-level objects and library functions (or library objects) for scheduling. Generally speaking, microsounds can be considered to belong to a lower-level than notes, as the entire sound output of a microsound synthesis technique is conceptually a note-level object, which consists of many microsounds; While many computer music languages still do not provide simple and effective means for it, scheduling is also conceptually a very simple operation.

Thus, the difficulty in microsound synthesis programming can be viewed as a problem of abstraction inversion, caused by the lack of objects and manipulations that can directly represent microsounds and related manipulations. It is desirable to be removed so that further creative exploration by computer musicians can be facilitated in the domain of microsounds; there is a strong necessity to investigate a more suit-

1. INTRODUCTION

able abstraction that can provide a terse programming model for microsound synthesis techniques.

1.2 Contribution

These three issues described above provide design opportunities for a new computer music language.

LC adopts the concept of prototype-based programming both at both levels of the compositional algorithms and sound synthesis, for the better support of runtime dynamism. Prototype-based programming is significantly flexible and robust against dynamic modifications at runtime, which are very favourable for both rapid-prototyping and live-coding.

While the strongly-timed programming concept can achieve precise timing behaviour with sample-rate accuracy by the explicit control in the advance of logical time, a time-consuming task that hinders the advance of logical time can easily cause a temporary suspension of real-time DSP, as the output samples in a strongly-timed program cannot be computed without the advance of logical time. LC proposes the mostly-strongly-timed programming concept, which extends the strongly-timed programming concept with explicit switching between non-preemptive/synchronous context and preemptive/asynchronous context. Such extension allows time-consuming tasks to be preempted so that they do not hinder the advance of logical time, while maintaining sample-rate accurate timing behaviour of strongly-timed programming. Furthermore, LC also introduces other desirable features such as timing constraints and timed-tagged message communications, which are beneficial for computer music programming.

LC also provides a novel abstraction for its underlying sound synthesis framework, which directly integrates objects and manipulations for microsound synthesis. Supported by such framework design and sample-rate accurate timing behaviour provided by mostly-strongly-timed programming, microsound synthesis algorithms can be tersely described and compute accurately in LC, without involving abstraction inversion as seen in unit-generator languages.

```

01: Server.default = s = Server.internal;
02: s.boot;
03: w = Wavesets.from("sound.aif");
04: (
05:   b = w.buffer;
06:   SynthDef(\wvst0, {
07:     arg out = 0, buf = 0, start = 0, length = 441,
08:     playRate = 1, sustain = 1, amp = 1;
09:     var phasor = Phasor.ar(rate:playRate, start:0, end:length) + start;
10:     var env = EnvGen.ar(Env([amp, amp, 0], [sustain, 0]), doneAction: 2);
11:     var snd = BufRd.ar(1, buf, phasor) * env;
12:     OffsetOut.ar(out, snd);
13:   }).add;
14: )
15: (
16:   var numOfWeeksets = w.lengths.size;
17:   var original = Pbind(
18:     \instrument, \wvst0,
19:     \startWs, Pseries(0, 1, numOfWeeksets),
20:     \numWs, 1,
21:     \playRate, 1,
22:     \bufnum, b.bufnum,
23:     \repeats, 1,
24:     \amp, 1,
25:     [\start, \length, \sustain], Pfunc( { |ev|
26:       var start, length, wsDur;
27:       #start, length, wsDur = w.frameFor(ev[\startWs], ev[\numWs]);
28:       [start, length, wsDur * ev[\repeats] / ev[\playRate].abs]
29:     }),
30:     \delta, Pkey(\sustain)
31:   );

```

Figure 1.1: A waveset harmonic distortion example in SuperCollider.

1. INTRODUCTION

```
32: var octup = Pbind(  
33:   \instrument, \wvst0,  
34:   \startWs, Pseries(0, 1, numOfWavesets),  
35:   \numWs, 1,  
36:   \playRate, 2,  
37:   \bufnum, b.bufnum,  
38:   \repeats, 2,  
39:   \amp, 1,  
40:   [\start, \length, \sustain], Pfunc( { |ev|  
41:     var start, length, wsDur;  
42:     #start, length, wsDur = w.frameFor(ev[\startWs], ev[\numWs]);  
43:     [start, length, wsDur * ev[\repeats] / ev[\playRate].abs]  
44:   } ),  
45:   \delta, Pkey(\sustain)  
46: );  
47:  
48: Ppar([original, octup]).play(SystemClock);  
49: )
```

Figure 1.1: A waveset harmonic distortion example in SuperCollider (continued).

```
01: function bit.and(x, y)  
02:   local digit = 1  
03:   local ret = 0  
04:   local limit = x > y and x or y  
05:   while digit <= limit do  
06:     if (x % (digit * 2)) >= digit and (y % (digit * 2)) >= digit then  
07:       ret = ret + digit  
08:     end  
09:     digit = digit * 2  
10:   end  
11:   return ret  
12: end
```

Figure 1.2: A bitwise operation (bitwise-and) example in Lua.

Thus, this thesis contributes to a solution for the above three issues in computer music language design research in our decade through the design and development of LC. The contributions are made by (1) the adoption of the programming concept for general-purpose languages to a domain-specific problem (the application of the prototype-based programming concept at both levels of compositional algorithm and sound synthesis), (2) the proposition of a new programming language concept (the mostly-strongly-timed programming concept, which extends the strongly-timed programming concept with explicit switch between synchronous context and asynchronous context), and (3) the novel approach to the sound synthesis framework design (the integration of the objects and manipulations for microsound synthesis in the sound synthesis framework).

1.3 Roadmap

The rest of this thesis is organized as follows. In the next chapter (Chapter 2), the three problems in computer music language design, which hinder creative exploration in our decade, are described. These problems are addressed as a significant opportunity for the development of a new computer music language. Then, the language design of LC, a new computer music language, is described in the following chapter with various examples in Chapter 3. The discussion on the benefits of the design of LC is presented in comparison with the other existing languages in Chapter 4. The conclusion and future work are placed in the last chapter (Chapter 5).

Additionally, Appendix I lists up the related publications by the author with brief descriptions regarding how each publication relates to this thesis. Appendix II describes a brief history of computer music languages and systems to endorse the argument that the problems in creative practices can lead to contributions to academic research. Appendix III provides brief descriptions of the implementation of the proof-of-concept prototype of LC. Appendix IV also provides some additional discussions on miscellaneous issues, which are not within the scope of this thesis, but may be desirable to consider for the further investigation.

1. INTRODUCTION

2

Background and Motivation: Three Problems in Today's Computer Music Programming Language Design

While the advance of computer technology has motivated the development of new computer music languages and systems, the aspiration for new artistic forms and the problems discovered through such creative practices also have suggested the limitations of existing computer music languages and systems, which also provide significant design opportunities for new languages and systems.

Even in the very early stages of computer music research and creation, the synergy between technology and creativity played a significant role in the development of computer music languages and systems. Shortly after the first digital sound synthesis program was developed by Mathews and his colleagues at the Bell laboratory (204) in the late 1950s, the researchers began designing special-purpose languages tailored for computer music with domain-specific abstractions. Two core abstractions for computer music languages were established in this era, as seen in MUSIC-III (developed in 1960) (94, p.26)(243) and these two core abstractions, the unit-generator concept and the score-orchestra model, are still widely applied not just to MUSIC-N family languages such as Csound (305), but also to many other recent computer music languages.

The desire to perform computer music compositions in real-time, especially those

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

generated by algorithms, led to the development of hybrid computer music systems. A hybrid computer music system in this era normally consists of a mini-computer and external synthesizer hardware. For instance, the GROOVE systems (203), the Yale synthesizer (117), and MUSYS (134) were systems of this kind and all of them were developed in the early 1970s; such efforts founded the basis for interactive music systems in the following decade when the MIDI interface standard (23) emerged in 1980s.

When special DSP hardware made real-time digital sound synthesis possible as seen in the series of IRCAM hardware (4A (229), 4B (8), 4C (98), 4X (214), IRCAM musical workstation (187)), Kyma/Platpus (258)(259), and MARS workstation (20)(72), the creative practices made the researchers of the time aware of the necessity to develop a more user-friendly flexible programming environment for rapid-prototyping and end-user programming under such environments; Max (230)(231) is possibly one of the most notable programming languages of this kind, which was originally developed for such special hardware in this era. Its concept of visual programming is still widely adopted in many widely-used computer music languages today.

Around the 1990s, when real-time sound synthesis was realized even on stand-alone computers without the assistance of external hardware, the researchers began developing the stand-alone real-time versions of existing computer music languages, such as the real-time version of Csound (305), Max/MSP (327), and RTcmix (122)(297), together with new computer music languages such as PureData (232) and SuperCollider (210).

Creative practices that were developed in the previous decades became new criteria for computer music language design, not separately as before, but altogether; computer music language researchers of the time began researching language design that can support algorithmic/interactive music systems, real-time digital sound synthesis, and new interfaces for musical expressions and the like, which used to be handled in different programming environments, in just one integrated environment.

Thus, computer music systems and languages have been fostered through synergy between technology and creativity. The readers who are interested in such an aspect of computer music programming languages and systems are recommended to read Appendix II, which provides a more detailed description of the historical development of computer music languages and systems with some emphasis on such synergy between

2.1 The insufficient support for dynamic modification of a computer music program

technology and creativity. Appendix II also provides more detailed information on the computer music languages and systems mentioned above.

Upon the perspective as above, it can be considered that the problems found in creative practices suggest, not just the limitation of the existing computer music languages, but also the necessity for the further research in computer music language design. This chapter describes three problems in today's computer music programming language design, which hinder creative exploration in computer music: (1) the insufficient support for dynamic modification of a computer music program, (2) the insufficient support for precise timing behaviour and other features with respect to time, and (3) the difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion. These problems are addressed as a significant motivation for the design and development of a new programming language.

2.1 The insufficient support for dynamic modification of a computer music program

2.1.1 Rapid-prototyping

Generally speaking, the programming concepts and paradigms that a programming language built upon have significant influences on how much the programming language can support rapid-prototyping of an application.

For instance, in (222), Ousterhout categorizes programming languages into two categories, *system programming languages* and *scripting languages*¹, and discusses the benefit of scripting languages for rapid application development in the detail.

Ousterhout also discusses how dynamically-typed scripting languages are beneficial for rapid application development, because “implementation inheritance causes the same intertwining and brittleness that have been observed when goto statements are overused” and “as a result, OO systems often suffer from complexity and lack of reuse” (222).

¹In (222), Ousterhout lists Pascal, C, C++, Java and the like as the examples of system programming languages, which are frequently used to develop software such as operating systems and database. On the other hand, the examples of scripting languages listed include Perl, Python, Unix shells and such, which are considered beneficial to ‘glue’ the existing software libraries together.

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

The issue of rapid-prototyping is a topic of significant interest in computer music language design today. Wang describes the rapid-prototyping nature of computer music programming as follows.

“Audio programming, in both computational acoustics research and in music composition and performance, is necessarily an experimental and empirical process; it requires rapid experimentation, verification/rejection/workshopping of ideas and approaches, and takes the forms of both short-term and sustained prototyping” (312, p.3).

In other words, computer music programming is highly exploratory in its nature and better support for rapid-prototyping is necessary for the facilitation of creative exploration in computer music; many papers on the IRCAM Music Workstation clearly state such support for rapid-prototyping as one of the design goals in the development of their computer music languages and systems (93)(95)(96)(186). Many recent works still discuss this issue of rapid-prototyping as their motivations as seen in STK (82), Marsyas(298), and CLAM (12).

2.1.2 Live-coding

The emergence and popularization of live-coding practices are also casting significant questions in computer music language design today. Figure 2.1 shows a picture of a live-coding performance by Wrongheaded (Matthew Yee-King and Click Nilson). As live-coding performances normally involve coding/modification activity on-the-fly on stage, the support for dynamic modification of a computer music program at runtime is an important feature to consider in language design.

Recent research on computer music programming language design often discusses the necessity for computer music programming languages with features to support live-coding activity. For instance, Wang puts a significant focus on live-coding in the design of his computer music language, ChuckK, and describes one of the central ideas as “a programming paradigm and run-time environment that allow on-the-fly programming, enabling dynamically modifiable programs for performance and experimentation” (312, p.39).

Sorensen and Brown, the main contributors of Impromptu computer music language, emphasize that they designed Impromptu programming language “to provide

2.1 The insufficient support for dynamic modification of a computer music program



Figure 2.1: A picture of live-coding performance - by Wrongheaded (Matthew Yee-King and Click Nilson). Photo by Dave Griffiths used under Creative Commons BY-SA 2.0.

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

a dynamic, real-time, multi-user platform capable of supporting the creation, modification, distribution and evaluation of source code in live performance” and “highly dynamic, real-time environment is ideal for crafting media art works” (61).

Such demand for dynamism is considered of importance, not just at the level of compositional algorithms, but also at the level of sound synthesis. One good example is the concept of *dynamic patching* proposed by Kaltenbrunner and his colleagues in a series of papers on ReacTable (Figure 2.2) (159)(160)(163). In dynamic patching, the objects for sound synthesis and signal processing are automatically reconnected, according to a set of rules, such as proximity between objects at runtime during a live performance (162).



Figure 2.2: A photo of ReacTable - by Daniel Williams. (This file is licensed under the Creative Commons Attribution-Share Alike 2.0 Generic license.)

The dynamic modification at the sound synthesis level can also be seen in the context of live-coding. For instance, *Pd~graz*, a group that consists of IOhannes m zmölnig and his colleague musicians, perform live-coding pieces in PureData, which involves the dynamic modification at the sound synthesis level¹. Rohrerhuber and his colleagues devel-

¹One of the known performances by *Pd~graz* is the piece ‘*Blind Date*’ at International Computer

2.1 The insufficient support for dynamic modification of a computer music program

```
01: p = ProxySpace.push;
02: ~x = { SinOsc.kr(4) };
03: ~y = { SinOsc.kr(13) };
04: ~z = { SinOsc.ar(~x * ~y % 0.4 * 500 + 600) * 0.2 };
05: ~z.play;
06:
07://now ~x and ~y can be replaced
08: ~x = { SinOsc.kr(0.4) };
09: ~y = { SinOsc.kr(1.3) };
10:
11: p.clear(2).pop; //release environment (2 sec fadeout)
```

Figure 2.3: A Just-in-Time programming example in SuperCollider (320, p.209).

oped the *just-in-time programming library* to extend SuperCollider with the capability of dynamic modification of sound synthesis graphs¹(22)(247)(320, chapter 7). Figure 2.3 shows an example code of the just-in-time programming library described in (320, p.209).

2.1.3 The problems in the existing computer music programming languages

As today’s computer music practices can involve dynamic modification of a computer music program to a significant degree, some recent computer music languages clearly state the intentions to support such dynamism, as seen in SuperCollider, Impromptu and Chuck. Yet, while both SuperCollider and Impromptu are highly dynamic at the compositional algorithm level, these languages seem to lack a terse and consistent programming model for dynamic modification at the sound synthesis level.

For example, SuperCollider depends on the just-in-Time programming library for the dynamic modification of a sound synthesis graph, yet it requires the use of *proxy objects* as seen in Figure 2.3. As Impromptu entirely depends on Apple’s Audio Unit framework for sound synthesis, it is not very expressive at the sound synthesis level in comparison with the unit-generator languages; In Impromptu, the dynamic modification at the sound synthesis level is limited only within what the Audio-Unit framework

Music Conference, Queens University Belfast, UK, 2008.

¹Just-in-time programming library can be also used for the dynamic modification of tasks, as described in (22, p.119) and (320, p.217)

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

allows.¹

On the other hand, in *ChuckK*, the dynamic modification of unit-generators can be performed relatively easily by using *ChuckK operator* (`=>`) and *UnChuckK operator* (`=<` and `!=>`). However, as *ChuckK* is designed as a statically-typed class-based language without first-class functions, replacing a unit-generator with another unit-generator with of a different type may cause various typing-related problems². Strong-typing may also cause similar problems at the compositional algorithm level.

Visual programming languages (e.g., *Max/MSP* and *PureData*) allow dynamic modification of a synthesis graph to a considerable degree. The connections between sound synthesis and signal processing objects can be freely disconnected and reconnected. The creation of a new object is also easy. Yet, under a visual programming environment, the modification of compositional algorithms and synthesis graphs can take more time compared to textual programming languages, as programming activity can require more user actions with the mouse and the keyboard.

Furthermore, visual programming languages are highly interactive and normally respond to any modification immediately as it is made; while this feature of *direct manipulation*³ may be desirable in many cases, it may lead to unwanted musical output in a live-coding performance when the intention is not to intermediately reflect any modification to the sound output until finished with all the intended modifications.

¹*Extempore* (277), a successor internal DSL of *Impromptu* built on Scheme, is currently under development by Sorensen and Swift so that it provides more expressiveness at the sound synthesis level. In *Extempore*, a user can define a callback function for DSP and the instruments with sound effects can be defined; yet the former may be too low level as an end-user programming language and the latter may have some limitations in its application domain; In the latter model, a note-level object can be defined as a function. An instrument level object plays such note-level objects with sound effects. While this model may be beneficial for some application domains, it may not be appropriate for other domains, such as dynamic-patching, in which it is required to modify a sound synthesis graph dynamically.

²For instance, suppose that one wants to replace a *SinOSC* object (sine wave oscillator) with a *SawOsc* object (sawtooth wave oscillator). As these two belong to different classes, one cannot simply assign a new instance of a *SawOsc* to a variable that is typed as *SinOSC*. While these two classes share the parent class *UGen*, using this parent class may cause another problem when one wants to access a field or a method specific to an inherited class; typecasting to the child class must be involved, yet such down-casting is unsafe.

³Cook summarizes Shneiderman's three principles of direct manipulation systems (266) in (81) as follows: (1) continuous representation of the objects of interest, (2) physical actions or presses of labelled buttons instead of complex syntax, (3) rapid incremental reversible operations whose effect on the object of interest is immediately visible.

2.2 The insufficient support for precise timing behaviour and other features with respect to time

Thus, while recent creative practices can involve dynamic modification of a computer music system to a considerable degree at both levels of compositional algorithms and sound synthesis, the existing computer music programming languages exhibit a certain degree of drawback, at least at either the compositional algorithm level or the sound synthesis level.

2.2 The insufficient support for precise timing behaviour and other features with respect to time

This section discusses the insufficient support for precise timing behaviour and other desirable features with respect to time in computer music languages. While many non real-time, non interactive computer music languages traditionally have provided precise timing behaviour in logical time, computer music systems today require capabilities of both real-time sound synthesis and interactivity. Furthermore, timing behaviour with sample-rate accuracy is becoming a significant criterion in computer music practices especially for microsound synthesis. In this section, we discuss this issue related to ‘time’ in computer music systems and languages.

2.2.1 Precise timing behaviour in non real-time computer music languages and systems

Non real-time/non-interactive computer music languages and systems can easily achieve precise timing behaviour. While there exists the distinction between *audio-rate* and *control-rate* (53, p.468), this was a problem due to the implementation of a sound synthesis framework that involves *audio-vectors* (53, p.467) in audio computation, musical events can be processed with sample-rate accurate timing precision just by setting the control-rate to the same rate as the audio-rate.

2.2.2 Precise timing behaviour in the era of the hybrid computer music systems

As live computer music began attracting interest among computer musicians, the issue of precise timing behaviour in real-time arose. Unlike non real-time/non-interactive computer music systems, interactive computer music systems must interact with musical events, compositional algorithms, and the user inputs, while also outputting the

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

generated musical events in real-time; thus, timing behaviour with better precision in real-time became an issue of significance for computer music programming languages and systems in this era.

The work related to FORMULA (Forth Music Language) (16) is of particular interests in this sense, as it addresses the challenges in timing precision in early interactive music research. In addition, FORMULA implemented several features with respect to time, such as nested timing constraints and background tasks, which seem to be lacking in many recent computer music languages; thus, FORMULA would be an interesting design exemplar to revisit even today.

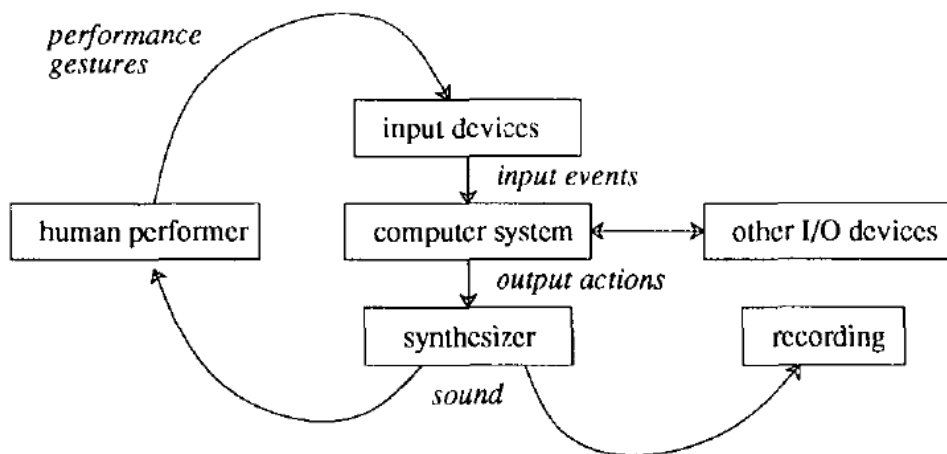


Figure 2.4: The components of a computer music performance system - as described by Anderson and Kuivila in ‘A System for Computer Music Performance’ (16).

As shown in Figure 2.4, the typical computer music performance system of this era consisted of a computer and an external synthesizer peripherals. Stand-alone real-time sound synthesis solely on a computer was still not possible. FORMULA adopted the concept of *discrete event simulation* (30)(180) to a computer music system of this kind and processed musical events in logical time rather than in real-time. Robinson explains the concept of discrete event simulations as follows:

“In discrete-event simulation, only the points in time at which the state of the system changes are represented. In other words the system is modelled as a series of events, that is, instants in time when a state-change occurs. Examples of events are

2.2 The insufficient support for precise timing behaviour and other features with respect to time

Time	Event
3	Customer arrives Operator 1 starts service
6	Customer arrives Operator 2 starts service
8	Operator 1 completes service
9	Customer arrives Operator 1 starts service
11	Operator 2 completes service
12	Customer arrives Operator 2 starts service
14	Operator 1 completes service
15	Customer arrives Operator 1 starts service
17	Operator 2 completes service
18	Customer arrives Operator 2 starts service
20	Operator 1 completes service
21	Customer arrives Operator 1 starts service
23	Operator 2 completes service
24	Customer arrives Operator 2 starts service

Table 2.1: Discrete-Event Simulation Approach: Simple Telephone Call Centre Simulation (taken from (245, p.16)).

customer arrives, a customer starts receiving service and a machine is repaired. Each of these occurs at an instant in time.” and “this obviously requires a time-slicing simulation to be carried out first” (245, p.15).

Table 2.1 shows an example of discrete event simulation of a simple telephone call centre simulation by Robinson(245, p.16). In this example, a customer arrives every three minutes and each operator (operator 1 & 2) requires five minutes to complete the services. In this simulation, each time-slice corresponds to one minute. As shown, “the system is modelled as a series of events” (245, p.15) in logical time and some events are treated as if they actually occurred precisely at the same moment. For instance, at minute three, a customer arrives and operator 1 starts service right at the same instant in logical time. Obviously, in real-time, these events would have some lag in the real world, as a customer must arrive first to be served by operator 1 and these events rarely happen exactly at minute three. Yet, discrete event simulation simplifies real-world by

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

a time-slicing simulation.

FORMULA applies this concept of discrete event simulation to computer music systems, and its system executes tasks in logical time, which is time-sliced as above. As tasks and events are scheduled in logical time, it is not affected by the passage of time in real time. These tasks and events can be scheduled and executed with precise timing in logical time. Thus, musical events can be handled with precise timing in logical time, which is time-sliced by a certain time-unit.

However, as a real-time computer music system, it is still necessary to manage the coordination between the system's internal logical time with real time. In the implementation of FORMULA, logical time is advanced ahead of real time to achieve better timing precision and FORMULA's operating system buffers the generated output events, which are associated with its own output timing in logical time. When real time actually catches up with the given timing, the buffered events are sent to the external hardware at once. By such a mechanism, even when two different events are generated with different timings in real time, if the timing of the output in logical time of these two are the same, FORMULA can minimize the difference in the output timing of these two events in real time when these events are sent to the external hardware, since these output events are buffered within the system and sent out together when real time catches up with the time stamp.

Instead, even when the events from the input devices are processed right after the system receives it, there can be some delay in response. For instance, if a system is set up to advance the logical time 100 ms before real time at maximum, even if the system generates a reaction to some input event immediately after it is received, the output can be delayed 100 ms at maximum in real time to be output to the external hardware as it is buffered until real time catches up its time stamp. Hence, the difference between logical time and real time should be minimized as much as possible to achieve faster interaction to the users, while such strategy contributes to the synchronization the output events from a system to its external synthesizer.

A similar approach to achieve better timing precision by the utilization of internal logical time can be often seen in different computer music languages and systems in the same era, (e.g., HMSL (226), Moxie (76) and the CMU MIDI Toolkit (88)).

2.2 The insufficient support for precise timing behaviour and other features with respect to time

As described by Anderson and Kuivila, the requirements for timing precision in FORMULA’s operating system (and other research in the same era listed above) were significantly lower than what is required today. For instance, in FORMULA’s operating system, input frequency for continuous gestures was considered “at a low rate (less than 200 samples per second)”, and the output rate for I/O devices was considered “usually low (less than 100 commands per second)” (16). While they also mention that the output rate may be higher for continuous note parameters such as timbre, what they refer to is the MIDI interface standard (23), the bandwidth of which is just 31.25K bits-per-second (= 4000 Bytes per second)¹.

The low communication rate of MIDI was also a significant problem, even when a computer music system was designed to schedule the output events with as precise timing as possible. A MIDI note-on event consists of three bytes; even if the system scheduled 13 notes simultaneously, there can be a gap of about 9.75 ms between the first output note and the last output note (13 notes * 3 bytes = 49 bytes and 49 bytes / 4000 bytes per second = 0.00975 Sec). To fill such a gap due to the bandwidth, additional care had to be taken, (e.g., buffering the events also on the side of a MIDI synthesizer).

2.2.3 Precise timing behaviour in the era of stand-alone real-time computer music systems

2.2.3.1 The necessity for precise timing behaviour with sample-rate accuracy

After stand-alone real-time sound synthesis was realized on personal computers, the issue of precise timing behaviour arose again, but in a different context. For instance, while Nyquist (89) was developed as a non real-time sound synthesis language², its design, which seamlessly integrates compositional algorithms and sound synthesis, brought a problem of synchronization between compositional algorithms and sound synthesis with sample-rate accuracy. Such a sample-rate accuracy in timing behaviour is also considered important for realizing musical compositions in the microsound time

¹This bandwidth is the MIDI 1.0 specification, which was widely used around that time.

²The later version of Nyquist can render the output sound in real time, yet it is still not a real-time, interactive computer music programming language in the sense that SuperCollider or Max/MSP is; the problem domain that Nyquist focuses on differs from the development of interactive computer music applications.

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

scale, for instance, ones by Xenakis (325), Wishart (322), and Roads (242).

As seen in computer music languages with the score-orchestra model (e.g. Csound), the synchronization with sample-rate accuracy between compositional algorithms and sound synthesis can be easily achieved for non real-time/non-interactive usage.

However, in a real-time interactive computer music system, such synchronization can be a more complicated issue. Unlike non real-time/non-interactive applications, computer music systems and languages must support interactive compositional algorithms, performing various tasks simultaneously. Musical events are not deterministic in an interactive music and a program must react to the data and events coming from the input devices. Furthermore, tasks may be activated or deactivated dynamically at runtime. Such non-deterministic nature of real-time computer music interactions complicates the issue of timing precision excessively, especially since the output samples must be computed in real time on the computer system while performing interactions; thus, how to coordinate interaction and real-time sound synthesis became a significant issue to consider in software design. The following sections discuss this issue in more details.

2.2.3.2 Timing behaviour in sound synthesis libraries and frameworks

Some real-time sound synthesis libraries run a separate thread for interactive compositional algorithms so that sound synthesis can be performed in a thread with a higher priority than other threads. JSyn for Java (65) and Csound API for Python (176) belong to this kind.

However, the sample-rate accurate synchronization between two separate threads can be very difficult under the preemptive multitasking environment, since many general-purpose programming languages (in the above case, Java and Python) and the operating systems abstract the concept of time away, as discussed by Lee in (178). The threads for user tasks and the high-priority thread for sound synthesis can hardly synchronize and coordinate their behaviours with sample-rate accuracy under this kind of sound synthesis framework/library design.

2.2 The insufficient support for precise timing behaviour and other features with respect to time

On the other hand, some class libraries provide a means to control the progress of logical time with sample-rate accuracy, explicitly within the audio computation thread. Synthesis Toolkit (82) (STK) would be a good example to illustrate such library design. In Synthesis Toolkit, the method, ‘tick()’ can be used to compute a single sample, and logical time can be considered to be advanced for one sample every time the ‘tick()’ method is called. Figure 2.5 describes an example of the real-time sound output of a single sine wave in STK.

Such software design is beneficial to achieve sample-rate accuracy in timing behaviour, since the ‘tick()’ method call controls the advance of logical time in sound synthesis. However, while precise timing control can be achieved, this programming model makes it difficult to integrate compositional algorithms. In Figure 2.5, the compositional algorithms must be executed before line 43 where ‘tick()’ is called to compute the next output sample. A program can involve a certain degree of complexity in some cases, especially when a task must be interleaved and fragmented by ‘sleep’.

To illustrate, a case involving two tasks follows. Task A first performs *action A* followed by sleep for 0.25 seconds and then performs *action B* followed by another sleep for 1 second before performing *action C*. Task B first performs *action E* followed by sleep for 0.75 seconds and then performs *action F*. As these two tasks can be expressed with one thread of execution for each, the pseudo code example can be written as shown in Figure 2.6.

However, when these two tasks collaborate, they must be described in one line of execution, and the tasks must be merged into one as in Figure 2.7. Clearly, this is cumbersome. To avoid such complexity in a program, the task scheduler is often integrated to invoke scheduled tasks (if any) before computing the next sample. The Figure 2.8 example shows a pseudo code with such a task scheduler. When *taskA()* and *taskB()* are called, it registers the actions to the scheduler, instead of performing the actions immediately. In the main loop, the scheduler function is first called to process the scheduled tasks and then computes the output samples.

Generally speaking, it is not ideal if the code must be separated into fragments every time sleep in logical time is required, as this can complicate a program even when the task to be performed is conceptually simple. Consider a simple program with

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

```
01:  // rtsine.cpp STK tutorial program
02:
03:  #include "SineWave.h"
04:  #include "RtWvOut.h"
05:  #include <cstdlib>
06:
07:  using namespace stk;
08:
09:  int main()
10:  {
11:      // Set the global sample rate before creating class instances.
12:      Stk::setSampleRate( 44100.0 );
13:      Stk::showWarnings( true );
14:
15:      int nFrames = 10000;
16:      SineWave sine;
17:      RtWvOut *dac = 0;
18:
19:      try {
20:          // Define and open the default realtime output device for one-channel playback
21:          dac = new RtWvOut( 1 );
22:      }
23:      catch( StkError & ) {
24:          exit ( 1 );
25:      }
26:
27:      sine.setFrequency( 441.0 );
28:
29:      // Option 1: Use StkFrames
30:      /*
31:      StkFrames frames ( nFrames, 1 );
32:      try {
33:          dac->tick( sine.tick( frames ) );
34:      }
35:      catch ( StkError &) {
36:          goto cleanup;
37:      }
38:      */
39:
```

Figure 2.5: An example of real-time sound synthesis in STK (rtsine.cpp from STK tutorial program).

2.2 The insufficient support for precise timing behaviour and other features with respect to time

```
40:  // Option 2: Single-sample computations
41:  for ( int i=0; i<nFrames; i++ ) {
42:      try {
43:          dac->tick( sin.tick() );
44:      }
45:      catch ( StkError & ) {
46:          goto cleanup;
47:      }
48:  }
49:
50: cleanup:
51:  delete dac;
52:
53:  return 0;
54: }
```

Figure 2.5: An example of real-time sound synthesis in STK (rtsine.cpp from STK tutorial program) (continued).

```
01: void taskA()
02: {
03:     actionA();
04:     sleep(0.25);
05:
06:     actionB();
07:     sleep(1);
08:
09:     actionC();
10:
11:     return;
12: }
13:
14: void taskB()
15: {
16:     actionE();
17:     sleep(0.75);
18:
19:     actionF();
20:
21:     return;
22: }
```

Figure 2.6: Two different tasks to be performed concurrently.

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

```
01: void advance_logical_time(float duration)
02: {
03:     int samples = (int)(duration * SAMPLE_RATE);
04:     for (int i = 0; i < samples; i++){
05:         //tick
06:         tick();
07:     }
08: }
09:
10: void taskC()
11: {
12:     actionA();
13:     actionE();
14:     //call tick() to advance logical time for 0.25 second
15:     tick(0.25);
16:
17:     actionB();
18:     //call tick() to advance logical time for 0.5 second
19:     tick(0.5);
20:
21:     actionF();
22:     //call tick() to advance logical time for 0.5 second
23:     tick(0.5);
24:
25:     actionC();
26:     return;
27: }
```

Figure 2.7: An example that combines two tasks (task A and task B).

2.2 The insufficient support for precise timing behaviour and other features with respect to time

```
01: void taskA()
02: {
03:     schedule(actionA, 0.0);
04:     schedule(actionB, 0.25);
05:     schedule(actionC, 1.25);
06: }
07:
08: void taskB()
09: {
10:     schedule(actionE, 0.0);
11:     schedule(actionF, 0.75);
12: }
13:
14: void scheduler_main_loop()
15: {
16:     while(true){
17:         //process the scheduled tasks
18:         process_scheduled_task();
19:
20:         //tick the logical time, perform DSP
21:         tick();
22:     }
23:     return;
24: }
```

Figure 2.8: An example with a scheduler function.

```
01: void taskD(){
02:     while(true){
03:         actionA();
04:         sleep(0.25);
05:
06:         actionB();
07:         sleep(0.25);
08:
09:         actionC();
10:         sleep(0.5);
11:     }
12: }
```

Figure 2.9: An example of the infite loop with the 1.0 second sleep inside.

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

```
01: void taskD()
02: {
03:     schedule(actionA, 0.0);
04:     schedule(actionB, 0.25);
05:     schedule(actionC, 0.5);
06:     //rescheduling this function 1.0 second after.
07:     schedule(taskD, 1.0);
08: }
```

Figure 2.10: A temporal recursion example.

an infinite loop as in Figure 2.9. Such a code may require modification to involve a programming pattern called *temporal recursion*¹ as in Figure 2.10.

While these two examples above seem similar, there is a substantial difference in their behaviours between the infinite loop example (Figure 2.9) and the temporal recursion example (Figure 2.10) when considering the execution in a real-time, interactive programming environment such as live coding. For instance, while the infinite loop example may be processed in its own thread, the termination of the thread can simply stop the actions inside the loop at any time. It is possible to stop the thread right after *actionB()* is executed as there is an interval of 0.25 seconds before *actionC()*. When the thread is terminated there, *actionC()* will not be executed.

Yet, in the example of the temporal recursion, as the tasks are scheduled at once, including the rescheduling of *taskD*, the function *taskD* is executed without any sleep unlike the infinite loop example. Then, to terminate the task right before the scheduling of *actionC* by user interaction can be difficult. While the user has to stop the execution of *taskD* right after *actionB* is scheduled on line 04, there is no interval; thus, Figure 2.9

¹While earlier papers that describe similar programming patterns can be found such as MOXI, a language for computer music performance by Collinge (76) and CMU MIDI toolkit by Dannenberg (88), both didn't use the term, *temporal recursion*, and this term likely may be first invented by Sorensen and his colleagues to explain the programming pattern that they often utilize in Impromptu programming language (274)(276). However, it may be arguable if the term *temporal recursion* itself is appropriate for the programming pattern; the programming model of *temporal recursion* as Sorensen describes it differs from a normal recursive call in that it is a function that reschedules another call to itself. It does not call the function directly as seen in a normal recursive call. Sorensen's temporal recursion may be more similar to *continuation-passing style* (17), in that the function passes where the computation should continue as an argument for another function. While recursion can be easily converted into continuation-passing style (CPS), since *temporal recursion* in Impromptu as Sorensen describes, just schedules the function itself to Impromptu's scheduler so that it can be invoked some time in the future. Yet, we do not discuss this issue of the definition of the term *temporal recursion* further, as it is outside of the scope of the interests in this thesis.

2.2 The insufficient support for precise timing behaviour and other features with respect to time

and Figure 2.10 are not completely equivalent and timing behaviours at runtime significantly differ between these two examples, when considering user interaction. Extra care is needed in cases where this difference matters.

2.2.3.3 The use of coroutines in a sound synthesis framework

Coroutines can be beneficial for tersely describing a program that must concurrently process multiple tasks, without involving code fragmentation as seen in the examples in the previous section. The concept of coroutines is fairly mature in the history of programming languages and the term ‘coroutine’ itself was originated in the paper by Conway (80) written in 1963. Lerusalimschy briefly describes the concept as shown below, and Figure 2.11 describes a simple example of coroutine in the Lua programming language (150, p.73).

“A coroutine is similar to a thread (in the sense of multi-threading): a line of execution, with its own stack, its own local variables, and its own instruction pointer”, yet “the main difference between threads and coroutines is that, conceptually (or literally, in a multiprocessor machine), a program with threads runs several threads concurrently. Coroutines, on the other hand, are collaborative: A program with coroutines is, at any given time, running only one of its coroutines and this running coroutine only suspends its execution when it explicitly requests to be suspended.” (150).

Some software sound synthesis frameworks make the best use of the collaborative nature of coroutines for achieving precise timing behaviour with sample-rate accuracy in real-time sound synthesis. LuaAV by Wakefield and his colleagues (306)(308)(307) is a good example of this kind. The code in Figure 2.12 illustrates a simple example in LuaAV (307). In this example, the function call *wait(1)* actually suspends the execution and asks the underlying scheduler to resume this coroutine after 1 second. The underlying audio synthesis engine within the same framework can compute output samples until the timing (in logical time) when any coroutine is scheduled. When the logical time reaches the timing, the framework resumes the scheduled coroutine(s). Thus, the collaborative nature of coroutines is beneficial to achieve sample-rate accuracy in timing behaviour while avoiding the code fragmentations or the utilization of a particular programming pattern such as temporal recursion.

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

```
01: co = coroutine.create(function ()
02:     for i=1,10 do
03:         print("co", i)
04:         coroutine.yield()
05:     end
06: end)

coroutine.resume(co) --> co 1
coroutine.resume(co) --> co 2
coroutine.resume(co) --> co 3
...
coroutine.resume(co) --> co 10
coroutine.resume(co) -- prints nothing
```

Figure 2.11: A coroutine example in LuaA (150, p.73).

```
01: -- define a function to print a message
02: -- repeatedly, every 1 second
03: function printer(message)
04:     while true do
05:         print(message)
06:         wait(1)-- wait 1 second
07:     end
08: end
09: -- start ticking:
10: go(printer, "tick")
11: -- start tocking after 0.5 seconds
12: go(0.5, printer, "tock")
```

Figure 2.12: A LuaAV example (307).

2.2 The insufficient support for precise timing behaviour and other features with respect to time

01: <code>do</code>	Wait for a hit on a READY button within a time limit of 10 SECOND; in case of timeout, emit an ALARM; while waiting, any hit on the STOP button should ring a BELL.
02: <code>do</code>	
03: <code>every STOP do emit RING_BELL end</code>	
04: <code>upto READY</code>	
05: <code>watching 10 SECOND</code>	
06: <code>timeout emit ALARM end</code>	

Figure 2.13: An Esterel example and its specification (37).

2.2.4 Strongly-timed programming

2.2.4.1 Synchronous programming

While the use of coroutines in a sound synthesis framework can be used to achieve timing behaviour with sample-rate accuracy as seen in LuaAV, Wang considered such a problem in the context of synchronous programming in the design of his ChuckK audio programming language (312). Synchronous programming languages (e.g. Esterel (37)(38), Argos (198), Lustre (71) and SIGNAL (177)) are different with respect to time from real-time programming languages; while real-time programming languages (e.g., real-time Java (50) and Ada95 (67)) deal with the timing constraints given by the passage of physical time, synchronous programming languages replace the notion of the passage of physical time with the concept of *logical-instant*.

Synchronous programming languages are based on the ideal synchronous hypothesis (37), which states “ideal systems produce their outputs synchronously with their inputs” and “all computation and communications are assumed to take zero time (that is, all temporal scopes are executed instantaneously)” (66, p.360). Clearly, this assumption of ‘an infinitely fast machine’ is impossible to realize. Instead, “during implementation, the ideal synchronous hypothesis is interpreted to imply the system must execute fast enough for the effects of the synchronous hypothesis to hold” (66, p.360).

Many synchronous programming languages are designed to develop reactive systems. Reactive systems are “computer systems that continuously react to their environment at a speed determined by this environment”, whereas transformational systems are computer systems “whose inputs are available at the beginning of the execution and which deliver their outputs when terminating” and interactive systems “continuously interact with their environment, but at their own rate” (135). Figure 2.13 describes an example in Esterel and its specifications. As shown, an Esterel program is described as reactions to the external logical events.

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

2.2.4.2 ChuckK, a strongly-timed programming language

In the design of ChuckK, Wang proposed the concept of strongly-timed programming as a variation of synchronous programming, which integrates the explicit advance of logical synchronous time into an imperative programming language. A simple ChuckK program in Figure 2.14 illustrates the concept of strongly-timed programming.

On line 02, a simple synthesis patch is created by assigning a sine wave oscillator to the variable *foo* and connecting its output to *dac* (digital-to-analog converter) for sound output. In the infinite loop between line 05 and line 11, a random floating point value between 30 and 1000 is generated and set to the *freq* field of the *foo*, to change the frequency of the sine wave oscillator. As the strongly-timed programming concept is likewise founded on the ideal synchronous hypothesis, no advance in logical synchronous time is made so far (until line 9) , since (*all computation and communications are assumed to take zero time* (66, p.360)).

The core concept of strongly-timed programming is the best illustrated on line 10. The expression '*100::ms => now*' suspends the execution of the current thread and then advances ChuckK's internal logical time for 100 ms¹ before the thread is resumed. While the thread is sleeping, the underlying sound synthesis framework computes the output audio samples from the unit-generator graph until the time when the next thread must be awakened. Hence, the frequency of the sine wave oscillator patch in the program is changed every 100 ms precisely in logical time with sample-rate accuracy. Additionally, Figure 2.15 shows a simple ChuckK program that performs the same task as the LuaAV example in Figure 2.12².

2.2.4.3 Discrete event simulation in FORMULA, coroutines in LuaAV and strongly-timed programming in ChuckK

Even though there exists some similarity between the concept of discrete event simulation and the concept of synchronous programming, and there even exists a compiler that translates a synchronous program in Esterel into the equivalent C program that performs discrete event simulation (107), these two are different concepts; the former

¹While 100 ms is equal to 4410 samples in the system with the sample-rate of 44100Hz, in the case such a value as 1 ms (=44.1 samples) is used, it is truncated to 44 samples in ChuckK. This may accumulate an error in timing.

²However, as ChuckK doesn't have the features to schedule the start of a thread in the future, one must explicitly advance the logical time as seen on line 10.

2.2 The insufficient support for precise timing behaviour and other features with respect to time

```
01: // synthesis patch
02: SinOsc foo => dac;
03:
04: // infinite time loop
05: while(true)
06: {
07:     // randomly choose a frequency
08:     Std.rand2f(30, 1000) => foo.freq;
09:     // advance time
10:     100::ms => now;
11: }
```

Figure 2.14: A ChuckK program to generate a sine wave, changing its frequency of oscillation every 100 milliseconds. (312, p.43).

```
01: fun void printer(string message)
02: {
03:     while(true){
04:         <<< message >>>;
05:         1::second +=> now;
06:     }
07: }
08:
09: spork ~ printer("tick");
10: 0.5::second +=> now;
11: spork ~ printer("tock");
12: //so to keep the childe threads alive.
13: while(true) 10::second +=> now;
```

Figure 2.15: A simple ChuckK program, which performs the equivalent task as the Figure 2.12 example in LuaAV.

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

(discrete event simulation) is a model of a system for a discrete sequence of events in time and the latter (synchronous programming) is a programming language concept. As Wang emphasizes in (312), ChuckK's strongly-timed programming concept is in the context of synchronous programming. It is designed upon *ideal synchronous hypothesis*, in which all computation and communications are considered to consume zero time. Such a concept of ideal synchronous hypothesis does not exist in discrete event simulation.

The difference in the problem domains between FORMULA and ChuckK/LuaAV should be also emphasized. FORMULA adopts discrete event simulation mainly for communication with the external synthesizer hardware in precise timing, while the problem domain of ChuckK and LuaAV focus on the accurate computation of output samples when sample-rate accuracy in timing precision is required; the motivations and the concepts significantly differ between FORMULA and ChuckK/LuaAV, even though both involve internal logical time in their design.

One may also consider that the strongly-timed programming concept is similar to the use of coroutines in LuaAV. However, coroutine is a different concept in language design and has nothing to do with the ideal synchronous hypothesis. While coroutines can be used to achieve a precise timing behaviour, this is an issue in software design rather than a programming concept.

Strongly-timed programming is a concept in programming language design and differs from the concept of discrete event simulation, which is a model of a system, and differs from coroutines, which is another concept in programming language design. Strongly-timed programming is a variation of synchronous programming and significantly relies on the ideal synchronous hypothesis, which is not taken into account in discrete event simulation and coroutines.

2.2.4.4 Visual computer music programming languages

While this thesis focuses on the design of a textual computer music language, we briefly discuss the issue of precise timing behaviour in visual computer music languages in this section. Generally speaking, many visual computer music languages exhibit a timing behaviour similar to the synchronous programming concept. In visual languages, the

2.2 The insufficient support for precise timing behaviour and other features with respect to time

tasks for compositional algorithms often run between each DSP computation cycle. The documentation of PureData (1) clearly explains such behaviour.

“Audio and message processing are interleaved in Pd. Audio processing is scheduled every 64 samples at Pd’s sample rate; at 44100 Hz. This gives a period of 1.45 milliseconds. You may turn DSP computation on and off by sending the ‘pd’ object the messages ‘dsp 1’ and ‘dsp 0’.”

“In the intervals between, delays might time out or external conditions might arise (incoming MIDI, mouse clicks, or whatnot). These may cause a cascade of depth-first message passing; each such message cascade is completely run out before the next message or DSP tick is computed. Messages are never passed to objects during a DSP tick; the ticks are atomic and parameter changes sent to different objects in any given message cascade take effect simultaneously.”

As above, since the system does not allow the computation of the next output samples until it finishes processing all the messages, any message processing in PureData conceptually consumes zero time in logical time. While the control-rate still exists in PureData¹, the effort has been made to realize sample-rate accurate event handling as seen in (194). Furthermore, setting the audio vector size to one sample would suffice to achieve sample-rate accuracy in such languages, even though this may lead to considerable inefficiency in real-time DSP performance.

Recent versions of Max have several different strategies for handling events (2)(168). When *Scheduler in Audio Interrupt (SIAl)* of Max/MSP is turned on, the high priority events can be processed inside the audio thread. Max uses this strategy normally in non real-time mode. Yet, MAX normally processes the events in other threads so to avoid the suspension of DSP by timing-consuming tasks. This event-processing strategy outside the audio thread also classifies events to high-priority events and low-priority events. However, speaking generally, processing events in a separate thread (not in the

¹As it computes the audio output by the audio vector of 64 samples as described, PureData can interact with the events only at this rate.

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

audio thread), causes a lack of precise timing behaviour and synchronization between the tasks, as discussed in Section 2.2.3.2.

2.2.5 The problems in the existing computer music programming languages

As described so far, precise timing behaviour has been traditionally a topic of significant interest, especially after the emergence of live interactive computer music practices. Unlike in non real-time/non-interactive computer music environments, which only have to deal with deterministic events offline, the requirements for real-time interaction complicate this issue, since such systems must interact with incoming events and data while keeping a certain precision in timing behaviour. In the era when a live computer music system still consisted of a computer and external hardware, the problem of precise timing behaviour was seen in the context of the interaction with the external devices. However, after real-time DSP was realized on a stand-alone computer without the aid of external hardware, the focus of this issue shifted onto the achievement of sample-rate accuracy in the synchronization between compositional algorithms and real-time sound synthesis. The difference in the problem domains between FORMULA and ChuckK/LuaAV may illustrate such transitions well.

However, while the approaches taken in ChuckK and LuaAV successfully achieved precise timing behaviour with sample-rate accuracy, a problem still exists for truly interactive computer music applications. The solutions proposed by ChuckK and LuaAV still exhibit a significant problem in the presence of time-consuming tasks. As it is seen more clearly in ChuckK's strongly-timed programming, it is assumed that a task can be completed before the beginning of the next DSP cycle. When the advance of internal logical synchronous time is hindered by a time-consuming task, the underlying sound synthesis framework can easily fail to provide output sound samples to the sound devices; thus, temporary suspension in real-time DSP can occur.

Yet, a computer music program can often involve time-consuming tasks. For instance, loading a large sound file from the disk and analysing it for wavetable synthesis can be time-consuming and can easily suspend real-time DSP. In addition, audio computation can never be resumed if an error is made in writing an infinite loop that does

2.2 The insufficient support for precise timing behaviour and other features with respect to time

not contain the advance of logical-time within.

While the design of the scheduler in Max/MSP may provide a certain solution for this issue, it is still problematic as it causes another problem. As described in the previous section, processing the tasks in a different thread than the audio thread makes it almost impossible to realize the precise synchronization between compositional algorithms and sound synthesis. This is fundamentally the same issue in the sound synthesis framework design that processes the compositional algorithms in separate threads from the audio computation thread. The same problem can be also found in SuperCollider and Impromptu, as both perform compositional algorithms in a different thread (or a process). While the execution of time-consuming tasks does not suspend real-time DSP in these languages, the synchronization between sound synthesis and compositional algorithm is significantly damaged.

Moreover, not just synchronization between compositional algorithms and sound synthesis, but also synchronization between compositional algorithms is also an issue to consider if compositional algorithms are also performed by several different threads. Even in the situation where two different threads output events that should be performed at exactly at the same timing, these two events may not be processed right with the same timing¹.

There still remains a problem in precise timing behaviour, even in the latest computer music languages. While the design of FORMULA suggests the benefits of processing time-consuming tasks in the background, since it is for the research on hybrid computer music systems with external synthesizer, both motivation and requirements for timing behaviour largely differ from what we currently require; today's computer music systems must achieve sample-rate accurate timing behaviour and stand-alone real-time DSP at the same time.

The integration of background tasks in LuaAV would require significant redesign and modification, not just in the software framework, but also in the run-time system of the host programming language, especially because the Lua programming language, upon which LuaAV is built, intentionally excludes preemptive threading from the language

¹Such an issue is actually another benefit of the synchronous language design with coroutines as in LuaAV or lightweight concurrency as in ChuckK.

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

specifications and supports multi-tasking only by coroutines. ChuckK's strongly-timed programming concept is based on the ideal synchronous hypothesis that assumes an instruction consumes zero time. Hence, its language design does not consider the concept of background tasks, as any task can be conceptually finished without the advance of logical time, except when the task explicitly advances logical time.

Other desirable features with respect to time are not integrated well into those languages with sample-rate accurate timing behaviour. Even those features seen in the era of the hybrid computer music systems seem to disappear in recent computer music languages. For instance, FORMULA had *time-control structures* such as *maxtime(n)*¹ and *mintime(n)*² for timing constraints.

For another instance, Impromptu is an example of recent computer music languages that consider such features with respect to time in their language design. In (276), Sorensen and Gardner discuss desirable features with respect to time in the design of Impromptu, referring to the paper 'Motivating time as a first class entity' by Lee et al.(179). The list of the features Sorensen and Garder discussed includes: (1) the ability to express timing constraints, (2) timed communication, (3) enforcement of timing constraints, (4) tolerance to violations and constraints, (5) maintaining consistency in distributed real-time systems, and (6) static timing verification; yet, while Impromptu succeeded in achieving the design goals to some degree, Impromptu has not realized the sample-rate accurate synchronization between compositional algorithms and sound synthesis, as Impromptu processes compositional algorithms and sound synthesis in different threads.

As above, while ChuckK and LuaAV have achieved precise timing behaviour with sample-rate accuracy, they still exhibit the serious problem of suspension of real-time DSP in the presence of time-consuming tasks. Even though the designers of Impromptu made efforts to integrate other desirable features with respect to time into the computer music language, such features are not realized with sample-rate accuracy. It is still an

¹ "The *maxtime* (*n*) structure specifies that the statement is to consume at most *n* units of virtual time. When the statement is entered, the upper limit (time-position + *n*) is recorded" (16).

² "The *mintime* (*n*) structure specifies that the statement is to be extended by an 'invisible' time advance, if necessary, so that it consumes at least *n* units of virtual time. When the statement is entered, the lower limit (time-position + *n*) is recorded. If, when the end of the statement is reached, the time position is less than this limit, a time-advance() is done to reach the limit" (16).

2.3 The difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion

issue of significant interest to investigate how precise timing behaviour can be accomplished while avoiding such temporary suspensions in real-time DSP. It should also be considered as to how other desirable features with respect to time can be integrated with precise timing behaviour; the existing computer music programming languages only partially achieve such design goals.

2.3 The difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion

The popularization of microsound synthesis techniques in creative practices reveals a problem in existing computer music programming languages. Generally speaking, a microsound synthesis techniques conceptualize the entire sound as the composition of many short sound particles (or microsounds), which may overlap onto others (overlap-add). This concept of microsound synthesis techniques significantly differs from what the traditional unit-generator concept assumes, as it is modelled after electronic sound synthesis by standard electronic equipments (e.g., oscillators and filters).

Such a gap in concepts leads to difficulty in programming microsound synthesis techniques in unit-generator languages¹. While it is still possible to encapsulate each microsound synthesis technique within a dedicated unit-generator, such an approach can cause a significant obstacle when exploring microsound synthesis techniques, as it hinders further exploration beyond the ready-made functions and interfaces provided by unit-generators; such a situation is not ideal support of creative practices.

A user can modify the code of such unit-generators or even write a new unit-generator from scratch, to go beyond such a restriction. However, in this case, a certain expertise in programming is required, together with the understanding of the language's sound synthesis framework, which hardly can be expected of end-users. Moreover, even for a user with the best expertise in programming, such a situation is not ideal for rapid-prototyping and creative exploration, as it can require a considerable amount of time to develop a new unit-generator.

¹Roads categorizes those computer music programming languages built upon the unit-generator concept as *unit-generator languages* in (241, p.787).

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

Thus, it is desirable to implement microsound synthesis techniques within a computer music language, since the algorithms can be directly modifiable in the computer music programming language, and the creative exploration can be made easier. However, this strategy also exhibits another type of problems in computer music programming. As the traditional abstraction of the unit-generator concept does not provide direct counterpart objects and manipulations to the ones that appear in the conceptualization of microsound synthesis techniques, normally each microsound is modelled as a note-level object in a unit-generator language, and overlap-add of microsounds is performed by scheduling of such note-level objects.

Moreover, some additional programming patterns or the use of library functions may need to be involved due to the lack of precise timing behaviour in some computer music programming languages; as a result, even when implementing a microsound synthesis technique that is conceptually very simple, the actual code can be more complicated.

The following sections in this chapter describe such difficulty in microsound synthesis programming in the existing computer music programming languages. The problem is discussed mainly in the context of the software anti-pattern (13) called *abstraction inversion* (28).

2.3.1 The unit-generator concept and microsound synthesis techniques

2.3.1.1 The unit-generator concept

The unit-generator concept first appeared in MUSIC-III in 1960. A unit-generator is “a software module that emits audio or control signals (envelopes) or modifies these signals” (241, p.787), “which can be interconnected to form synthesis instruments or patches that generate sound signals” (241, p.89). Figure 2.16 shows the pictorial representation of a unit-generator graph. The origin of the unit-generator concept is considered to be rooted in analogue sound synthesizers. Mathews and his colleagues, the inventors of the unit-generator concept, clearly state that unit-generators perform “conceptually similar functions to standard electronic equipment used for electronic sound synthesis” in the book they wrote in 1969 (204, p.36).

Since its conception, the unit-generator concept has been serving as one of the most important core abstractions in computer music language design. While there exists a certain degree of difference in how the unit-generator concept is integrated and

2.3 The difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion

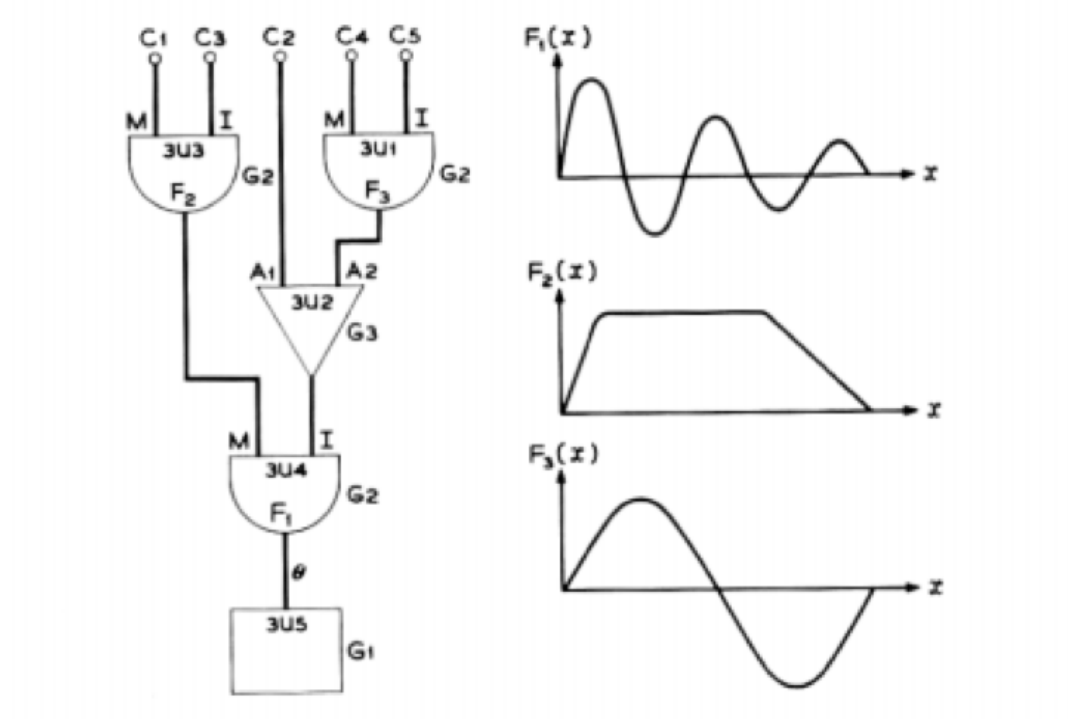


Figure 2.16: An Instrument with attack, decay, and vibrato - as seen in *An Acoustic Compiler for Music and Psychological Stimuli* by Mathews (201).

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

implemented¹, the concept is still seen even in the latest computer music languages. Languages such as Csound (52), Max/MSP (327), PureData (227)(232), SuperCollider (211)(320), and ChuckK (312)(314) all depend on the unit-generator concept for digital sound synthesis.

2.3.1.2 Microsound synthesis techniques

Around the mid-1940s, Dennis Gabor, a British physicist, first proposed the concept of sound that originated microsound synthesis techniques. While the theory proposed by Gabor is more strongly associated with time-frequency analysis rather than digital sound synthesis, the concept that Gabor presented is considered to have had the most significant influence in the emergence of microsound synthesis techniques as we have today. In (242, p.57), Roads briefly summarizes the concept as follows.

“In Gabor’s conception, any sound can be decomposed into a family of functions obtained by time and frequency shift of a single Gaussian particle. Another way of saying this is that any sound can be decomposed into an appropriate combination of thousands of elementary grains.”

Generally speaking, microsound synthesis techniques involve short-duration sound particles that may overlap-add each other to constitute the entire sound output; normally the duration of such sound particles extends between “the threshold of timbre perception (several hundred microseconds) up to the duration of short sound objects (~ 100 ms)” and they span “the boundary between the audio frequency range (approximately 20 Hz to 20 kHz) and the infrasonic frequency range (below 20 Hz)” (242, p.21).

Even though the implementations of microsound synthesis techniques as a computer program were realized long after Gabor’s original publication², various new microsound synthesis techniques have been developed since then. Such synthesis techniques as gran-

¹For instance, some computer music languages such as Max/MSP and ChuckK extends the unit-generator concept with methods as in object-oriented programming languages while other languages such as Csound and SuperCollider simply implement the unit-generator without such extensions.

²One of the earliest known study that involves digital granular synthesis was composed by Roads in 1974, using Music V language (242, p.302).

2.3 The difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion

ular synthesis (240), FOF synthesis (246), FOG synthesis (74), and waveset synthesis (322) all belong to the category of microsound synthesis.

2.3.2 Abstraction inversion in microsound synthesis programming

2.3.2.1 Abstraction inversion

In this subsection, we discuss the difficulty in microsound synthesis programming in the context of *abstraction inversion*, one of the major software anti-patterns. The term *software anti-patterns* is defined as “a collection of tasks/techniques/actions that have proven ineffective for developing software” (13) and abstraction inversion “occurs when a programmer is forced to use a combination of higher-level abstractions to express a lower-level abstraction” (28).

One of the most well-known examples of abstraction inversion is the mutex¹ example in the Ada programming language as Baker describes in (28); since the early versions of Ada did not provide mutex,² mutual exclusion must be implemented by using *rendezvous*, which is “a kind of synchronous communication mechanism between the client task and the server task” (193). In Ada, a server task accepts only one particular request at a time and the other client tasks will be blocked until their requests are accepted. By using this mechanism, mutex can be implemented by writing a server task with the infinite loop that first accepts a ‘Lock’ request and then accepts an ‘Unlock’ request. Figure 2.17 describes the example by Baker (28).

However, the ‘rendezvous’ mechanism involves mutual exclusion internally to guarantee the server task process only one request at a time, blocking the other requests. Hence, this Ada’s mutex example is considered a typical abstraction inversion in that the higher-level abstraction (*rendezvous*) is used to describe a lower-level abstraction (mutual exclusion).

Bitwise operation in the Lua programming language (151) is another widely-known example of abstraction inversion. Since the earlier versions of Lua³ had no bitwise

¹‘Mutex’ stands for mutual exclusion that ensures that only a single task (or thread) can execute the critical section of a program.

²However, the language specification of Ada was extended to include mutex from Ada95 (285).

³prior to the Lua 5.2.

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

```
01: task body Mutex is
02: begin
03:   loop
04:     accept Lock;
05:     accept Unlock;
06:   end loop;
07: end Mutex;
```

Figure 2.17: A mutex example with abstraction inversion in Ada (28).

```
01: function bit_and(x, y)
02:   local digit = 1
03:   local ret = 0
04:   local limit = x > y and x or y
05:   while digit <= limit do
06:     if (x % (digit * 2)) >= digit and (y % (digit * 2)) >= digit then
07:       ret = ret + digit
08:     end
09:     digit = digit * 2
10:   end
11:   return ret
12: end
```

Figure 2.18: A bitwise operation (bitwise-and) example in Lua (reproduced from Figure 1.2).

operators (e.g., ‘|’ (bitwise or) and ‘&’ (bitwise and) as seen in the C programming language), one must implement bitwise operators in Lua as shown in Figure 2.18, or must use the external module written in the C programming language must be used. As even simple bitwise operations must be written as the combination of loops and conditional branches, this Lua example is considered an abstraction inversion problem; thus, the inappropriate abstraction in programming language design can easily lead to unnecessary complexity.

2.3.2.2 The microsound synthesis examples in SuperCollider and ChuckK

The gap between the unit-generator concept, which originates the sound synthesis by analogue electronic equipment, and microsound synthesis techniques, which considers short sound particles to be building blocks of the entire sound, can be clearly seen in the actual source code for microsound synthesis techniques. As described in the

2.3 The difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion

following examples, implementation can involve a certain degree of complexity even for a conceptually simple microsound synthesis technique.

Figure 2.19 shows a pictorial representation of waveset harmonic distortion. A waveset is defined as “the distance from a zero-crossing to a 3rd zero-crossing” (323, p.50), as seen in the segments separated by the dotted lines on three waveforms (a) (b) (c) in Figure 2.19. Generally speaking, Waveset synthesis techniques constitute its output by transforming and scheduling wavesets in various ways. Table 2.2 describes the list waveset synthesis techniques that are available in the Composer’s Desktop Project software by Trever Wishart (322)(323) (taken from (242, p.207)).

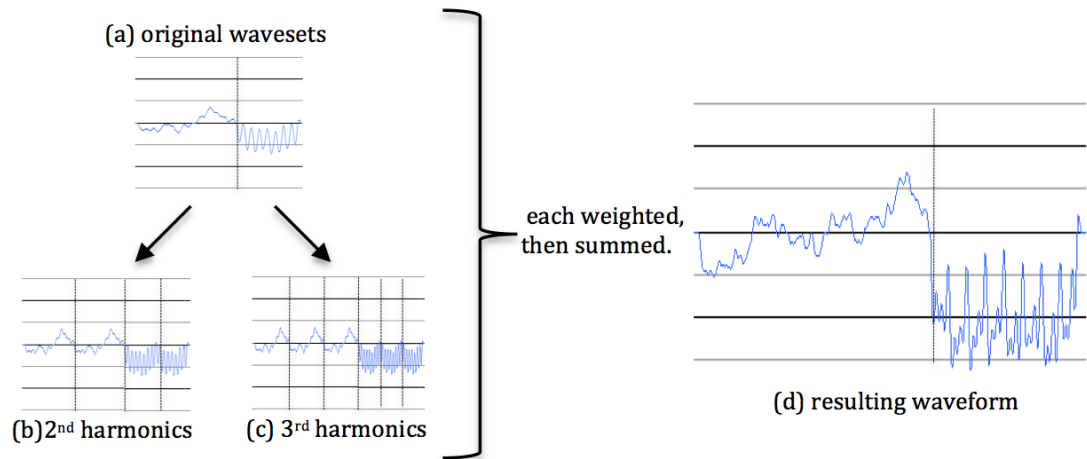


Figure 2.19: A pictorial representation of waveset harmonic distortion.

As seen in the table, Waveset harmonic distortion is a simple sound synthesis technique, which “superimposes N harmonics on the waveset fundamental with a scaling M relative to the previous harmonic” (242, p.207). As already described previously, when implementing microsound synthesis algorithms in a unit-generator language, each microsound is normally modelled as a note-level object. Figure 2.20 is one of the typical strategies to implement waveset harmonic distortion in SuperCollider computer music language (320). While we do not explain the detail of the code in this section, this example code clearly seems more complicated when compared to the simple concept of waveset harmonic distortion.

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

Waveset transposition	substitutes N copies of a waveset in the place of M wavesets, for example 2 in the space of 1, or 1 in the space of 4, for doubling and quartering of frequency, respectively
Waveset reversal	reverses individual wavesets while retaining their order; reversals can be of each individual waveset or collections of N wavesets at a time.
Waveset shaking	alternates between compressing (in time) and expanding (in time) successive wavesets
Waveset inversion	inverts the phase of all wavesets in a signal; in Wishart's diagram half-wavesets are inverted
Waveset omission	deletes every Nth waveset, leaving silence in its place; controlled by initial and final density from 0 to 100 %
Waveset shuffling	permutes collections of wavesets. A simple shuffle of successive wavesets starting with (a, b, c, d) becomes (d, c, a, b)
Waveset distortion	distorts a signal by squaring and cubing the signal; the example given in Wishart does not indicate how it is tied to waveset boundaries
Waveset substitution	replaces wavesets by a stipulated waveform of the same amplitude, frequency, and time span as the original waveset
Waveset harmonic distortion	superimposes N harmonics on the waveset fundamental with a scaling factor M relative to the previous harmonic
Waveset averaging	creates a signal containing N new wavesets that are the average duration and the average amplitude and time function of the N original wavesets; the overall duration of the signal is unchanged
Waveset enveloping	applies an envelope to 1 to N wavesets at a time
Waveset transfer	substitutes the waveform of the wavesets in signal A into the time frames of the wavesets in signal B
Waveset interleaving method 1	substitutes wavesets from signal A into alternate wavesets of signal B; applied either to individual wavesets or groups.
Waveset interleaving method 2	interleaves wavesets from signal A with wavesets of signal B, thus lengthening the output signal; applied either to individual wavesets or groups
Waveset time-stretching	repeats each waveset N times
Waveset time-shrinking	retains only the first of every N wavesets or retains only the loudest of every N wavesets
Waveset normalizing	normalizes every N wavesets above a stipulated amplitude threshold, thus a 10% threshold has a greater effect than a 90% threshold

Table 2.2: Waveset transformations in the Composer's Desktop Project software (242, p.207).

2.3 The difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion

```
01: Server.default = s = Server.internal;
02: s.boot;
03: w = Wavesets.from("sound.aif");
04: (
05:   b = w.buffer;
06:   SynthDef(\wvst0, {
07:     arg out = 0, buf = 0, start = 0, length = 441,
08:     playRate = 1, sustain = 1, amp = 1;
09:     var phasor = Phasor.ar(rate:playRate, start:0, end:length) + start;
10:     var env = EnvGen.ar(Env([amp, amp, 0], [sustain, 0]), doneAction:2);
11:     var snd = BufRd.ar(1, buf, phasor) * env;
12:     OffsetOut.ar(out, snd);
13:   }).add;
14: )
15: (
16:   var numOfWeeksets = w.lengths.size;
17:   var original = Pbind(
18:     \instrument, \wvst0,
19:     \startWs, Pseries(0, 1, numOfWeeksets),
20:     \numWs, 1,
21:     \playRate, 1,
22:     \bufnum, b.bufnum,
23:     \repeats, 1,
24:     \amp, 1,
25:     [\start, \length, \sustain], Pfunc( { |ev|
26:       var start, length, wsDur;
27:       #start, length, wsDur = w.frameFor(ev[\startWs], ev[\numWs]);
28:       [start, length, wsDur * ev[\repeats] / ev[\playRate].abs]
29:     }),
30:     \delta, Pkey(\sustain)
31:   );
```

Figure 2.20: A waveset harmonic distortion example in SuperCollider (reproduced from 1.1).

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

```
32: var octup = Pbind(  
33:   \instrument, \wvst0,  
34:   \startWs, Pseries(0, 1, numOfWavesets),  
35:   \numWs, 1,  
36:   \playRate, 2,  
37:   \bufnum, b.bufnum,  
38:   \repeats, 2,  
39:   \amp, 1,  
40:   [\start, \length, \sustain], Pfunc( {|ev|  
41:     var start, length, wsDur;  
42:     #start, length, wsDur = w.frameFor(ev[\startWs], ev[\numWs]);  
43:     [start, length, wsDur * ev[\repeats] / ev[\playRate].abs]  
44:   } ),  
45:   \delta, Pkey(\sustain)  
46: );  
47:  
48: Ppar([original, octup]).play(SystemClock);  
49: )
```

Figure 2.20: A waveset harmonic distortion example in SuperCollider (reproduced from 1.1) (continued).

Synchronous granular synthesis is another widely-used microsound synthesis technique, in which “sounds result from one or more streams of grains”. “Within each stream, one grain follows another, with a delay period between the grains. Synchronous means that the grains follow each other at regular intervals” (242, p.93). Figure 2.21 describes a pictorial representation of synchronous granular synthesis. As shown, even all the grains involved in the synthesis are completely identical to each other, simply altering the interval between the grains can result in various pitches and waveforms in synchronous granular synthesis.

While synchronous granular synthesis is conceptually simple, its implementation in a unit-generator language can also involve a certain degree of complexity, similar to that seen in the previous waveset synthesis example in SuperCollider. The code in Figure 2.22 is an example of synchronous granular synthesis in the Chuck programming language. As Chuck does not offer any note-level object in its abstraction, each microsound is modelled as a thread in this example¹.

¹In (312) and other publications, Wang uses the word ‘*shred*’ for a non-preemptive thread in Chuck. Yet, as this word is a Chuck-specific technical term and not used in any other languages, we simply use the word ‘thread’ (or ‘non-preemptive thread’) in this thesis.

2.3 The difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion

While the example in ChuckK in Figure 2.22 may seem terser than the wavetable synthesis example in SuperCollider in Figure 2.20, the code still exhibits a certain degree of complexity when considering the very simple concept of synchronous granular synthesis. As the current implementation of ChuckK does not provide any garbage collection mechanism, the code can be further complicated to minimize memory leak. The use of the *SndBuf* object within each thread is also problematic as it loads the sound file from the disk when it is played and disk-access occurs every time a grain is played. Figure 2.23 shows an example of the version with less memory-leak and disk-access. To minimize the allocation of new objects and disk-access as much as possible, the program launches only the required number of the threads that keep on looping during sound synthesis.

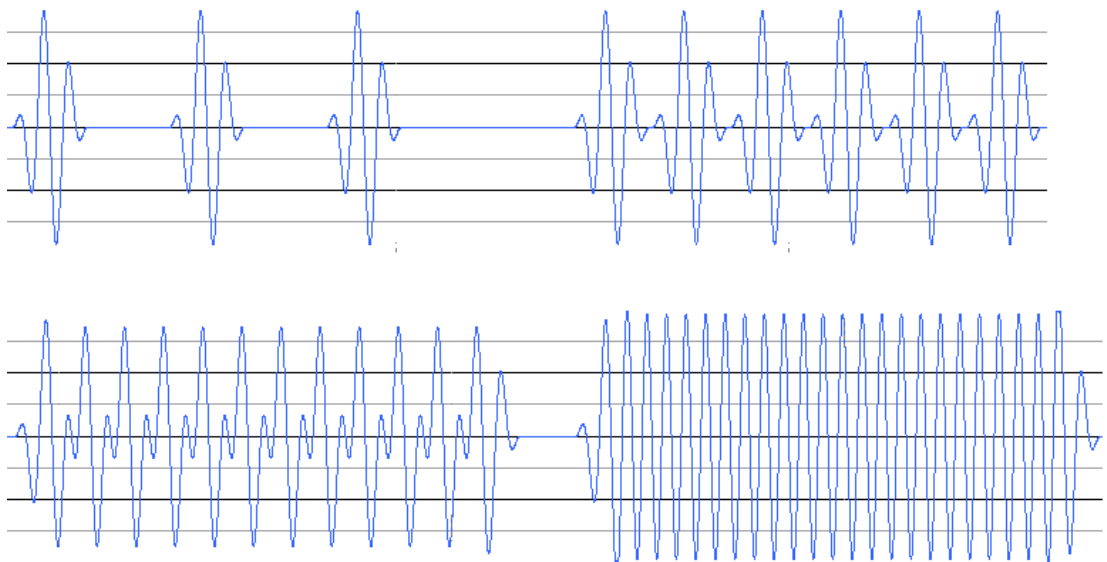


Figure 2.21: A pictorial representation of synchronous granular synthesis. - All the grains are identical; the change of the interval results in the various pitches and waveforms.

Furthermore, the conceptual gap between microsound synthesis techniques and the unit-generator concept can lead to a further problem in programming. Given library functions for microsound synthesis may be considered enough and users may not have to understand the detail of the whole implementation. However, while the use of the library functions may reduce the complexity at the surface level, the implementation details may need modified, which may be abstracted away by encapsulation, for further exploration of a certain microsound synthesis technique.

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

```
01: "sound.aif" => string filename;
02:
03: 22050 => int startPos;
04: 0.1 => float sustain;
05: 40 => int rep;
06: 0.05 => float interval;
07:
08: fun void grain()
09: {
10:
11:   SndBuf buf => Envelope env => dac.left;
12:   filename => buf.read;
13:
14:   startPos => buf.pos;
15:   sustain::second / 2 => env.duration;
16:
17:   env.keyOn();
18:   (sustain * 0.5)::second => now;
19:   env.keyOff();
20:   (sustain * 0.5)::second +=> now;
21:
22:   return;
23: }
24:
25: for(0 => int i; i < rep; i + 1 => i){
26:   spork ~ grain();
27:   interval::second +=> now;
28: }
29:
30: sustain::second +=> now;
```

Figure 2.22: A synchronous granular synthesis example in ChucK.

2.3 The difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion

```
01: "a11w1k01-44_1.aif" => string filename;
02:
03: 22050 => int startPos;
04: 0.1 => float sustain;
05: 40 => int rep;
06: 0.05 => float interval;
07: 0 => int cnt;
08:
09: (sustain / interval) $ int => int numOfThreads;
10: if (sustain / interval > numOfThreads){
11:     numOfThreads + 1 => numOfThreads;
12: }
14: fun void grain()
15: {
16:     SndBuf buf => Envelope env => dac.left;
17:     filename => buf.read;
18:
19:     while(cnt < rep){
20:         cnt + 1 => cnt;
21:
22:         now + numOfThreads * interval::second => time nextGrainStartTime;
23:
24:         startPos => buf.pos;
25:         sustain::second / 2 => env.duration;
26:
27:         env.keyOn();
28:         (sustain * 0.5)::second +=> now;
29:         env.keyOff();
30:         (sustain * 0.5)::second +=> now;
31:
32:         nextGrainStartTime =>now;
33:     }
34:     return;
35: }
36:
37: for(0 => int i; i < numOfThreads; i + 1 => i){
38:     spork ~ grain();
39:     interval::second +=> now;
40: }
41: (interval * (rep - numOfThreads) + sustain)::second +=> now;
```

Figure 2.23: Another synchronous granular synthesis example in ChuckK with less memory-leak.

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

For instance, *waveset inversion* may involve such modification. Waveset inversion is a waveset synthesis technique that inverts the half-wavecycles (322, p.42) as pictorially represented in Figure 2.24. As an inverted waveset cannot be generated simply by changing the playback speed as in the case of waveset harmonic distortion, it is necessary to modify a unit-generator graph to invert a waveform and to perform pre-analysis to obtain the parameters required for the inversion. The difficulty in microsound synthesis can not be solved by providing the library code that encapsulates the details, since the encapsulated details may need to be accessed for further exploration.

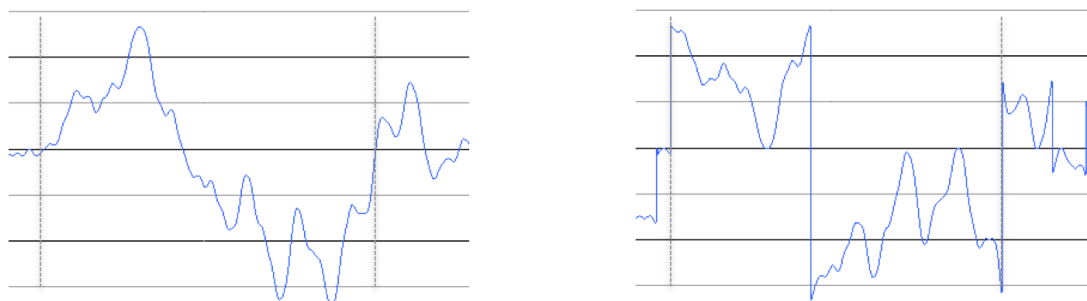


Figure 2.24: A pictorial representation of waveset inversion - the original waveset (left) and the inverted waveset (right).

2.3.2.3 The microsound synthesis examples in visual programming languages

While this thesis focuses on the design of the textual programming languages, we also additionally describe an example of granular synthesis in a visual programming language. Figure 2.25 (the whole patch) and Figure 2.26 (inside the *rgrain2~*— subpatch) show a granular synthesis example by Dudas. While the program is written in Max/MSP, the same programming pattern can be applied to PureData, too.

The programming pattern used in this patch is to cascade the subpatches (*rgrain2~*), each of which plays a single grain. When a subpatch that is already active receives the request to play a single grain, it forwards the request to the next cascaded subpatch. The overlap-add of the grains is performed in this manner¹.

¹However, such delegation mechanism is not necessary when using a *poly* object provided in the recent version of Max/MSP, which makes it possible to use a subpatch as if it were a polyphonic synthesizer.

2.3 The difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion

Basically speaking, The same problem of abstraction inversion as seen in the textual computer music language examples can be observed, since a subpatch is considered a higher-level abstraction than a single microsound. Encapsulation by the subpatch is not beneficial when experimenting with the different kind of microsounds, as the algorithm encapsulated within the subpatch may need modified (as in the case of waveset inversion).

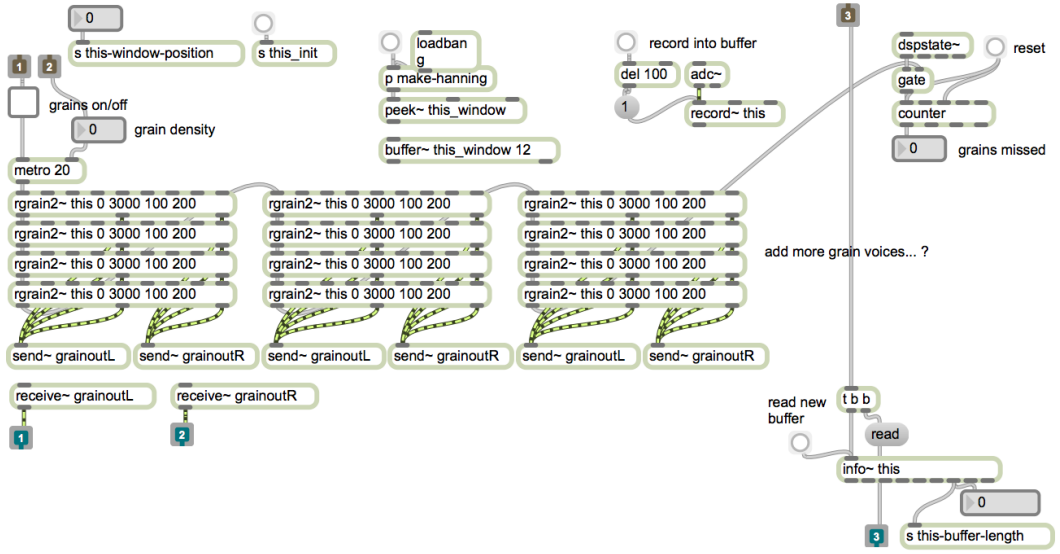


Figure 2.25: A granular synthesis patch by Richard Dudas (the whole patch) - the courtesy of the composer.

2.3.2.4 The lack of objects and manipulations for microsound synthesis in the sound synthesis software frameworks

Table 2.3 describes the hierarchical classification of *the time-scales of music* by Roads (242, p.3). As shown, microsound time-scale is conceptually placed in the lower level in this hierarchy. While the entire sound produced by a microsound synthesis technique consists of many microsounds, which may overlap-add each other, in practice, each microsound is modelled as a note-level object as seen in the previous examples in unit-generator languages. Hence, the complexity exhibited in the code examples can be considered as abstraction inversion; while the task to be performed is simply scheduling of microsounds, users have to implement it as the scheduling of many note-level objects,

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

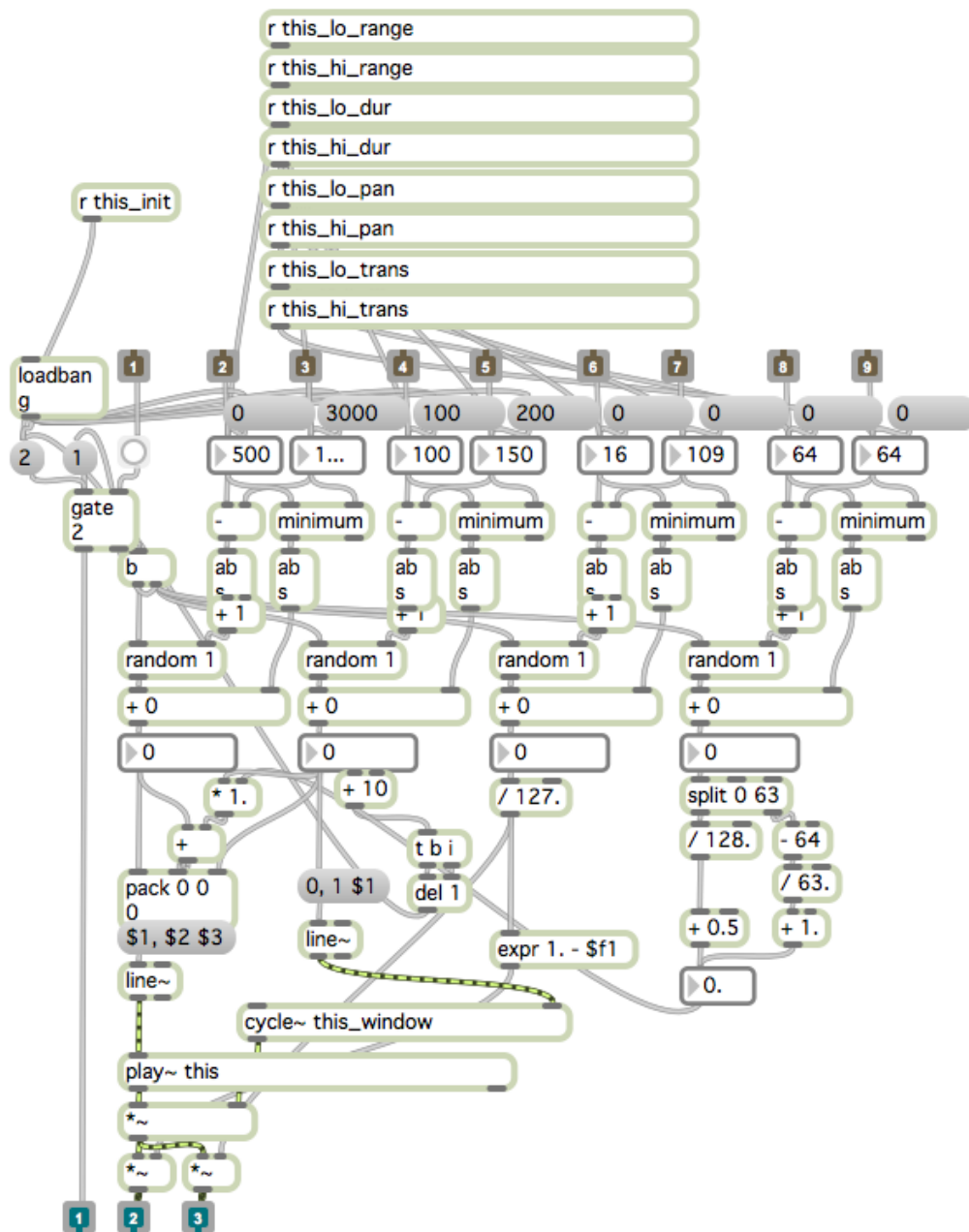


Figure 2.26: A granular synthesis patch by Richard Dudas (the inside *rgrain2~* subpatch) - the courtesy of the composer.

2.3 The difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion

Time-scale	Description
<i>Infinite</i>	The ideal time span of mathematical durations such as the infinite sine waves of classical Fourier analysis.
<i>Supra</i>	A time scale beyond that of an individual composition and extending into months, years, decades, and centuries.
<i>Macro</i>	The time scale of overall musical architecture or form, measured in minutes or hours, or in extreme cases, days.
<i>Meso</i>	Divisions of form. Grouping of sound objects into hierarchies of phrase structures of various sizes, measured in minutes or seconds.
<i>Sound object</i>	A basic unit of musical structure, generalizing the traditional concept of note to include complex and mutating sound events on a time scale ranging from a fraction of a second to several seconds.
<i>Micro</i>	Sound particles on a time scale that extends down to the threshold of auditory perception (measured in thousandths of a second or milli-seconds).
<i>Sample</i>	The atomic level of digital audio systems: individual binary samples or numerical amplitude values, one following another at a fixed time interval. The period between samples is measured in millionths of a second (microseconds).
<i>Subsample</i>	Fluctuations of a time scale too brief to be properly recorded or perceived, measured in billionths of a second (nanoseconds) or less.
<i>Infinitesimal</i>	The ideal time span of mathematical durations such as the infinitely brief delta functions.

Table 2.3: Nine time scales of music by Roads (242, p.3).

together with the use of library functions (*Pbind/Ppar* in the SuperCollider example) or multi-threading (in the ChuckK example).

It would be also helpful to clarify the difference between these examples of microsound synthesis and the examples of Ada’s rendezvous & Lua’s bitwise operations. In the examples of Ada’s rendezvous, mutex is internally used to implement Ada’s rendezvous mechanism. In Lua’s bitwise operation, every value inside the Lua’s virtual machine is internally expressed in bits. The problems of abstraction inversion in these languages occur when these languages abstract the lower-level details away in the language design; the problems of abstraction inversion were caused by the inaccessibility of the lower-level details, and they can be easily solved by making these details accessible from the language level, as seen in the later versions of these languages.

Yet, the problems in the examples in SuperCollider and ChuckK differ slightly, in

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

Abstraction-level		Description
<i>note/synth/patch</i>		These correspond to the sound object time-scale in nine musical time-scales by Roads. While a “note” in CSound score file and a “synth” in SuperCollider are clearly close to ‘the traditional concept of note’ as in Figure 2.3, a ‘patch’ in visual computer music languages such as Max/MSP or PureData may also be considered to include the upper level time-scales such as meso or macro, since it normally includes musical control algorithms together within synthesis algorithms.
<i>unit-generator</i>	audio vector	In many computer music systems, unit-generators perform DSP by fixed size vectors of samples called audio vectors (53, p.467). Normally, audio vectors are not directly visible to users.
	sample	Within unit-generators, each sample for signal output is computed by iterating input audio-vectors, (typically in a loop). Yet, some recent computer music programming languages (such as ChuckK) perform sample-by-sample computation without involving audio-vectors.

Table 2.4: The typical abstraction hierarchy in sound synthesis framework design

that the abstractions of the microsound objects and the related manipulations do not exist at all in the underlying sound synthesis framework. The problem is not about the accessibility of the lower-level details, but about the inappropriate abstraction in the underlying software frameworks. Generally speaking, domain-specific languages are normally built on a certain software framework or library¹. Hence, the design of a computer music programming language is significantly influenced by the underlying software framework or library.

Table 2.4 describes the typical sound synthesis software framework design in a unit-generator language. While some languages do not involve the audio-blocks in sound synthesis and compute the sound output sample-by-sample, the direct counterpart objects and manipulations for microsounds are entirely lacking in many frameworks, as seen in SuperCollider or ChuckK; therefore, the problems caused by abstraction inversion in microsound synthesis programming cannot be resolved by recovering the accessibility to the internal lower-level abstractions, since such internal entities do not exist in the

¹As Fowler describes in (114, p.29), “the most common way to build in abstraction is by implementing a library or framework” and “in this view a DSL is a front-end to a library providing a different style of manipulation to the command-query API.”

2.3 The difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion

underlying framework, unlike in the cases of Ada’s mutex and Lua’s bitwise operators¹.

However, there can be found several previous works that contain the internal representations for microsounds in the underlying software frameworks. In (36), Bencina discusses the software design for a real-time granular synthesizer, which is suitable for hosting several different types of granular synthesis techniques, such as Tapped Delay Line Granular Synthesis, Stored Sample Granular Synthesis, and Synthetic Grain Granular Synthesis.

Figure 2.27 describes the structure of the granular synthesizer proposed by Bencina. As seen in this figure, there is an object for *Grain* in this software design. Figure 2.28 and Figure 2.29 describe the event sequences when samples are requested and when a new grain is activated.

Brandt proposed and implemented the concept of *temporal type constructors* in Chronic (56), an internal domain-specific language² for non real-time sound synthesis built upon OCaml (182). Brandt mainly argues that the problems, such as difficulty in expressing FOF synthesis in unit-generator languages, are due to the inappropriate typing and inaccessibility to the lower-level details caused by the unit-generator concept. A brief explanation on type constructors by Brandt himself can be found in (56, p.7) as follows.

“A type constructor builds complex types from simpler ones. For example, C has the ‘pointer to...’ type constructor. We can write this as ‘ α **pointer**’, where α is a free type variable which might be, for example, **int**. Or α can be a non-atomic type, like

¹One may argue that the definition of abstraction inversion should be limited to in exposure of a lower-level function/object. However, it seems that the majority of academic publications use the term ‘*abstraction inversion*’ in a much broader sense. Even the publication by Baker (28), which is often referred to for the definition of abstraction inversion, describes many cases that do not fit within the narrower definition. In addition, unit-generator languages often provide unit-generators dedicated for microsound synthesis techniques. Since there may exist some objects that represent microsounds within such unit-generators, they are not directly accessible. Thus, the situation described in this chapter can be considered abstraction inversion even in the narrower definition. We additionally discuss such an issue with more detail in Appendix IV, Section 9.1.

²“An internal DSL is a DSL represented within the syntax of a general-purpose language. It’s a stylized use of that language for a domain-specific purpose” (114, p.15). “Internal DSLs morph the host language into a DSL itself - the Lisp tradition is the best example of this” and are “often called ‘embedded DSLs’ ” (115).

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

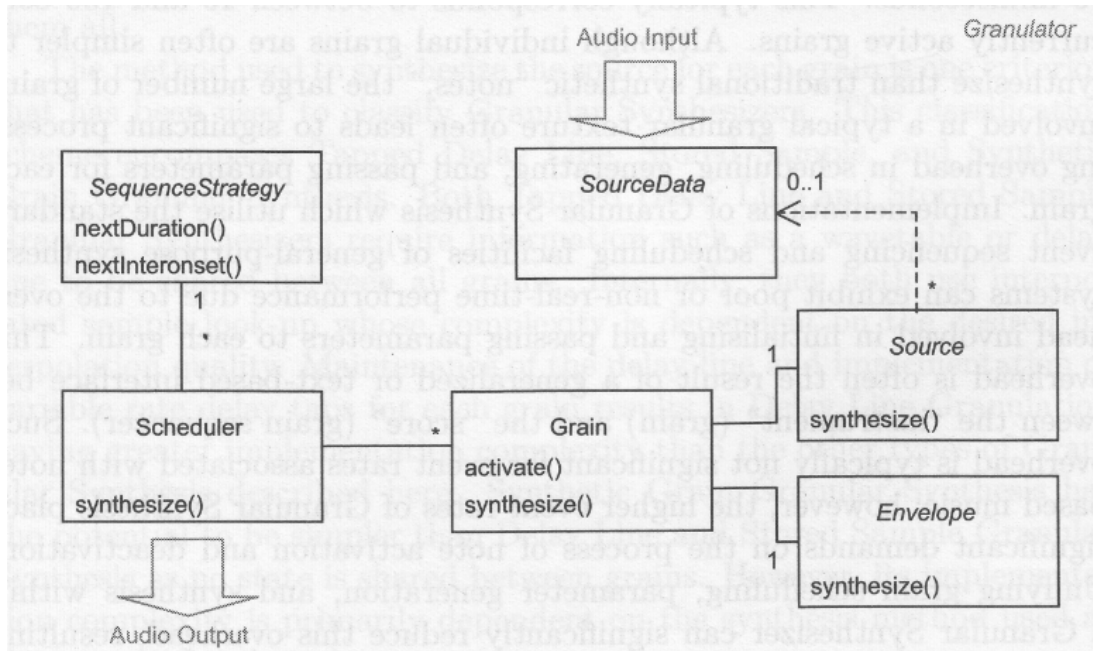


Figure 2.27: Object-oriented granulator structure - as described by Bencina in (36).

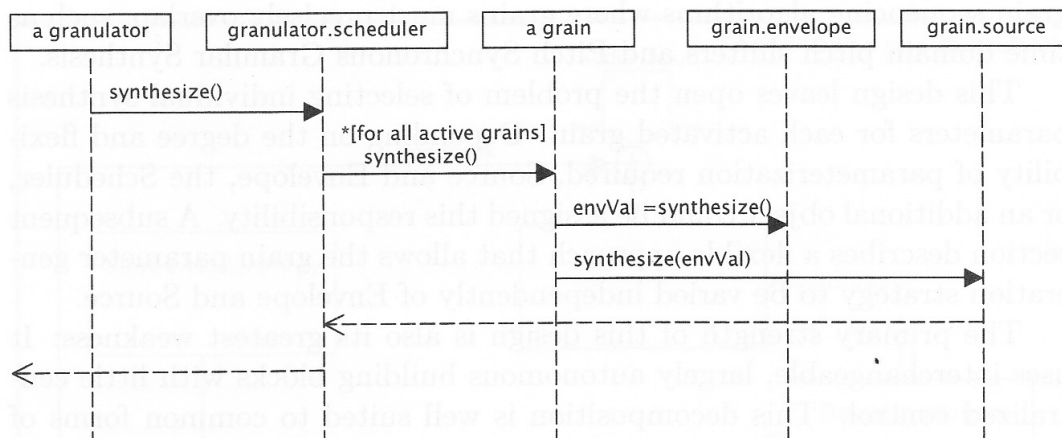


Figure 2.28: Event sequence when samples are requested from a granulator - by Bencina (36).

2.3 The difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion

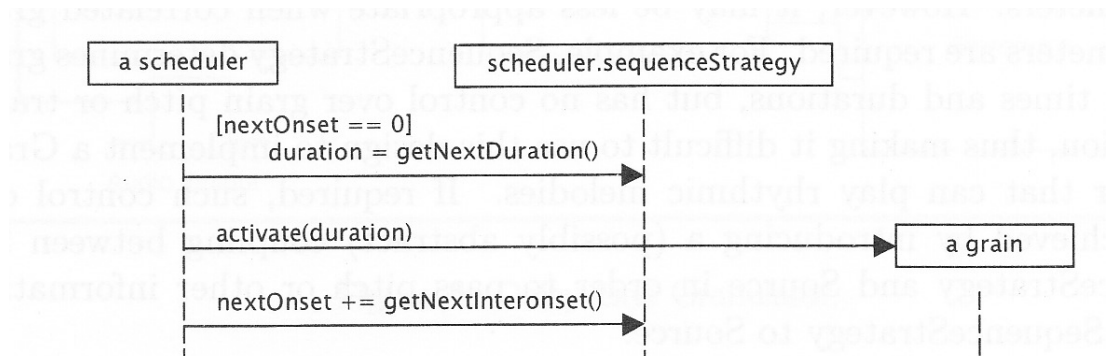


Figure 2.29: Event sequence when it is time to activate a new grain - by Bencina (36).

“**int pointer**”, leading to “**int pointer pointer**”. In this way a single type constructor can build a series of types”.

Brandt explains that a temporal type constructor “is one that introduces a relation to a one-dimensional axis, which we call time” (57) to the concept of type constructor. Figure 2.5 describes three temporal type constructors proposed by Brandt and Figure 2.30 shows several pictorial representations of temporal type constructors by Brandt (56), such as multi-channel audio as *Sample ivec vec* (or an vector of the infinite length vectors of Samples) on the left, chord progression as *Pitch vec event vec* (or a vector of events that consist of the vector of pitches) in the middle, granular sound synthesis as *Sample vec event ivec* on the right. Figure 2.31 describes a very simple example of a score to play sine wave oscillators in Chronic.

Brandt discusses that ‘unit-generators’ are “black-box primitives” in computer music language design and that “if a desired operation is not present, and cannot be represented as a composition of primitives, it cannot be realized within the language”; thus, the motivation for temporal type constructors in Chronic is in recovering such accessibility and also in providing more complex “time-structured types needed for the problems at hand, from ‘audio stream’ to complex score structures” (56, pp.4-5).

Both of the previous research by Bencina and Brandt discuss the software/language design that takes microsound synthesis into account and both contain the internal representations for microsounds. However, Bencina’s software design focuses on gran-

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

Type	Description
α event	an event whose timestamp is of type Time
α vec	a finite vector, indexed by Time
α ivec	an infinite vector, indexed by Time

Table 2.5: Three type constructors in Chronic (56, p.8).

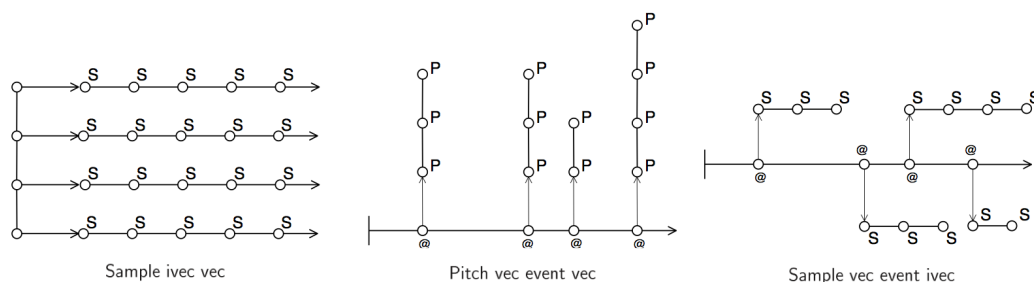


Figure 2.30: A pictorial representation of temporal type constructor examples given by Brandt - multi-channel audio (left), chord progress (middle), and granular synthesis sound (right) (56, pp8-12).

ular synthesizer software rather than computer music programming language design in general; when microsound synthesis techniques are explored in a computer music language, it is necessary to allow a user program to control the scheduling and the definition/instantiation of microsounds; as Bencina's software design does not consider how grains should be created and activated/deactivated by a user program, it still lacks the generality required for computer music language design.

Brandt himself considers that Chronic still leaves 'an open problem' in causality, as its programming model accepts both forward dependency and backward dependency (56, p.77). In other words, since a Chronic program allows the future events and values to influence the past events and values, its programming model may not be easily applicable for real-time synthesis and interaction, while such acausal behaviour may simplify non real-time computer music programs.

Thus, even though both works by Bencina and by Brandt consider the internal representations for microsounds, both of them do not place the programming language design for live computer music within the scope of interest. Further consideration is still required to investigate more appropriate abstractions for real-time sound synthesis and interactive computer music systems. We also briefly describe such an issue as an

2.3 The difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion

```
01: (* score example with sine beeps *)
02:
03: open Vec
04: open Event
05: open Evvec
06: open Chroniclib
07: open Util
08:
09: let fs = 44100.
10: let secs_to_samples s = int (fs *. s)
11:
12: (** handwritten score of note events *)
13: let makenote time_secs db hz dur_sec = (* save a little typing *)
14:   (db, V.const hz (secs_to_samples dur_secs)) @@ (secs_to_samples time_secs)
15:
16: (* score is (db: float, hz: float vec) event vec *)
17: let score = [
18:   (* time dB      Hz    dur *)
19:   makenote 0.  (-.3.)  400.  0.5;
20:   makenote 1.  (-.10.) 400.  0.5;
21:   makenote 1.5 (-.10.) 500.  0.5;
22:   makenote 2.  (-.10.) 600.  0.5;
23: ]
24:
25: (** define synthesizer of sine beeps *)
26: let sinetab = LV.table_of_sine 4096
27: (** synth_beep: (float * float vec) -> float vec *)
28: let synth_beep (db, hz) = (* dur is implicit in length of hz vector *)
29:   let freq = V.scale (1./fs) hz in
30:   let beep = LV.osci_v sinetab freq 0.
31:   in V.scale (L.db_to_amp db) beep
32:
33: (** synthesize, mix and output *)
34: let sound = EV.vmix (V.map (E.lift synth_beep) score)
35: let _ = V.print_floats sound
```

Figure 2.31: A score example with sine beeps in Chronic (56, p.26).

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

HCI related topic in Appendix IV: Additional Discussion, Section 9.2.2. However, we also would like to emphasize such an HCI issue is not a main topic of interest; we focus on the software engineering aspects (abstraction inversion) in this thesis.

2.3.3 The problems in the existing computer music programming languages

The previous sections described the problems related to microsound synthesis in computer music programming. While it is possible to encapsulate whole algorithms within the built-in unit-generators, such an approach is not beneficial when users intend to explore the microsound synthesis algorithms, as their exploration can be significantly limited by the constraints given by the interfaces and functionalities of unit-generators. On the other hand, the implementations of microsound synthesis techniques solely within existing computer music languages exhibit a considerable degree of complexity even when the techniques to implement are conceptually simple.

We discussed this difficulty in the context of software anti-pattern of abstraction inversion, which is caused by the lack of objects and manipulations in the software sound synthesis framework underlying computer music languages. In many computer music languages, this leads certain programming patterns to model each microsound as a note-level object and to perform overlap-add by scheduling such note-level objects. As the programming patterns involve the combination of higher-level abstractions (note-level objects and scheduling of note-level objects), because of the inaccessibility and/or lack of lower-level abstractions (microsounds objects and related manipulations), the difficulty involved in microsound synthesis programming can be assessed as abstraction inversion.

While previous research on software design for a granular synthesizer by Bencina directly involves the objects for grains internally, it focuses on a stand-alone synthesizer application and does not take the programming language design into account. Bencina's Chronic language (36) is appropriate for certain kinds of computer music applications, yet it still lacks the consideration for how such software design can be integrated into computer music language design for real-time sound synthesis and interactive applications; as Brandt himself admits, it has a significant problem in causality and does not

2.3 The difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion

provide appropriate abstraction for such a domain(56, p.77).

Thus, there is still a significant necessity to investigate a more appropriate software sound synthesis framework design, which can reduce the difficulty in microsound synthesis programming, to facilitate creative exploration in the domain of microsounds in interactive, real-time computer music languages.

In addition, *GEN*, one of the features in the recent version of Max in which users can describe sound synthesis/processing algorithms and compile it into a native code in C++ or a Max/MSP object, may also be considered an example of abstraction inversion in a visual programming language, as GEN is another programming environment integrated seamlessly within Max. Since single sample feedback is not possible by Max as it performs DSP by audio vectors¹(53, p.467), “higher-level structure to implement a low-level concept”²(319); a program must be written within GEN, another programming environment, (high-level structure), to achieve single sample feedback (a lower-level concept). Some other sound synthesis/processing techniques such as physical modelling may also require GEN to be implemented within Max.

Such a problem of single sample feedback in Max, which leads to the integration of GEN within the programming environment, is largely due to the utilization of audio vectors in real-time DSP. It is also possible to view this problem as an issue of the sound synthesis framework design. While this may be an interesting topic, which may be beneficial to some readers, we do not discuss it further as this thesis focuses on abstraction inversion in microsound synthesis programming. However, such a view may be beneficial some readers as a topic for further discussion in sound synthesis framework design.

¹Of course, if the size of audio vectors is set to 1 sample, even computer music languages that compute audio by audio vectors can achieve single sample feedback, yet this can also lead to significant damage to performance efficiency, which is the reason why audio vectors are utilized.

²This explanation of abstraction inversion is by Waroquiers, which can be found in (319) must be utilized. Section 9.1 in Appendix IV additionally discusses the definition of abstraction inversion found in previous works.

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

2.4 The problems as design opportunities

In this chapter, three problems in computer music programming language design were described: (1) the insufficient support for dynamic modification of a computer music program, (2) the insufficient support for precise timing behaviour and other features with respect to time, and (3) the difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion.

The insufficient support for dynamic modification of a computer music program can be a significant obstacle for programming activity such as rapid-prototyping and live-coding. While computer music programming languages should offer better support for dynamic modification both at the levels of compositional algorithms and sound synthesis, existing computer music languages fail to provide a terse and consistent programming model that can be applied to both levels.

Regarding the insufficient support for precise timing behaviour and other features with respect of time, many computer music languages still fail to provide sample-rate accurate timing behaviour, which is of significant importance both at the rhythmic level and at the acoustic level. While some of the recent computer music languages such as *ChucK* and *LuaAV* provide sample-rate accuracy in timing based on synchronous behaviour, these languages still have a significant problem: real-time DSP can be easily suspended by a time-consuming task. Furthermore, while some recent languages consider other desirable features with respect to time (e.g., timed communications and timing constraints), there is no computer music language that implements such desirable features with sample-rate accurate timing precision, yet; better support for precise timing behaviour and other features with respect to time is still demanded in computer music language design.

Programming microsound synthesis techniques can involve a considerable degree of difficulty even when the microsound synthesis techniques to implement are simple in their concepts. This problem is an example of the software anti-pattern called abstraction inversion, which is caused by the lack of objects and manipulations for microsound synthesis in the underlying sound synthesis framework. While some researchers suggest that software design with internal representations for microsounds can be beneficial for this issue, as the designs proposed by related works only focus on the implementation

2.4 The problems as design opportunities

of stand-alone synthesizer applications or non real-time sound synthesis, further investigation is still required for the sound synthesis framework design for computer music languages, taking interactivity and real-time sound synthesis into consideration.

These problems provide significant motivation for the design and development of a new computer music programming language; as seen in the history of the evolution of computer music languages, the problems arisen in creative practices often not only clarify the limitation of existing computer music languages but also indicate the direction for further research.

2. BACKGROUND AND MOTIVATION: THREE PROBLEMS IN TODAY'S COMPUTER MUSIC PROGRAMMING LANGUAGE DESIGN

3

Design: LC, a Mostly-strongly-timed Prototype-based Computer Music Programming Language that Integrates Objects and Manipulations for Microsound Synthesis

In this chapter, we describe LC, a new computer music programming language, the design of which was motivated by the three problems in computer music language design described in the previous chapter.

Generally speaking, by adopting the concept of prototype-based programming at both levels of compositional algorithms and sound synthesis, LC supports dynamic modification of a computer music system to a significant degree, with a terse and consistent program model.

LC also proposes the concept of mostly-strongly-timed programming in its language design, which extends the strongly-timed programming concept by the explicit

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

switching between synchronous/non-preemptive context and asynchronous/preemptive context, so that time-consuming tasks can be preempted without suspending real-time DSP. Additionally, other desirable features with respect to time, such as timing constraints and time-tagged message communication, are integrated together with sample-rate timing precision.

While LC is still equipped with unit-generators, LC also directly integrates the objects and manipulations for microsound synthesis in the abstraction of its sound synthesis framework. Such design makes it possible to provide a simple and terse programming model for microsound synthesis techniques, without involving abstraction inversion, which can lead to a considerable complexity in microsound synthesis programming. The collaboration between this novel abstraction and the traditional unit-generator concept is also considered in its design so that it can be performed as seamlessly as possible.

The following sections describe the core features of LC's language design as listed above, after a brief description of basic language design such as typing and control structures.

3.1 The basic language features

3.1.1 The grammar

Figure 3.1 describes the grammar of LC¹ in EBNF².

3.1.2 Operators and primitive types

Table 3.2 describes the operators and the operator precedence in LC. There is no suffix increment/decrement ('i++' or 'i--' as in Java) and prefix increment/decrement ('++i' or '--i' as in Java) and ternary conditional ('x > y ? x : y' as in Java) in the current prototype of LC.

Table 3.3 describes the primitive types available in LC. All the objects are allocated in the heap and variables just keep the references to the heap objects.

¹As LC is still just a proof-of-concept prototype, this grammar is still only for the current prototype. There is a plan for further extension and modification.

²As proposed by ISO/IEC 14977 standard (279)

3.1 The basic language features

PROGRAM	=	{ STMT } ;
STMT	=	((VAR_LIST GVAR_LIST IF WHILE LABELED_WHILE FOR LABELED_FOR BREAK CONTINUE RETURN SYNC ASYNC WITHIN TRY_CATCH THROW RECEIVE EXPR EMPTY_STMT), ';') COMPOUND_STMT ;
COMPOUND_STMT	=	{ ' ', { STMT }, ' ' } ;
EMPTY_STMT	=	ε ;
VAR_LIST	=	'var', IDENT, ['=', EXPR], { ',', IDENT, ['=', EXPR] } ;
GVAR_LIST	=	'global', IDENT, ['=', EXPR], { ',', IDENT, ['=', EXPR] } ;
IF	=	'if', '(', EXPR, ')', STMT, ['else', STMT] ;
LABELED_WHILE	=	IDENT, ':', WHILE ;
WHILE	=	'while', '(', EXPR, ')', STMT ;
LABELED_FOR	=	IDENT, ':', FOR ;
FOR	=	'for', '(', FOR_INIT, ';', FOR_COND, ';', FOR_LOOP, ')', STMT ;
FOR_INIT	=	[VAR_LIST (EXPR, { ',', EXPR })] ;
FOR_COND	=	[EXPR] ;
FOR_LOOP	=	[(EXPR, { ',', EXPR })] ;
RETURN	=	'return', [EXPR] ;
BREAK	=	'break', [IDENT] ;
CONTINUE	=	'continue', [IDENT] ;
SYNC	=	'sync', STMT ;
ASYNC	=	'async', STMT ;
WITHIN	=	'within', '(', EXPR, ')', STMT ['timeout' STMT] ;
TRY_CATCH	=	'try', STMT, { 'catch', '(', IDENT, [':', SYMBOL, { ',', SYMBOL }], ')', STMT }- ;
THROW	=	'throw', EXPR ;
RECEIVE	=	'receive', '(', 'var', IDENT, ')', { ' ', { (RECV_CASE RECV_COND 'default'), ':', STMT }, ' ' } ;
RECV_CASE	=	'case', EXPR ;
RECV_COND	=	'cond', EXPR ;
EXPR	=	SEND_EXPR ;
SEND_EXPR	=	ASSIGN_EXPR, { '<-', ['@', ASSIGN_EXPR, ',',] , ASSIGN_EXPR } ;
ASSIGN_EXPR	=	LOR_EXPR, { ('=' '+=' '-=' '*=' '/=' '%=' '&=' ' =' '^=' '<=>' '>>=' '>>>='), EXPR } ;
LOR_EXPR	=	LAND_EXPR, { ' ', LAND_EXPR } ;
LAND_EXPR	=	BITOR_EXPR, { '&&', BITOR_EXPR } ;
BITOR_EXPR	=	BITXOR_EXPR, { ' ', BITXOR_EXPR } ;
BITXOR_EXPR	=	BITAND_EXPR, { '^', BITAND_EXPR } ;
BITAND_EXPR	=	EQL_EXPR, { '&', EQL_EXPR } ;
EQL_EXPR	=	REL_EXPR, { ('==' '!='), REL_EXPR } ;
REL_EXPR	=	CONCAT_EXPR, { ('<' '<=' '>' '>=' 'instanceof'), CONCAT_EXPR } ;

Table 3.1: The grammar of LC.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```

CONCAT_EXPR = BITSHIFT_EXPR, { '..', BITSHIFT_EXPR } ;
BITSHIFT_EXPR = ADD_EXPR, { ( '<<' | '>>' | '>>>' ), ADD_EXPR } ;
ADD_EXPR = MUL_EXPR, { ( '+' | '-' ), MUL_EXPR } ;
MUL_EXPR = DUR_EXPR, { ( '*' | '/' | '%' ), DUR_EXPR } ;
DUR_EXPR = PRIMARY, { ':', ( 'minute' | 'second' | 'ms' | 'samp' ) } ;
PRIMARY = [ PREFIX ], ( SYMBOL | STRING | INTEGER | DOUBLE | 'true' |
    'false' | 'now' | 'null' | 'SampleRate' | NEW_EXPR | ARRAY_EXPR |
    FUNCTION_EXPR | PATCH_EXPR | UPDPATCH_EXPR ( IDENT, [ POSTFIX ] ) |
    ( GVAR, [ POSTFIX ] ) ) | ( '(', EXPR, ')', [ POSTFIX ] ) ;

PREFIX = '-' | '!' | '+' | ' ';
POSTFIX = { ( '.', IDENT ) | ( '[', EXPR, ']' ) | FUNCTION_CALL | NEW_THREAD |
    METHOD_CALL } ;

FUNCTION_EXPR = 'function', '(', [ PARAM_LIST ], ')', STMT, [ POSTFIX ] ;
PARAM_LIST = ('var', IDENT, [ '=', EXPR ], { ',', IDENT, [ '=', EXPR ] } ) ;

ARRAY_EXPR = '[', EXPR, { ',', EXPR }, ']', [ POSTFIX ] ;

NEW_EXPR = 'new', IDENT, '(', [ ARG_LIST ], ')', [ POSTFIX ] ;
FUNCTION_CALL = '(', [ ARG_LIST ], ')', [ POSTFIX ] ;
NEW_THREAD = '@', '(', [ ARG_LIST ], ')', [ POSTFIX ] ;
METHOD_CALL = '->', ( FUNCTION_CALL | THREAD_CREATION ), [ POSTFIX ] ;
ARG_LIST = ( [ IDENT, ':' ], EXPR ), { ',', [ IDENT, ':' ], EXPR } ;

PATCH_EXPR = 'patch', '{', PSTMTS, '}', [ POSTFIX ] ;
UPDPATCH_EXPR = 'update_patch', '(', EXPR, ')', '{', PSTMTS, '}', [ POSTFIX ] ;
PSTMTS = { PSTMT ';' } ;
PSTMT = [ IDENT, ':' ], EXPR, { ( '=>', '=|' ), [ IDENT, ':' ], EXPR } ;

```

NOTE: Each of SYMBOL, STRING, INTEGER, DOUBLE, IDENT and GVAR corresponds to a symbol value, a string value, an integer value, a floating point value, an identifier, and a global variable name (*\$varname*), respectively.

Table 3.1: The grammar of LC in EBNF (continued).

3.1 The basic language features

	Operator	Description
1	() [] . -> ::	grouping, array access, member access, method call, and duration operator
2	! ~ -	unary operators (logical negation, one's complement, negative)
3	* / %	multiplication, division, modulo
4	+ -	addition, subtraction
5	<< >>	bitwise shift left, bitwise shift right
6	..	string concatenation operator
7	< <= > >= instanceof	less than, less than or equal, greater than, greater than or equal, instanceof
8	== !=	equal and not equal
9	&	bitwise and
10	^	bitwise exclusive or
11		bitwise or
12	&&	logical and
13		logical or
14	= += -= *= /= %= &= = ^= <<= >>= >>>=	assignment operators
15	<-	inter-thread messaging

Table 3.2: The relative precedence levels of operators in LC.

Value Types	
Type	Description
boolean	boolean (true or false)
integer	integer (64-bit)
float	floating point (64-bit)
symbol	a symbol internally bound to its own unique ID. A symbol starts with a backslash (e.g., \hanning, \Exception)
time	the logical time in samples elapsed since the virtual machine start-up (64-bit)
duration	duration in samples (64-bit)

Reference Types (built-in native objects)	
Type	Description
Function	a first class function object
Exception	an exception object
String	an immutable string object
Samples	an immutable vector of samples
SampleBuf	a mutable vector of samples
Array	an array object
Thread	a thread object
Patch	a patch object
Table	a table object
UGen	unit-generator objects

Table 3.3: The data types available in LC.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
01: //local variables
02: var a = 1234; //assigning an integer value.
03: var b = 0.123; //assigning a float value.
04:
05: //global variables. use 'global' for declaration.
06: global c;
07: c = 10;
08: //$identifier' can be used for a global variable.
09: //(this '$d' is the same variable as 'global d').
10: $d = "I am a global variable.";
11: //the below prints out 'I am a global variable.'
12: println($d)
13: //the below also prints out 'I am a global variable.'
14: global d;
15: println(d);
```

Figure 3.1: Local variables and global variables in LC.

3.1.3 Typing and variable scope

LC has both local variables and global variables as shown in Figure 3.1. By using ‘\$identifier’, a global variable can be accessed without declaration. As LC is dynamically-typed, types are evaluated at runtime as in Lua (150)(152) and Ruby (294), whereas statically-typed languages such as Java (128) and ChuckK (312)(314) verify the type safety of a program during compilation.

LC is a strongly-typed language, which “detects when two types are compatible, throwing an error or coercing the types if they are not” and does little implicit type conversion (290, p.141), like Ruby, Python (301), and Self (300), whereas weakly-typed languages such as PHP (181) and Perl (310) perform implicit type conversion as much as possible¹. Figure 3.2 shows a brief example of dynamic-typing and strong-typing in LC.

3.1.4 Control structure

LC supports standard control structures as seen in other imperative programming languages. Currently, LC supports **if**, **else**, **for**, and **while**. The **break** and **continue**

¹For instance, Ruby throws an exception for such an expression as “456” + 7 (string + integer), while PHP returns 463 by implicitly converting “456” to an integer value of 456. Some other weakly-typed language may instead return string “4567”, by converting 7 to a string “7”.

```
01: //dynamic-typing
02: var a = 1234 ; // assigning an integer value.
03: var b = "567"; // assigning a reference to a String object.
04:
05: //strong-typing
06: //'integer + string' causes a runtime error
07: var c = a + b;
08: //it is required to use the string concatenation operator
09: //to convert an integer value to a string value.
10: var d = a .. b;
11: //this prints out "1234567".
12: println(d);
```

Figure 3.2: An example of dynamic-typing and strong-typing in LC.

statements can be labelled. Figure 3.3 describes a simple example of the control structures in LC.

3.1.5 Lexical closure

If a programming language can enclose a function in another function and allows full access to local variables to the enclosing function from the enclosed function, the language is said to have the feature of *lexical scoping* (150, p.47). Furthermore, if a language treats a function as a first-class value¹ and a function can capture variables in such lexical context, the language is said to have the feature of *lexical closure*. Some languages such as Pascal (321) and Algol (215) have lexical scoping, but no lexical closure. The languages such as Scheme (6), JavaScript (127), and Lua (152) are equipped with the lexical closure feature.

LC supports lexical closure. Figure 3.4 describes a simple example of a lexical closure in LC. As seen between lines 10-11, the local variable *i* (line 02) is captured by the function (lines 04-07)². LC also supports keyword arguments (or named parameters) and default values as described in Figure 3.5.

¹Generally speaking, a function is said to be a first-class value if “a function is a value with the same rights as conventional values like numbers and strings”. A first-class function “can be stored in variables (both global and local)” and “be passed as arguments, and can be returned by other functions” (151, p.18).

²As the current prototype of LC doesn’t provide a syntax sugar for function definition, an anonymous function is assigned to the variable ‘newCounter’ in this example.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

<pre> 01: //if-else 02: var a = 10; 03: if (a % 2 == 0){ 04: println("variable 'a' is even"); 05: } 06: else { 07: println("variable 'a' is odd"); 08: } 09: 10: //for loop 11: for (var i = 0; i < 10; i += 1){ 12: println("count:" .. i); 13: } 14: 15: //while loop 16: var i = 0; 17: outer_loop: 18: while(true){ 19: i += 1; 20: if (i >= 5){ 21: break outer_loop; 22: } 23: for(var j = 0; j < 5; j +=1){ 24: if (i < 3) continue outer_loop; 25: println("i:" .. i ", j:" .. j); 26: } 27: } </pre>	<pre> variable 'a' is even count:0 count:1 count:2 count:3 count:4 count:5 count:6 count:7 count:8 count:9 i:3, j:0 i:3, j:1 i:3, j:2 i:3, j:3 i:3, j:4 i:4, j:0 i:4, j:1 i:4, j:2 i:4, j:3 i:4, j:4 </pre>
---	---

Figure 3.3: An example of control structures in LC(left) and its output(right).

```

01: var newCounter = function(){
02:     var i = 0;
03:     // anonymous function
04:     return function(){
05:         i = i + 1;
06:         return i;
07:     };
08: };
09: var c1 = newCounter();
10: println(c1()); // 1;
11: println(c1()); // 2;

```

Figure 3.4: The examples of lexical closure in LC.

3.1 The basic language features

```

01: var f = function(var value = 10, message = "hello"){
02:   println("value:" .. value .. ", message:" .. message);
03: };
04: //using the default parameters.
05: //this prints out "value:10, message:hello".
06: f();
07:
08: //give only 'message' for the argument.
09: //this prints out "value:10, message:have a nice day".
10: f(message:"have a nice day");
11:
12: //change the order of the arguments by giving the names.
13: //this prints out "value:99, message:good night".
14: f(message:"good night", value:99);

```

Figure 3.5: An example of default parameters/keyword arguments in LC.

The constructor definition	
Exception(type=Exception, message = null, value = null, parent = null)	
Arguments	
type	a symbol value for the exception type.
message	a string value, which is used for the user-friendly message.
value	any value, which can be used to retain detailed information for the exception.
parent	a parent exception in the exception hierarchy.

Table 3.4: The constructor definition of Exception object in LC

3.1.6 Exception handling

LC has an exception handling mechanism. *Exception* is provided as in Table 3.4. Both built-in exceptions and user-defined exceptions can be used. Exceptions are organized hierarchically. Figure 3.6 shows the built-in exception hierarchy in the current prototype. Figure 3.7 describes how to throw and catch an exception in LC. Multiple exceptions can be caught by one catch block as in example (3) and user-defined exceptions can also have its own parent in the exception hierarchy as in example (5). This exception hierarchy is realized by delegation mechanism (as described in the later section) and thus it can be specified at runtime; when a catch block examines the type of the exception, LC also traverses the ancestor exceptions in the hierarchy dynamically so to mimic the exception handling in class-based programming languages.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
\Exception
+-- \StandardError
    +-- \ArithmeticError
        +-- \ZeroDivisionError
    +-- \AttributeError
    +-- \MemoryError
    +-- \ThreadError
    +-- \TimeError
    +-- \TypeError
    +-- \ValueError
```

Figure 3.6: The built-in exception hierarchy example in LC (the prototype version).

3.1.7 Tail call optimization

A *tail call* is a function call that “happens when a function calls another as its last action so it has nothing else to do” (151, p.58). Figure 3.8 shows a very simple example of a tail call. As shown, after calling $g(x)$, the function immediately returns and has nothing else to do.

When a tail call is performed, “the program does not need to return to the calling function when the called function ends. Therefore, after the tail call, the program does not need to keep any information about the calling function in the stack” (151, p.58). For instance, in the Figure 3.8 example, assume another function h made a call to function f . Then, function f calls function g . After calling g , as there is nothing left to do in function f , the code can immediately return to function h , not to f , without changing the result of the entire function call.

If a compiler generates such a code (or an interpreter behaves as above) in such a situation, it is said that the compiler (or interpreter) supports *proper tail calls* and this optimization is called *tail call optimization*. The benefit of tail call optimization is that it can avoid the unnecessary allocation of the extra stack space. As functional programming languages often lack the loop structure and therefore require a recursive call, tail call optimization is an essential language feature in functional programming languages. This is the same for an imperative programming language to describe an algorithm that involves a recursive call.

LC supports tail-call optimization in the form of *return any_function_call(x)*. If the

```
try-catch example(1)
01: //try-catch example(1)
02: try {
03:     //below zero division throws an \ZeroDivisionError exception.
04:     println("10 / 0 = " .. 10 / 0);
05: }
06: //the below catches the exception.
07: catch(e:\ZeroDivisionError){
08:     println("in the catch block.");
09:     println("exception type = " .. e.type);
10: }
```

The output of try-catch example(1)
in the catch block.
exception type = \ZeroDivisionError

```
try-catch example(2)
01: //try-catch example(2)
02: try {
03:     //below zero division throws
04:     //an \ZeroDivisionError exception.
05:     println("10 / 0 = " .. 10 / 0);
06: }
07: //the above \ZeroDivisionError will be caught by the below catch block,
08: //since \ArithmeticException is its ancestor in the hierarchy.
09: catch(e:\ArithmeticException){
10:     println("in the 1st catch block.");
11:     println("exception type = " .. e.type);
12: }
13: //the code never reach below.
14: catch(e:\ZeroDivisionError){
15:     println("in the 2nd catch block.");
16:     println("exception type = " .. e.type);
17: }
```

The output of try-catch example(2)
in the 1st catch block.
exception type = \ZeroDivisionError

Figure 3.7: An example of exception handling in LC.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

try-catch example(3)

```
01: //try-catch example(3)
02: try {
03:     //below zero division throws an \ZeroDivisionError exception.
04:     println("10 / 0 = " .. 10 / 0);
05: }
06: //the above exception will be caught by the below catch block,
07: //A catch block can catch multiple exception.
08: catch(e:\TimeError, \ZeroDivisionError){
09:     println("in the catch block.");
10:     println("exception type = " .. e.type);
11: }
```

The output of try-catch example(3)

in the catch block.

exception type = \ZeroDivisionError

try-catch example(4)

```
01: //try-catch example(4)
02: try {
03:     //throwing a user-defined exception.
04:     throw new Exception(\MyException, "user defined exception", 12345);
05: }
06: catch(e:\Exception){
07:     println("in the catch block.");
08:     println("exception type = " .. e.type);
09:     println("exception message = " .. e.message);
10:     println("exception value = " .. e.value);
11:     if (e.parent != null){
12:         println("exception parent = " .. e.parent.type);
13:     }
14:     else {
15:         println("no parent exception");
16:     }
17: }
```

The output of try-catch example(4)

in the catch block.

exception type = \MyException

exception message = user defined exception

exception value = 12345

no parent exception.

Figure 3.7: An example of exception handling in LC (continued).

try-catch example(5)

```
01: //try-catch example(5)
02: try {
03:     //a user-defined exception with the parent exception specified.
04:     throw new Exception(\MyException, parent:new Exception(\StandardError));
05: }
08: catch(e:\StandardError){
09:     println("in the catch block.");
10:     println("exception type = " .. e.type);
11:     println("exception message = " .. e.message);
12:     println("exception value = " .. e.value);
13:     if (e.parent != null){
14:         println("parent exception = " .. e.parent.type);
15:     }
16:     else {
17:         println("no parent exception");
18:     }
19: }
```

The output of try-catch example(5)

```
in the catch block.
exception type = \MyException
exception message = user defined exception
exception value = 12345
parent exception = \StandardError
```

Figure 3.7: An example of exception handling in LC (continued).

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
01:  var f = function(var x){  
02:      return g(x);  
03:  };
```

Figure 3.8: A simple tail call example in LC.

expression that follows *return* statement is a function call, LC generates the code with tail-call optimization. The same kind of the specification can be seen in Lua (150)(151).

3.1.8 Strongly-timed programming

LC proposes a new programming concept of *mostly-strongly-timed programming*, which extends the concept of strongly-timed programming proposed by Wang in his ChuckK programming language (312). While LC is a dynamically-typed language, the basic concept and the primitive types used for strongly-timed programming are still applicable to LC; LC also provides both types of *time* and *duration*, each of which is equivalent to ChuckK's *time* and *dur*. The value of *time* represents the time elapsed since the start of the virtual machine and *duration* represents the duration. Both are based on LC's internal logical synchronous time with sample rate accuracy.

Table 3.5 describes the arithmetic operations for these types. A simple example of strongly-timed programming in LC is shown in Figure 3.9. As described in the example, the special variable *now* represents the current logical synchronous time in the system and the assignment to *now* lets the current thread sleep until the given wake-up time. The thread is woken up with sample-rate accuracy in timing.

The sample-rate accurate timing behaviour in LC also makes it possible to compute the output samples directly without depending on the unit-generators or microsound synthesis objects in LC. Figure 3.10 shows an example that directly generates white noise just by sample-by-sample computation with strongly-timed programming.

3.1.9 Lightweight concurrency and multitasking

Generally speaking, native threads (operating system's threads) are considered *heavy-weight* as it takes a considerable amount of time and memory space¹ for instantiation

¹For instance, Xie describes that "a thread on Red Hat Enterprise Linux (RHEL4), needs a 10MB stack, which means at least 10MB is leaked if you haven't joined it" (Avoiding memory

3.1 The basic language features

type	op	type		result type	commute
duration	+	duration	->	duration	yes
duration	-	duration	->	duration	no
duration	*	float	->	duration	yes
duration	/	float	->	duration	no
duration	/	duration	->	float	no
time	+	duration	->	time	yes
time	-	duration	->	time	no
time	-	time	->	duration	no

Table 3.5: Arithmetic operations on time and duration in LC.

A strongly-timed programming example

```

01: //store the current time as the task start time.
02: //'now' represents the current VM time in samples.
03: var startTime = now;
04:
05: //loop for 10 seconds
06: var endTime = now + 10::second;
07: while (now <= endTime){
08:     println("elapsed " .. (now - startTime) / SampleRate .. " sec");
09:
10:     //sleep for 1 second
11:     now += 1::second;
12:     //alternatively you can write as below, too
13:     //now = now + 1::second
14: }

```

The output from the above example

```

elapsed 0 sec
elapsed 1 sec
elapsed 2 sec
elapsed 3 sec
elapsed 4 sec
elapsed 5 sec
elapsed 6 sec
elapsed 7 sec
elapsed 8 sec
elapsed 9 sec
elapsed 10 sec

```

Figure 3.9: A strongly-timed programming example in LC.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
01:  //generating white noise directly without unit-generators
02:  while(true){
03:      var out = Rand(-1.0, 1.0);
04:      PanOut(out, 0);
05:      now += 1::samp;
06:  }
```

Figure 3.10: Directly computing output samples without unit-generators in LC.

and activation, whereas green threads (or software threads implemented by a virtual machine) are considered *lightweight* and perform significantly better in thread activation and synchronization, with less memory space, as reported in (284). It is also reported that the *Erlang* programming language, which uses software processes¹, could host 20 million lightweight tasks at once during a benchmark test².

Such features of lightweight concurrency are also appropriate to achieve precise timing behaviour with sample-rate accuracy, as described in Section 2.2; many computer music programming languages utilize coroutines (as in LuaAV) or its own software threads (as in ChuckK), since the green threads can suspend/switch threads with very low overhead in comparison with native threads. LC also realizes multi-tasking by green threads for this reason.

To facilitate multi-tasking, LC provides a simple syntax to create and start a new thread from a user program. Figure 3.11 describes a simple example of multi-threading in LC. As shown, a new thread object can be created by placing ‘@’ before the argument list. This returns a new thread object, which is still not activated, instead of calling a function immediately. The thread object can be started by calling the *start()* method.

leaks in POSIX thread programming, <http://www.ibm.com/developerworks/linux/library/l-memory-leaks/index.html>).

¹Erlang’s process is implemented as software process as well as green threads.

²Ulf Wiger (14 November 2005). ”Stress-testing erlang”, <https://groups.google.com/forum/#!original/comp.lang.functional/5kldn1QJ73c/T3py-yqmtzMJ> , accessed on Oct-02-2013.


```
01: //any function can be used as an entry point of a thread.
02: var f = function(var name, message, period, repeat){
03:   for (var i = 0; i < repeat; i+=1 ){
04:     println(name .. ":" .. message .. ", count=" .. i);
05:     now += period;
06:   }
07: };
08:
09: //placing '@' before the argument list will create a thread object,
10: //instead of a function call.
11: var john = f@("John Bull", "How are you?", 1::second, 3);
12: var jane = f@("Jane Doe ", "Fine. And you?", 1.5::second, 2);
13:
14: //starting each thread.
15: //the first argument must be thread object itself: Thread.start(self).
16: john.start(john);
17: //Yet a syntax sugar is provided to abbreviate the first 'self' argument.
18: //The -> operator will automatically add the 'self' argument as below.
19: jane->start();
```

The output of the above example

```
John Bull:How are you?, count=0
Jane Doe :Fine. And you?, count=0
John Bull:How are you?, count=1
John Bull:How are you?, count=2
Jane Doe :Fine. And you?, count=1
```

Figure 3.11: A simple multi-threading example in LC.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

3.2 The Core Language Features

3.2.1 Prototype-based programming

Prototyped-based programming is relatively newer in comparison with class-based programming. The first prototype-based programming language was *Self* developed in 1986 (300), whereas the first class-based programming language, *Simula*, was developed in 1967 (85). Yet, a number of prototype-based languages have developed since then. While the list includes programming languages such as Omega (47)(48), Kevo (286)(287), GlyphicScript (264), and NewtonScript (272)(270), perhaps the most well-known prototyped-based languages today would be JavaScript (127) and Lua (152), due to their recent rapid popularization among programmers.

While prototype-based programming is a concept that belongs to the object-oriented programming paradigm, the concept significantly differs from that of class-based languages such as C++ (283), Java (128), etc. In prototype-based programming languages, there are no classes¹. Dony describes prototype-based languages: “are all based on a similar set of basic principles: object-centered representation, dynamic addition (deletion) of slots², cloning and message delegation” (101). Yet, Donny et al. also pointed out that “current prototype-based languages differ in the semantics of object representation, object creation, object encapsulation, object activation and object inheritance” (102) and prototype-based languages exhibit considerable variations in the language design.

Such features make prototype-based programming languages highly flexible and tolerant against the dynamic modification of a program and also favourable for the support of rapid-prototyping. As discussed in the earlier chapter, while some of the recent computer music languages support such dynamic modification, they still have drawbacks at least at the levels of either compositional algorithms or sound synthesis and merely provide a consistent programming model at these two levels. To provide a

¹For this reason, prototype-based programming is often referred as ‘classless’ languages.

²Slots treats the variables and methods in the same way. Some languages such as OBJECT-LISP distinguish these two; for instance OBJECT-LISP uses the ‘*defobjfub*’ function for method definition and uses ‘*have*’ method for attribute definition, while other languages such as Self instead simply provide *slots*.

prototype-based programming example(1)

```
01: //create a Table object
02: var obj = new Table();
03:
04: //then, attach values and functions to its slots
05: obj.name = "John";
06: obj.age = 34;
07: obj.print = function(var self){
08:   println("name : " .. self.name .. ", age:" .. self.age);
09: };
10:
11: //calling the method.
12: obj.print(obj);
13: //alternatively, use -> operator to abbreviate 'self'.
14: obj->print();
```

The output from the above example(1)

```
name :John, age:34
```

Figure 3.12: Table object examples in LC(1).

simple and consistent programming model for dynamic modification, which is a significant factor in supporting rapid-prototyping and live-coding, LC adopts the concept of prototype-programming at both levels of compositional algorithms and sound synthesis.

3.2.1.1 Prototype-based programming at the level of compositional algorithms

Table object. LC provides the *Table* object for prototype-based programming at the level of compositional algorithms. LC's table object is similar to *Object* in JavaScript or *Table* in Lua. As in many other prototype-based programming languages, this *Table* object can be attached to any slot dynamically at runtime after creation. Figure 3.12 describes an example of how *Table* object can be created and initialized. In example (2), two variables ('_name' and '_age') are made invisible and inaccessible from outside by using a lexical closure, and the accessor methods (setName and setAge) must be used to change the values.

Similarly as in Lua (150), 'Table' in LC is implemented as a key-value map. Symbol, string, integer, float, and reference types can be used as a key to store/access any value.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

prototype-based programming example(2)

```
01:  //mimicking 'private' instance fields by lexical closure.
02:  var newInfo = function(var name, age){
03:      var this = new Table();
04:      var _name = name;
05:      var _age = age;
06:      this.print = function(var self){
07:          println("name : " .. _name .. " ", age: " .. _age);
08:      };
09:
10:      this.setName = function(var self, name){
11:          _name = name;
12:      };
13:
14:      this.setAge = function(var self, age){
15:          _age = age;
16:      };
17:      return this;
18:  };
19:
20:  var objA = newInfo("Jane", 11);
21:  var objB = newInfo("Mary", 92);
22:
23:  //'->' operator automatically add 'self' as a first argument.
24:  //(below is equivalent to objA.print(objA).)
25:  objA->print();
26:  objB->print();
27:
28:  objA->setName("Jennifer");
29:  objA->setAge(23);
30:  objA->print();
```

The output from the above example(2)

```
name :Jane, age:11
name :Mary, age:92
name :Jennifer, age:23
```

Figure 3.12: Table object examples in LC(1) (continued).

using a Table object as a hash map

```
01: //create a Table object, ex nihilo.
02: var map = new Table();
03:
04: //add key-value pairs
05: map[\name] = "Mary";
06: map["my_key"] = "my value";
07: map[1] = new Table();
08: map[10.5] = new Exception(\my_exception);
09: var t = new Table();
10: map[t] = "an object reference can be also used as a key.";
11:
12: //then access each value
13: println(map[\name]);
14: println(map["my_key"]);
15: println(map[1]);
16: println(map[10.5].type);
17: println(map[5]);
```

The output from the above example

```
Mary
my value
#<Table:0x7f9123819310>
\my_exception
an object reference can be also used as a key.
```

Figure 3.13: A Table example in LC(2).

The slot access of ‘obj.slotname’ is equivalent to ‘obj[\slotname]’. Figure 3.13 describes a simple example of Table object and index operator.

A set of library functions to manipulate a Table object is also provided in LC. The global variable *\$Table* holds a reference to a Table object, which retains the library functions in its slots. Table 3.6 shows the list of the library functions¹. The main reason these functions are not directly assigned to each instance of a Table object is because one may want to create a slot with the same name for other purposes or even to write their versions of these library functions; for instance, as seen in the next section, a user may want to create a deep copy of a Table object instead of a shallow copy as provided by the ‘clone’ function. In such cases, always attaching such library functions

¹Such a design strategy can be also seen in Lua’s table object which retains the related library functions.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

Table object: the library functions
setDelegate(var self, delegate)
set the delegation from the table <i>self</i> to the parent table <i>delegate</i> .
getDelegate(var self)
returns the delegation table of <i>self</i> , if any. If there is no delegation set up, returns <i>null</i> .
hasSlot(var self, slot)
check if the table <i>self</i> or any tables in its delegation chain has the slot <i>slot</i> (a symbol value). This function returns a boolean value.
hasOwnSlot(var self, slot)
check if the table <i>self</i> has its own slot <i>slot</i> (a symbol value) (not in the delegated tables). This function returns a boolean value.
clone(var self)
returns a shallow copy of the table <i>self</i> .
getSlotNames(var self)
returns an array of the available slot names in the table <i>self</i> .

Table 3.6: The list of library functions for Table objects.

to the Table object's slots by default may cause some problems, especially if they are assigned to slots implicitly (see Figure 3.15 for an example).

Delegation. Instead of the inheritance mechanism in class-based languages, prototype-based languages provide *delegation*. Delegation is a message forwarding mechanism, first introduced in the Act 1 language (184). Dony briefly explains the essence of the delegation mechanism below.

“The basic idea of delegation is to forward message that cannot be handled by an object to another object called its parent in Self or proxy in Act 1” and “the key point of delegation is that the pseudo-variable ‘self’ still points to the original receiver of the message, even if the method used to answer the message is found in one of its parent (185). Delegation is proposed as a mean for an object to retrieve and share knowledge provided by another object” (101)

The advocates of prototype-based programming often argue that delegation is “more powerful than class-inheritance” (31); for instance, Lieberman discusses that he can easily simulate class-inheritance by using delegation, “but not map the reverse” (31)(185).

LC’s Table object provides this delegation mechanism, by using the *setDelegate(self, delegate)* method of *\$Table*. Figure 3.14 describes an example of delegation in LC. A similar mechanism is also used in the previous example of exception handling for the exception hierarchy; each catch block traverses ancestor exceptions to examine if the thrown exception is of the type to be caught by the block.

Cloning. As shown in the previous examples, a Table object can be created ex nihilo (from scratch) by using the ‘new’ operator. As in many prototype-based languages, it is possible to create an object by cloning. LC performs a shallow copy¹ by the ‘clone(var self)’ function.

Figure 3.15 describes an example of object-cloning in LC. As shown on line 15, simply cloning a Table object doesn’t copy the objects in its slots and copies the reference to the same object. A user would need to write their own version of the ‘clone’ method for a deep copy when it is necessary as on lines 23-31.

Duck-typing. *Duck-typing* allows polymorphism “based on what an object can support rather than that object’s inheritance hierarchy” (290, p.39) and is supported by many dynamically-typed languages. For instance, Ruby (206), Lua (150), Python (301), Groovy (166), and JavaScript (127) all support duck-typing. The concept of duck-typing is often explained by the phrase, “if it walks like a duck and quacks like a duck, it must be a duck” (118, p.61). Unlike in class-based languages, in which the inheritance hierarchy decides if an object can respond to a certain message, in dynamic languages that allow duck-typing, an object can respond to any message as long as the object supports the message, regardless of the object type.

LC also allows duck-typing as shown in the Figure 3.16 example. In this example, the two Table objects (‘man’ and ‘dog’) are totally independent from each other, yet both of them have slots with the same name and functions set to the slots

¹a shallow copy “merely copies a single object and the object references within it”, whereas a deep copy “copies every object recursively” (252).

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

a delegation example

```
01: //create a parent table first.
02: var t1 = new Table();
03: //add key-value pairs to the parent
04: t1.hello = function(var self){
05:     println("Hello, world!");
06: };
07:
08: //create a another table and set up the delegation.
09: var t2 = new Table();
10: //below 'Table' is a built-in global object.
11: $Table.setDelegate(t2, t1);
12: //the access to t2.hello(var self) is forwarded to t1.
13: t2->hello()
14:
15: //assigning another function to t1.hello.
16: t1.hello = function(var self){
17:     println("Hello, my friend!");
18: };
19: //calling t2.hello(var self) again.
20: t2->hello();
21:
22: //assigning t2's own slot.
23: t2.hello = function(var self){
24:     println("Hello, my dear!");
25: };
26: t2->hello();
27: //calling t1's hello.
28: t1->hello();
29: //indexed access is also delegated.
30: t1["myslot"] = "delegated(t1)!"
31: println(t2["myslot"]);
32: //assigning t2's own thread.
33: t2["myslot"] = "I am t2.";
34: println(t2["myslot"]);
35: println(t1["myslot"]);
```

The output from the above example

Hello, world!	line 13
Hello, my friend!	line 20
Hello, my dear!	line 26
Hello, my friend!	line 28
delegated(t1)!	line 31
I am t2.	line 34
delegated(t1)!	line 35

Figure 3.14: A delegation example in LC.

an object-cloning example

```
01: //create a Table object first.
02: var t1 = new Table();
03: //set some value to it slot
04: t1.message = "Hello, I am the 'message' slot of t1.";
05: //then, create another object.
06: var t2 = new Table();
07: $Table.setDelegate(t2, t1); //then, set up delegation to t1.
08: println(t2.message); //"Hello, I am the 'message' slot of t1."
09:
10: //clone the object t2.
11: var t3 = $Table.clone(t2);
12: println(t3.message); //"Hello, I am the 'message' slot of t1."
13:
14: //check if the reference to the delegated object is the same.
15: println($Table.getDelegate(t2) == $Table.getDelegate(t3)); //"true"
16: //assigning the reference to 'clone' function to the slot 'clone' of t3.
17: //t3->clone() will be the same as $Table.clone(t3);
18: t3.clone = $Table.clone;
19: var t4 = t3->clone(); //t4 will be also a shallow copy.
20: println(t4.message); //"Hello, I am the 'message' slot of t1."
21:
22: //create a deep copy version of 'clone'.
23: t3.clone = function(var self){
24:     var clone = $Table.clone(t3);
25:     //copy the parent (delegation) table, too.
26:     var delegate = $Table.getDelegate(t3);
27:     delegate = $Table.clone(delegate);
28:     delegate.message = "I am the message slot, but not of t1";
29:     $Table.setDelegate(clone, delegate);
30:     return clone;
31: };
32: var t5 = t3->clone();
33: println(t3.message);
34: println(t5.message);
35:
36: //"changing the value at t1.message. (t1 is a parent table of t3.)"
37: t1.message = "Hello, I am changed.";
38: println(t3.message);
39: println(t5.message);
40: println($Table.getDelegate(t3) == $Table.getDelegate(t5)); //"false"
```

Figure 3.15: An object-cloning example in LC.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

The output from the above example

Hello, I am the 'message' slot of t1.	line 08
Hello, I am the 'message' slot of t1.	line 12
true	line 15
Hello, I am the 'message' slot of t1.	line 20
Hello, I am the 'message' slot of t1.	line 33
I am the 'message' slot, but not of t1	line 34
Hello, I am changed.	line 38
I am the 'message' slot, but not of t1	line 39
false	line 40

Figure 3.15: An object-cloning example in LC (continued).

have the same number of arguments; thus they can respond to the same method call ('object->greet();') as on line 19.

3.2.1.2 Prototype-based programming at the level of sound synthesis

Patch object and unit-generators. While we discuss a different abstraction of digital sound synthesis for microsound synthesis in the later section, LC is still equipped with traditional unit-generators in its sound synthesis framework. LC provides the *Patch* object to support prototype-based programming in sound synthesis by the unit-generators. The reason LC provides two different objects, (*Table* and *Patch*), is that digital sound synthesis is a highly domain-specific feature that requires intimate collaboration with the underlying software framework. Furthermore, providing another object dedicated to sound synthesis makes it easier to provide the features for frequently performed tasks; for instance, cloning a sound-object normally supposes a deep copy as the purpose of cloning is basically to generate another sound object independent from the original sound-object.

Only a unit-generator or another patch object (subpatch) to the slots of a Patch object. A runtime exception is thrown when assigning the other type of values. A unit-generator (and subpatch) can belong to only one patch and cannot be shared between two or more patches. Once it is attached to a patch, it cannot be assigned to any other patch. A Patch object has its own methods attached to its slots by default and the assignment to these slots also causes a runtime exception, whereas the library functions for Table objects are separately retained by **\$Table** global variable; Table 3.7 lists the

a duck-typing example

```

01: //create a Table object.
02: var man = new Table();
03: //set some value to it slots
04: man.message = "This is the 'message' slot of 'man'.";
05: man.greet = function(var self){
06:     println("Hello!");
07: };
08:
09: //create another Table object.
10: var dog = new Table();
11: dog.message = "This is the 'message' slot of 'dog'.";
12: dog.greet = function(var self){
13:     println("Bow wow!");
14: };
15:
16: var test = function(var object){
17:     println("the slot 'message' contains the value:" .. object.message);
18:     //calling the 'greet(var self)'.
19:     object->greet();
20: };
21:
22: //pass each object to 'test' function.
23: test(man);
24: test(dog);

```

The output from the above example

the slot 'message' contains the value:	line 17
slot of 'man'.	
Hello!	line 19
the slot 'message' contains the value:	line 17
slot of 'dog'.	
Bow wow!	line 19

Figure 3.16: A duck-typing example in LC.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

available methods for a Patch object.

While the methods and attributes of unit-generators significantly vary, as each unit-generator has its own unique methods and attributes, LC provides the basic set of the methods and attributes shared by all the unit-generators. Table 3.8 lists these methods and attributes. As described, they are mostly for the collaboration between the unit-generators and microsound synthesis objects/manipulations in LC's sound synthesis framework¹.

Figure 3.17 is a simple Patch object example to play a sine wave oscillator. As shown in this example, the unit-generators are not directly connected to each other. Instead, the Patch object manages all the connections between the slots. This design makes it easier to replace a unit-generator with another one by assignment as on lines 36-42. There is no necessity to remove an existing connection and build a new connection between the unit-generators, as seen in other languages².

LC also provides a useful syntax sugar for creating a patch object, *patch* and *update_patch(patch)* as shown Figure 3.18, which performs the equivalent task as Figure 3.17. The expression, *patch* {...} creates and returns a Patch object. Inside the block following *patch*, *=>* operator and *=|* operator can be used to connect/disconnect the connections between the slots. The unit-generators can also be created by placing its name after a slot name followed by *'.'*. The other expression *update_patch* receives a Patch object inside parentheses and modifies it in the same way as the *patch* expression does. See Figure 3.19 for some more examples.

As shown in these examples, LC's *Patch* object allows a significant degree of dynamism in the creation and modification of sound objects by adopting the concept of prototype-based programming at the sound synthesis level.

Subpatches. LC's *Patch* object can also be used as a subpatch, as shown in Figure 3.20. *Inlet~* and *Outlet~* unit-generators are provided. By using the slot names given to these unit-generators, signals can be routed between a subpatch and its parent patch.

¹The detail of these methods will be described in the later section dedicated to microsound synthesis.

²The detailed discussion on such a problem is described in the chapter, Discussion.

3.2 The Core Language Features

The Patch object's methods	
clone(var self)	returns a deep copy of the object <i>self</i> . All the unit-generators and subpatches in the patch are recursively copied.
start(var self, duration=-1::samp, offset=0::samp)	activates (start playing) the patch <i>self</i> for <i>duration</i> after <i>offset</i> . Giving a negative number for <i>duration</i> makes the patch active until it is explicitly deactivated.
stop(var self)	immediately deactivates (stop playing) the patch.
connect(var self, src, outlet, dst, inlet)	makes a connection between the outlet <i>outlet</i> of the unit-generator (or subpatch) of <i>src</i> to the inlet <i>inlet</i> of the unit-generator (or subpatch) of <i>dst</i> . The modification won't be reflected until <code>compile(var self)</code> is explicitly called.
disconnect(var self, src, outlet, dst, inlet)	disconnect the connection between the outlet <i>outlet</i> of the unit-generator (or subpatch) of <i>src</i> to the inlet <i>inlet</i> of the unit-generator (or subpatch) of <i>dst</i> . The modification won't be reflected until <code>compile(var self)</code> is explicitly called.
write(var self, samples, inlet=\defin, offset=0::samp, ugen_inlet=\defin)	writes a <i>Samples</i> object to the inlet <i>ugen_inlet</i> of the unit-generator/subpatch stored at the slot <i>inlet</i> , with <i>offset</i> to the future. This function is used for the collaboration between the traditional unit-generator-based sound synthesis and LC's microsound synthesis framework
process(var self, dur)	process the patch for <i>dur</i> . This function is used for the collaboration between the traditional unit-generator-based sound synthesis and LC's microsound synthesis framework.
read(var self, dur, outlet=\defout, offset=0::samp, ugen_outlet=\defout)	read the output for the last <i>dur</i> duration from <i>ugen_outlet</i> of the unit-generator/subpatch stored at the slot <i>outlet</i> . This function returns a <i>Samples</i> object and is used for the collaboration between the traditional unit-generator-based sound synthesis and LC's microsound synthesis framework.
pread(var self, dur, outlet=\defout, ugen_outlet=\defout)	process the patch for <i>dur</i> and returns the output from <i>ugen_outlet</i> of the unit-generator/subpatch stored at the slot <i>outlet</i> . This function returns a <i>Samples</i> object and is used for the collaboration between the traditional unit-generator-based sound synthesis and LC's microsound synthesis framework.

Table 3.7: The list of Patch object's methods

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

The unit-generator's methods
clone(var self)
returns a deep copy of the object <i>self</i> . All the unit-generators and subpatches in the patch are recursively copied.
write(var self, samples, inlet=\defin, offset=0::samp)
writes a Samples object to the inlet <i>inlet</i> of the unit-generator/subpatch stored at <i>slot</i> , with <i>offset</i> to the future. This function is used for the collaboration between the traditional unit-generator-based sound synthesis and LC's microsound synthesis framework
process(var self, dur)
process the patch for <i>dur</i> . This function is used for the collaboration between the traditional unit-generator-based sound synthesis and LC's microsound synthesis framework.
read(var self, dur, slot=\defout, offset=0::samp)
read the output for the last <i>dur</i> duration from <i>outlet</i> of the unit-generator/subpatch stored at <i>slot</i> . This function returns a <i>Samples</i> object and is used for the collaboration between the traditional unit-generator-based sound synthesis and LC's microsound synthesis framework.
pread(var self, dur, outlet=\defout)
process the patch for <i>dur</i> and returns the output from the outlet <i>outlet</i> of the unit-generator. This function returns a <i>Samples</i> object and is used for the collaboration between the traditional unit-generator-based sound synthesis and LC's microsound synthesis framework.
retain
'retain' is an attribute (not a method), which returns and sets up the number of the output samples that should be retained. For instance, 'var a = new Sin~(); a.retain = 1::second;' will let the sine wave oscillator retain the last 1 second output samples.

Table 3.8: The list of unit-generator's methods.

A simple sine wave oscillator example

```
01: //create a Patch object.
02: var p = new Patch();
03: //store unit-generator objects to its slots.
04: p.src = new Sin~(440); //u-gen names always start with a tilde.
05: p.dac = new DAC~();
06: //connect the default output of a sine wave osc to DAC's default input.
07: p->connect(\src, \defout, \dac, \defin);
08: //update the unit-generator graph.
09: p->compile();
10: //start playing the patch immediately.
11: p->start();
12:
13: //wait for 1 second and change the frequency.
14: now += 1::second;
15: p.src.freq = 880;
16:
17: //wait for 0.5 second and disconnect sin and dac.
18: now += 0.5::second;
19: p->disconnect(\src, \defout, \dac, \defin);
20: //it is necessary to update the unit-generator graph again.
21: p->compile();
22:
23: //wait for 0.5 second again, connect to DAC's right channel(ch1).
24: now += 0.5::second;
25: p->connect(\src, \defout, \dac, \ch1);
26: p->compile();
27: //wait for 1 second again, deactivate the patch.
28: now += 1::second;
29: p->stop();
30:
31: //wait for 0.5 second, activate the patch again.
32: now += 0.5::second;
33: p->start();
34:
35: //swap a sine wave osc with a phasor.
36: var tmp = p.src; //store a sinewave osc to tmp.
37: p.src = new Phasor~(440);
38: p->compile();
39: //restore a sinewave osc after 1 sec.
40: now += 1::second;
41: p.src = tmp;
42: p->compile();
```

Figure 3.17: A simple sine wave oscillator example in LC.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

Another simple sine wave oscillator example

```
01: //create a Patch object.
02: //patch { ... } expression is a syntax sugar to create a new patch.
03: var p = patch {
04:     //=> operator builds a connection.
05:     //if no inlet/outlet is specified it connects defout to defin.
06:     src:Sin~(440) => dac:DAC~();
07: };
08: p->start();
09: now += 1::second;
10: p.src.freq = 880;
11:
12: now += 0.5::second;
13: //update_patch { ... } expression is a syntax sugar to update a patch.
14: update_patch(p) {
15:     //|= operator removes an existing connection.
16:     //when no inlet/outlet is specified, it disconnects defout and defin.
17:     src =| dac;
18: };
19:
20: update_patch(p){
21:     //=> operator can also specify inlet/outlet explicitly.
22:     src { \out => \ch1 } dac;
23: };
24: now += 1::second;
25: p->stop();
26:
27: now += 0.5::second;
28: p->start();
29:
30: var tmp = p.src;
31: update_patch(p){
32:     src:Phasor~(440);
33: };
34: now += 1::second;
35: update_patch(p){
36:     src:tmp;
37: };
```

Figure 3.18: Another sine wave oscillator example in LC.

a patch example

```
01: //a simple patch.  if no slot names given,
02: //LC automatically generates slotnames such as _0, _1, _2 ...
03: var p1 = patch {
04:   Sin~(440) => DAC~();
05: };
06:
07: //slot names given and used again.
08: //the connection specifies inlet and outlet.
09: var p2 = patch {
10:   src:Sin~(440) { \out => \ch0 } DAC~();
11:   src { \out => \ch1 } dac;
12: };
13:
14: //connecting one outlet to several inlets at once.
15: var p3 = patch {
16:   src:Sin~(440) { \out => [\ch0, \ch1] } DAC~();
17: };
18:
19: //cascading the unit-generators.
20: var p4 = patch {
21:   Sin~(10, amp:1) { \out => \amp } src:Sin~(880) => DAC~();
22: };
```

Figure 3.19: A patch expression example in LC.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

a subpatch example

```
01: //create a patch to be used as a subpatch.
02: var sub1 = patch {
03: //create an inlet and connect its output as a freq input of a sine osc.
04:   defin:Inlet~() {\out => \freq} Sin~() => defout:Outlet~();
05: };
06:
07: //create another patch to be used also as a subpatch.
08: var sub2 = patch {
09: //this a triangle wave version.
10:   defin:Inlet~() {\out => \freq} Triangle~() => defout:Outlet~();
11: };
12:
13: //create one more patch. use 'sub1' as a subpatch
14: var parent = patch {
15:   sig:Sig~(220) => s:sub1 => DAC~();
16: };
17: parent->start();
18:
19: //wait for 1 second and replace the subpatch, then change the freq.
20: update_patch(parent){
21:   s:sub2;
22: };
23: parent.sig.amp = 1760;//Sig~ ugen output the constant signal given by amp.
```

Figure 3.20: A subpatch example in LC.

a patch-cloning example

```

01: //create a patch to be used as a prototype.
02: var p = patch {
03:   //a simple sine wave oscillator.
04:   s:Sin~() => DAC~();
05: };
06:
07: //clone it 8 times to play a phrase.
08: for (var i = 1; i <= 8; i += 1){
09:   //first, clone the prototype.
10:   var note = p->clone();
11:   //'note' is a deep copy.  changing 'freq' doesn't affect other objects.
12:   note.s.freq = i * 440;
13:   note->start(dur:1::second);//play it for 1 second.
14:   now += 0.5::second;//wait 0.5 second before the next note.
15: }

```

Figure 3.21: A patch-cloning example in LC.

Cloning. While *Table* returns a shallow copy when cloned, *Patch* returns a deep copy. All the subpatches and unit-generators in a patch are recursively copied. This makes it easier to use a Patch object as if it were a note-level object in other computer music languages (for instance as in Csound). Figure 3.21 shows an example to clone a Patch object.

Duck-typing. As duck-typing is applicable to any object in LC, patches and unit-generators can also receive the benefits of duck-typing. Figure 3.22 describes a simple example of duck-typing for Patch objects. While *p1* and *p2* are totally independent from each other, since both patches have the slot *s* and both unit generators (*Phasor~* and *Sin~*) have the slots *freq*, the patches can be treated in the same manner in the function *test*. Since the table created between lines 11-28 has the slots required in the function *test*, there occurs no runtime-error in this example.

3.2.2 Mostly-strongly-timed programming and other features with respect to time

In the previous chapter, the problem of precise timing behaviour and the lack of other features with respect to time were described. The recent programming languages such

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

A patch duck-typing example

```
01: //create a patch.
02: var p1 = patch {
03:   s:Phasor~() {\out => \ch0 } DAC~();
04: };
05:
06: //create another patch.;
07: var p2 = patch {
08:   s:Sin~() {\out => \ch1 } DAC~();
09: };
10:
11: //create/initialize tables.
12: var t1 = new Table();
13: var t2 = new Table();
14: t1.s = t2;
15: t1.message = "Hello, world!";
16: t1.start = function(var self, dur){
17:   println(self.message);
18:   println("self.s.freq=" .. self.s.freq);
19:
20:   var stime = now;
21:   var etime = now + dur;
22:   while(now <= etime){
23:     var elapsed = (now - stime) / SampleRate;
24:     println("time elapsed:" .. elapsed);
25:     now += 1::second;
26:   }
27:   println("Bye!");
28: };
29:
30: //a test function for duck-typing.
31: var test = function(var obj, duration=5.0::second)
32: {
33:   obj.s.freq = Rand(1, 10)* 440;
34:   obj->start(dur:duration);
35: };
36:
37: //pass patches and a table as an argument.
38: test(p1); //p1 will start playing.
39: test(p2); //p2 will start playing.
40: test(t1); //t1.start(var self, dur) will be executed.
```

Figure 3.22: A duck-typing example (for Patch object) in LC.

The output from the above example

```
Hello, world!  
self.s.freq=1320  
time elapsed:0  
time elapsed:1  
time elapsed:2  
time elapsed:3  
time elapsed:4  
time elapsed:5  
Bye!
```

Figure 3.22: A duck-typing example (for Patch object) in LC (continued).

as LuaAV (307) and ChucK (312) provide sample-rate accurate behaviour by explicitly controlling the progress of the internal logical time, and compositional algorithms can be easily synchronized with real-time sound synthesis. However such software design still leaves a significant problem in timing behaviour.

As logical time cannot be progressed without explicit control in these languages, time-consuming tasks can easily invalidate the underlying assumption that tasks will advance logical synchronous time before the deadline to provide its output samples for real-time sound synthesis; missing the deadline causes temporal suspension in real-time DSP. As tasks must be always performed synchronously with logical time, these languages do not allow describing such time-consuming tasks as preemptive background threads.

However, such time-consuming tasks can be frequently seen in computer music. For instance, it can consume a considerable amount of time to load a sound file from disk and then analyse the sound data to extract wavesets. While a user may consider inserting the explicit advance of logical time (or explicit task-switching) inside the time-consuming part of a task so that the task can be performed without suspending real-time DSP, the resulting program can be unnecessarily complicated. It should be also noted that such a strategy cannot be applied to a certain kind of tasks that can be suspended by I/O block; for instance, DISK I/O can block the execution of a thread, yet it is very unpredictable within a computer program as to when the disk I/O can be finished.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

Furthermore, even the most recent computer music languages lack some desirable features with respect to time, with which general purpose real-time programming languages are normally equipped. While the designers of Impromptu considered such features in its language design (276), Impromptu’s software design regarding execution time constraints has a significant flaw in expressing the nested constraints. Moreover, due to Impromptu’s software design, it does not guarantee precise timing behaviour and a user often has to take some special care when dealing with time.

To overcome such problems as above, LC proposes and implements the concept of mostly-strongly-timed programming, also integrating other features with respect to time together.

3.2.2.1 Mostly-strongly-timed programming

As described in the previous section, the strongly-timed programming concept is applied to LC’s language design. Yet, as already mentioned, time-consuming tasks can cause a temporary suspension of real-time DSP under such a language design. LC solves this problem by extending the strongly-timed programming concept, introducing the explicit switch in runtime between synchronous/non-preemptive context and asynchronous/preemptive context.

There are two keywords used in mostly-strongly-timed programming for context switching. After the keyword *async*, the virtual machine executes its following block in the asynchronous context, in which a thread can be preempted without the explicit advance of time; instead, the explicit control of the advance of logical time is lost until the ‘*async*’ block is over, as the advance of logical time occurs regardless of the progress of the tasks. However, if a thread advances the logical time explicitly, for instance by such an expression as ‘*now* += 1::second’, which advances the logical time by *duration*, the thread resumes with sample-rate accurate timing precision, even in the asynchronous context. Yet, when wake-up timing is given by a specific *time* (e.g., *now* = SomeSpecificWakeUpTime) an exception can be thrown if the logical time is already advanced beyond the given time. If not, the thread can be resumed with sample-rate accuracy in timing.

On the other hand, after the keyword *sync*, the virtual machine switches the thread to the synchronous context, in which any thread cannot be suspended without the

explicit advance of logical time (as in strongly-timed programming). These *sync* and *async* blocks can be nested.

Thus, mostly-strongly-timed programming resolves the problem of temporary suspension of real-time DSP caused by time-consuming tasks in strongly-timed programming, while maintaining its sample-rate accurate timing precision. Mostly-strongly-timed programming also can be used to realize a background task, as a thread in asynchronous/preemptive context can process its task regardless of the advance of logical time. Figure 3.23 briefly describes an example of mostly-strongly-timed programming in LC.

3.2.2.2 Timed-tagged message communication

In addition to mostly-strongly-timed programming, LC also provides other desirable features with respect to time. As LC's timing behaviour is based on its logical synchronous time, all these features can be performed with sample-rate accurate timing precision.

Concurrency: synchronization and communication. As a computer program often requires multi-tasking, programming languages also need to provide a means for the communication and synchronization between tasks. The communication mechanisms are generally based into two programming models, shared-memory and message-passing. Burns et al. briefly describes these models as below.

“In the shared-memory approach some or all of a program's variables are accessible to multiple threads. For a pair of threads to communicate, one of them writes a value to a variable and the other simply reads it. In a message-passing programming model, threads have no common state. For a pair of threads to communicate, one of them must perform an explicit send operation to transmit data to another” (66, p.671).

“Synchronization refers to any mechanism that allows the programmer to control the relative order in which operations occur in different threads. Synchronization is generally implicit in message-passing models: a message must be sent before it can be received. If a thread attempts to receive a message that has not yet been sent, it will

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

A mostly-strongly-timed programming example

```
01: //create/play a sine wave oscillator patch to hear the suspension of real-time DSP
02: var p = patch {
03:   s:Sin~(440) => DAC~();
04: };
05: p->start();
06:
07: //do some time-consuming tasks. the code below will be blocked for file I/O
08: //and can causes a temporary suspension of real-time DSP, which is audible;
09: //one may hear the sine wave oscillator stops its sound output for a while.
10: for (var i = 0; i < 16; i += 1){
11:   //load sample0.wav - sample15.wav to the buffers no. 0-15
12:   LoadSndFile(i, "/sound/sample" .. i .. ".wav");
13: }
14:
15: //the below infinite loop suspends the DSP forever, since there is no explicit
16: //advance of the logical time within, while the thread is in the 'sync' context.
17: /*
18: while(true){
19: }
20: */
21: //-----
22: // mostly-strongly-timed programming
23: //-----
24: //an array with 16 elements.
25: var wsarray = new Array(16);
26:
27: //'async' block switches to the preemptive/asynchronous context. The current
28: //thread can be preempted even when there is no explicit advance of logical time.
29: async {
30:   //the below doesn't suspend the real-time DSP.
31:   for ( var i = 0; i < 16; i += 1){
32:     //load sample0.wav - sample15.wav to the buffers no. 0-15
33:     LoadSndFile(i, "/sound/sample" .. i .. ".wav");
34:   }
35:
36:   //'sync' block switches to the non-preemptive/synchronous context.
37:   sync{
38:     for (var i = 0; i < 10; i += 1){
39:       p.s.freq = Rand(1, 10) * 220;
40:       now += 1::second;
41:     }
42:     //switching to the async context again.
43:     async {
```

Figure 3.23: A mostly-strongly-programming example in LC.


```

44:      //analyse/extract wavesets from the sound buffers.
45:      //(the extraction can consume lots of time).
46:      for (var i = 0; i < 16; i +=1){
47:          wsarray[i] = ExtractWavesets(i);
48:      }
49:  }//the end of async block(lines 43-49); switching back to the sync context.
50: }//the end of sync block(lines 37-51); switching back to the async context.
51:
52: //this infinite loop don't suspend DSP, since this is the async context and
53: //the underlying scheduler can preempt the thread without
54: //the explicit advance of logical time.
55: while(true){
56: }
57: }//the end of async block (lines 29-57); switching back to async context.

```

Figure 3.23: A mostly-strongly-programming example in LC (continued).

wait for the sender to catch up. Synchronization is generally not implicit in shared-memory models: unless we do something special, a ‘receiving’ thread could read the ‘old’ value of a variable, before it has been written by the sender” (66, p.671).

Besides these two models, *tuple space*, which originates in the Linda language by Gelernter and his colleagues (124)(125), may be particularly notable. Impromptu (276) provides this model as its language feature. While the tuple space model is also considered as a kind of distributed shared space memory model (69, pp.177-185), yet, unlike the shared-memory model described above, it involves implicit synchronization. Matsuoaka and Kawai briefly explain this concept as below.

In the Tuple¹ Space, “the list of formal and actual arguments given in the send request forms a Tuple. The sender process inserts the Tuple into the Tuple Space. Each Tuple is a unique, independent existence in the Tuple Space. The receiver process gives its own list of arguments in its Tuple withdrawal request. Withdrawal occurs when there is a Tuple matching the receiver’s specific request; otherwise, the receiver process waits until such a Tuple becomes available in the Tuple Space. The receiver obtains

¹A tuple is a sequence of elements. For example, (1, 5, 10) denotes a 3-tuple and (“foo”, “bar”, a_object, yet_another_object, 1123) denotes a 5-tuple.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICROSOUND SYNTHESIS

the necessary information from elements of the Tuple where the formal argument of the receiver matched the actual element of the Tuple.” (207)

As Burns et al. also discuss in (66, p.660), the above programming models for communication and synchronization can be realized both as a feature in programming language design or as a library package. The recent computer music languages seem more reliant on the shared-memory programming model. For instance, ChuckK provides the native *Event* class, which can be considered a variation of monitor-based synchronization¹, and the communication between threads is mainly performed by the shared-memory model. SuperCollider also provides the *Condition* object for monitor-based synchronization. The communication between threads is performed by the shared-memory model. A notable example of the message-passing model in computer music language design is RPC (Remote Procedure Call) in Impromptu (276). In addition to the RPC mechanism, Impromptu also provides *tuple space*.

Time-tagged message communication in LC. LC adopts the message-passing model and provides the feature in the forms of library functions and language specification². Each message can be tagged with its delivery timing and LC’s strongly-timed programming feature also guarantees messages can be received at the specified time with sample-rate accuracy; if the receiver thread is not waiting for a message at the specified time, the message will remain in the message queue of the receiver thread and will be received immediately when the receiver thread starts receiving.

Figure 3.24 is an example of time-tagged message communication in LC, in which Thread’s *recv* method and *queueMessage* method are used. The *queueMessage* can specify delivery timing when the message can be readable by the receiver thread. Both

¹Synchronization can be categorized into *busy-wait synchronization* and *scheduler-based synchronization*. The former (busy-wait synchronization) loops until a thread acquires the requesting lock as seen in the *spin lock* mechanism. The later blocks the thread that is currently running and then switches to a different thread, as seen in *semaphores* and *monitors* (66, Chapter 12.3: Shared Memory). Java’s synchronization mechanism, which provides *notify()/notifyAll()/wait()* methods, is an example of monitor-based synchronization.

²As the prototype of LC focuses on the adoption of the message-passing model into a computer music language and does not particularly offer some mechanism for mutual exclusion. However, the current version of LC does not require a mutual exclusion mechanism since in the *sync* context, only one thread can be activated and it is non preemptive. Thus, the part of the code which requires resource locking can be enclosed in the *sync* context for mutual exclusion. We would like to leave the issue of whether LC should adopts mutual exclusion as a topic of further investigation, as the issue is also related to parallelisation; the current prototype of LC is concurrent, but not parallel.

time value (the delivery time) as on lines 29-30 and *duration* value (offset from now) can be given for the timing. The *mode* parameter (`\blocking` or `\nonblocking`) can be also specified. In the non-blocking mode, *recv* immediately returns the symbol value, `\nomessage`, if there is no message available. In the case `\nomessage` need to be sent from the sender thread, a user can use Thread's *isMessageReceived* method to make sure if `\nomessage` is returned because there was no message available.

Alternatively, LC also provides the *receive* statement and `<-` operator for time-tagged message communication, as shown in Figure 3.25. The `<-` operator can be used in the same manner as the *queueMessage* method, as seen between lines 62-74¹. As shown between line 69-70, the `<-` operator can specify the message delivery timing by '@' followed by a *time* value or a *duration* value. When '@' is not given, it is delivered immediately to the receiver's message queue.

The *receive* statement is used between lines 06-44. As on line 06, the receive statement is followed by a variable declaration within the parentheses. The receive statement blocks until when a message is received and then the received message is first stored into this variable (in the Figure 3.25 example, it is stored to 'msg'). Then, each *case* and *cond* will be examined one-by-one from the beginning if the received message matches the given conditions. When *case* is examined, it simply applies the '==' operator between the received message and its following value, while *cond* evaluates the following boolean condition. If the evaluation result is true, the following compound statement (`{ ... }`) is executed. When there is no matching *case/cond*, the code jumps to the *default* label (as on line 53) if any. When there is neither matching *cond/case* nor *default*, the message will be temporarily neglected and the *receive* statement will start waiting for the next message. These neglected messages are pushed to the 'unmatched' queue and when the code reaches the next *receive* statement, all the unmatched messages will be returned to the thread's message queue in the same order as when they were first received (as seen between lines 47-56).

¹As the `< -` operator returns the receiver thread object and is left-associative, the operator can be cascaded as shown in Figure 3.25.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

A timed interthread messaging example in LC(1)

```
01:  //create a function object.
02:  var f = function(){
03:      var thread = GetCurrentThread();
04:      while(true){
05:          //receive a message in the blocking mode.
06:          var message = thread->recv(\blocking);
07:          if (message == \quit){
08:              break;
09:          }
10:          println("received :" .. message);
11:      }
12:
13:      println("bye!");
14:      return;
15:  };
16:
17:  //starting the function in a new thread
18:  var thread = f@();
19:  thread->start();
20:
21:  //queue a message to the thread's mailbox
22:  thread->queueMessage("Hello!");
23:  now += 1::second;
24:
25:  //since the return value of queueMessage() is the thread itself,
26:  //it can be cascaded as below.
27:  //it is also possible to specify the message delivery timing.
28:  thread->queueMessage("Sending the first message")
29:      ->queueMessage("This should be received after 1 sec", now + 1::second)
30:      ->queueMessage("This should be received after 2 sec", now + 2::second);
31:
32:  if duration value is used as timing, it is interpreted as 'now + duration'.
33:  thread->queueMessage(\quit, 3::second);
```

The output from the above example

```
received :Hello!
received :Sending the first message
received :This should be received after 1 sec
received :This should be received after 2 sec
bye!
```

Figure 3.24: A timed interthread messaging example in LC(1).

A timed interthread messaging example in LC(2)

```

01: //create a function object.
02: var f = function(){
03:     var thread = GetCurrentThread();
04:     while(true){
05:         //receive statement . the received message will be set to 'msg'.
06:         receive(var msg){
07:             //if msg == \quit
08:             case \quit:{
09:                 //exit this while loop.
10:                 break;
11:             }
12:
13:             //'case' can be juxtaposed.
14:             case \salut:
15:             case \hello:
16:             case \greet:{
17:                 println("hello (received:" .. msg .. ")");
18:             }
19:
20:
21:             //'cond' takes a boolean expression.
22:             cond msg instanceof \integer || msg instanceof \symbol:{
23:                 println("integer or symbol:" .. msg);
24:             }
25:             cond msg instanceof \float && msg >= 0:{
26:                 println("float (msg >= 0) :" .. msg);
27:             }
28:             cond msg instanceof \float && msg < 0:{
29:                 println("float (msg < 0) :" .. msg);
30:             }
31:             cond msg instanceof \Array:{
32:                 print("received an array [");
33:                 for (var i = 0; i < msg.size; i += 1){
34:                     print(msg[i]);
35:                     if (i == msg.size - 1){
36:                         println("]");
37:                     }
38:                     else {
39:                         print(", ");
40:                     }
41:                 }
42:             } //the end of 'for'.
43:         } //the end of the 'cond'.
44:     } //the end of the above receive statement.

```

Figure 3.25: A timed interthread messaging example in LC(2).

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```

45:   println("");
46:   println("received '\\quit'.");
47:   println("try receiving two more messages in the message queue.");
48:   for (var i = 0; i < 2; i+= 1){
49:     receive(var msg){
50:       cond msg instanceof \String:{
51:         println("string : " .. msg );
52:       }
53:       default:{
54:         println("default(received : " .. msg .. ")");
55:       }
56:     }
57:   }
58:   println("bye!");
59: };
60:
61: var receiver = f@();
62: receiver->start();
63: //send messages
64: receiver <- \salut
65:     <- @now + 1::second, \hello;
66: //send some more
67: receiver <- @5::second, \greet
68:     <- @2.5::second, 100
69:     <- 0.5
70:     <- @1.5::second, -100.0
71:     <- @now + 3::second, \some_other_symbol;
72: receiver <- @2::second , [1, \foo, 3, \bar];
73: receiver <- @4::second , "a string"
74:     <- @4::second , \quit
75:     <- @4::second , "yet-another string";

```

The output from the above example

```

hello (received:\salut)
float (msg >= 0) :0.500000
hello (received:\hello)
float (msg < 0) :-100.000000
received an array [1, \foo, 3, \bar]
integer or symbol :100
integer or symbol :\some_other_symbol

received '\\quit'.
try receiving two more messages in the message
queue.
string :  a string
string :  yet-another string
bye!

```

line 17	(sent on line 64)
line 26	(sent on line 69)
line 17	(sent on line 65)
line 29	(sent on line 70)
lines 32-42	(sent on line 72)
line 23	(sent on line 68)
line 23	(sent on line 71)
line 51	(sent on line 73)
line 51	(sent on line 75)
line 57	

Figure 3.25: A timed interthread messaging example in LC(2) (continued).

3.2.2.3 Timing constraints

Start-time constraint and execution-time constraint. One of the most important features, which is seen in many real-time programming languages, is to give timing constraints to a task (e.g., start-time constraints and execution-time constraints). Start-time constraint “provides the earliest deadline by which a function must begin execution” (or in other words, when the task can be started), and execution-time constraint “expresses the maximum time available for the execution” (or time-out) (276).

The feature of start-time constraints is relatively easy to implement; the underlying scheduler of a computer music language can simply schedule a task to be started at the scheduled time or a thread can sleep right after its entrance until the given start-time. However the constraint on execution-time requires more effort to implement. Unlike the exception-handling mechanism, in which an exception occurs at a certain point of the code, time-out can occur at any point in the code when the given execution-time constraint was violated; thus, timing constraints involves *asynchronous transfer of control* (ATC), which “is a transfer of control within a thread, triggered not by the thread itself but from some action by another thread or an event handler” (60).

Some programming languages provide the syntaxes and semantics for timing constraints and others do not (66). For instance, Ada (285) provides the mechanism for timeout as a part of its language specification while real-time Java handles it within the framework of exception handling by adding a special exception called *AsynchronouslyInterruptedException*. As for computer music programming languages, while many recent computer music languages still lack this feature of timing constraints, Impromptu is designed with such features provided by its scheduler (274)(276). Yet, as discussed in the next chapter, Impromptu has a flaw in its software design in execution-time constraints and cannot deal well with nested timing constraints.

Start-time constraint in LC. LC provides the start-time constraint feature for two objects: Thread and Patch, as both of these objects involve concurrency¹. Since LC is a mostly-strongly-timed language, this feature is performed with sample-rate accuracy.

Figure 3.26 and Figure 3.27 describe the start-time constraint examples in LC. As shown on Figure 3.26 (lines 07-09) and Figure 3.27 (lines 12-14), by giving an argument

¹Patch objects must generate sound output concurrently when activated.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

A thread start-time constraint example in LC

```
01: var f = function(var msg){
02:   print(msg);
03:   println(now);
04: };
05:
06: //starting three threads, each with a different start-time constraint.
07: f@("the 1st thread with 1.0 sec offset :")->start(offset:1::second);
08: f@("the 2nd thread with 1.5 sec offset :")->start(offset:1.5::second);
09: f@("the 3rd thread with 2.0 sec offset :")->start(offset:2.0::second);
10: print("now :");
11: println(now);
```

The output from the above example

```
now :#<time:192.261224sec(8478720::samp)>
the 1st thread with 1.0 sec offset :#<time:193.261224sec(8522820::samp)>
the 2nd thread with 1.5 sec offset :#<time:193.761224sec(8544870::samp)>
the 3rd thread with 2.0 sec offset :#<time:194.261224sec(8566920::samp)>
```

Figure 3.26: A thread start-time constraint example in LC.

offset (duration from now) to *start* method, Thread and Patch can be activated at the specified time.

Execution-time constraint in LC. LC is also equipped with the execution-time constraint feature as a part of its language design. As LC is a mostly-strongly-timed programming language as described earlier, this timing constraint is also performed in logical synchronous time with sample-rate accuracy.

The *within-timeout* statement provides the execution time constraint feature. Basically speaking, the *within* block will be given an execution-time constraint and if the given deadline is reached while executing the block, the code will jump to the *timeout* block immediately when the execution-time constraint is violated. If the *within* block is executed within the given time constraints, *timeout* block will be simply skipped after the *within* block is over.

In Figure 3.28, the *within* block between lines 07-13 is given the execution time constraint of five seconds as on line 05. While the block contains an infinite loop (lines 09-12), in which the frequency of a sine wave oscillator of the patch created between

A patch start-time constraint example in LC(1)

```

01: var p1 = patch {
02:   src:Noise~() => DAC~();
03: };
04:
05: var p2 = patch {
06:   src:Sin~(440) {\defout => \ch1} DAC~();
07: };
08:
09: var p3 = patch{
10:   src:Phasor~(440) => DAC~();
11: };
12: p1->start(offset:1::second);
13: p2->start(offset:2::second);
14: p3->start(offset:0.25::second);

```

Figure 3.27: A patch start-time constraint example in LC.

lines 01-04, the code will timeout after five seconds and then jump to the timeout block on line 14, because of the execution constraint. It is possible to write a *within* block without a matching *timeout* block as shown between lines 18-24, in a case when a user does not need to handle time-out.

One of the interesting features of LC’s execution-time constraint is that an execution-time constraint with zero time can be given, as in the synchronous context, LC’s logical time will not be advanced without explicit instruction by a user program. As the *within* statement between lines 27-33 is in the ‘sync’ context¹ and there is no explicit advance of logical time, the ‘for’ loop inside the *within* block can be processed within zero sample time. Thus, the following *timeout* block will never be reached in this example.

While some programming languages such as real-time Java or Ruby handle the execution-time constraint feature within the framework of exception-handling as seen in real-time Java, LC provides the dedicated syntax and semantics, separated from the exception-handling mechanism. The main reason is that the behaviours of execution-time constraints and exception-handling significantly differ when nested.

Generally speaking, in the exception handling mechanism, when an exception is thrown in the inner try-catch block, the system examines which catch block can handle

¹In the current version of LC prototype, the default context is ‘sync’.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICROSOUND SYNTHESIS

the exception from the most inner catch block towards the outer blocks. On the other hand, when the execution-time constraints are nested, the outer execution-time constraint may impose shorter duration than the inner constraint. In such a case, when the deadline of the outer constraint is reached, the thread of execution must jump directly to the handler of the outer constraint¹. Figure 3.29 describes an example of nested execution-time constraints in LC. As shown, exactly when the deadline of the outer execution-time constraint is reached, the control immediately jumps to the timeout block of the outer execution-time constraint, as normally expected for the nested execution-time constraints.

Furthermore, as LC is mostly-strongly-timed, the *within-timeout* statement can be effectively utilized for execution-time constraints in the asynchronous/preemptive context. In the Figure 3.30 example, if the task (between lines 06-11) to load the sound files and to extract wavesets fails to be completed before the deadline, the code replaces the wavesets with the previously used ones (line 15); thus, in LC, a time-consuming task can run in the background and a user can describe the handler when the background task failed to be finished before the given deadline. Such a feature would be desirable to dynamically record and process the instrumental/digitally-generated sounds in live computer music performances.

Time constrained communication in LC can also be achieved by the combination of an execution time constraint and inter-thread message communication described in the previous section; the deadline to receive the response from the other thread can be given as an execution time constraint as in Figure 3.31.

3.2.3 The Integration of objects and manipulations for microsound synthesis

In Chapter 2.3, the difficulty of microsound synthesis programming in the existing computer music languages was discussed and the problem were assessed as a case of the abstraction inversion software anti-pattern².

¹Such a behaviour of nested execution-time constraints differs from one of exception handling. Thus, in real-time Java, which handles execution-time constraint within the exception-handling mechanism must introduce a special behaviour for execution-time constraints, which differs from the other normal exceptions.

²The problem of cognitive misfits between the conceptualization of microsound synthesis techniques and language design was also briefly discussed in the same chapter.

A timeout example in LC(1)

```

01: var p = patch{
02:   sin:Sin~(440) => DAC~();
03: };
04: p->start();
05:
06: //timeout after 5 sec
07: within(5::second){
08:   //an infinite loop, which changes the sine wave frequency every 0.5 sec.
09:   while(true){
10:     p.sin.freq = Rand(1,10)* 440;
11:     now += 0.5::second;
12:   }
13: }
14: timeout{
15:   println("timeout!");
16: }
17:
18: //no timeout block
19: within(Rand(5,8)::second){
20:   while(true){
21:     p.sin.freq = Rand(1,10)* 440;
22:     now += Rand(2,4) * 0.125::second;
23:   }
24: }
25:
26: //zero sec timeout is possible as LC is strongly-timed.
27: within(0::sample){
28:   //print out 100 asterisks. this can be performed in 0 sample, as the thread is
29:   //now in the 'sync' context and there is no explicit advance of logical time.
30:   for(var i = 0; i < 100; i += 1){
31:     println("*");
32:   }
33: }
34: timeout{
35:   println("the code won't reach here.");
36: }

```

Figure 3.28: A timeout example in LC(1).

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

A timeout example in LC(2)

```
01:  var start = now;
02:  within(1::second){
03:    //advance the time 0.25 sec.  still before the deadline.
04:    now += 0.25::second;
05:    within(10::second){
06:      //advance the time 5 sec.  the outer deadline is violated.
07:      now += 5::second;
08:    }
09:    timeout{
10:      var elapsed = now - start;
11:      println("timeout!!  the inner timeout block!!");
12:      println(elapsed);
13:    }
14:  }
15:  timeout{
16:    var elapsed = now - start;
17:    println("timeout!!  the outer timeout block!!");
18:    println(elapsed);
19:  }
```

The output from the above example

```
timeout!!  the outer timeout block!!
#<dur:44100::samp>
```

Figure 3.29: A timeout example in LC(2).

A timeout example in LC(3)

```
01:  async {
02:      var tmp = wsarray;
03:      wsarray = new Array(16);
04:
05:      //try loading 16 new sound files and extract wavesets within 0.5 second.
06:      within(0.5::second){
07:          //try loading 16 sound files.
08:          for (var i = 0; i < 16; i +=1 ){
09:              LoadSndFile(i, "/sound/sample" .. i .. ".aif");
10:              wsarray[i] = ExtractWavesets(0);
11:          }
12:      }
13:      timeout{
14:          //in case we failed to meet above before the deadline, use the old one.
15:          wsarray = tmp;
16:      }
17: }
```

Figure 3.30: A timeout example in LC(3).

A timeout example in LC(4)

```
01:  //sending a message to 'receiver' thread.
02:  suppose the receiver thread is already created.
03:  receiver <- "hello";
04:  //the deadline for the response is 5 second later.
05:  within(5::second){
06:      receive(var msg){
07:          default:
08:              println("received : " .. msg);
09:      }
10:      println("done.");
11:  }
12:  //when the response didn't arrive before the deadline.
13:  timeout {
14:      println("missed the deadline for the response.");
15:  }
```

Figure 3.31: A timeout example in LC(4).

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

A *Samples* object creation example

```
01: //creating a new Samples object from the buffer no. 0.
02: //reading 256 samples, changing pitch one octave higher
03: //from 1 sec after the begging of the buffer.
04: var snd = ReadBuf(bufno:0, dur:256::samp, rate:2.0, offset:1::second);
05: //creating a new Samples object by generating a window by a library function.
06: var win = GenWindow(521::samp, \hanning);
07: //creating a new Samples object by calling the Samples' methods.
08: var grain = snd->applyEnv(win);
09: var halfAmp = snd->amplify(0.5);
10: var octUp = snd->resample(snd.size / 2);
11: var reversed = snd->reverse();
```

Figure 3.32: A *Samples* object creation example.

From such a perspective, it can be assumed that the difficulty in microsound synthesis programming can be reduced by designing a sound synthesis framework that does not cause abstraction inversion; LC integrates the direct counterpart objects for microsounds in its abstraction, together with the related methods and functions. LC also still provides the traditional unit-generators and the means for the collaboration between the unit-generators and microsound synthesis objects.

3.2.3.1 Objects and manipulations for microsound synthesis

Samples object. In LC's sound synthesis framework, each microsound is abstracted as a *Samples* object. A *Samples* object is immutable and contains an arbitrary number of samples within (as much as the operating system allows to allocate the memory for samples).

A *Samples* object can be created in many different ways. Figure 3.32 describes an example of the various ways to create *Samples* objects. As *Samples* is immutable, a new object is returned by these methods; the original object will not be changed at all. Table 3.9 describes the methods and attributes of *Samples*. Each sample within a *Samples* object can be directly accessed by index as shown in Figure 3.33.

SampleBuffer object. As *Samples* is an immutable object, LC provides *SampleBuffer* for tasks that involve the modification of the sample values within a microsound. *Samples* and *SampleBuffer* objects can be mutually convertible to the other type by

3.2 The Core Language Features

Samples object	
size	
the <i>size</i> slot retains the number of samples within the Samples object (read-only).	
dur	
the <i>dur</i> slot retains the duration in samples of the Samples object (read-only).	
resample(var self, size)	
returns a new Samples object by resampling <i>self</i> to <i>size</i> samples	
reverse(var self)	
returns a new Samples object created by reversing the waveforms of <i>self</i>	
amplify(var self, amp)	
returns a new Samples object created by amplifying <i>self</i> by <i>amp</i> .	
normalize(var self)	
returns a new Samples object created by normalizing <i>self</i> .	
add(var self, samples)	
returns a new Samples object created by adding <i>samples</i> to <i>self</i> .	
sub(var self, samples)	
returns a new Samples object created by subtracting <i>samples</i> from <i>self</i> .	
mul(var self, samples)	
returns a new Samples object created by multiplying <i>self</i> by <i>samples</i> .	
div(var self, samples)	
returns a new Samples object created by dividing <i>self</i> by <i>samples</i> .	
accumulatePhase(var self, phase)	
accumulates the phase information. The <i>phase</i> is added to <i>self</i> , but will be wrapped between $-\pi$ to π .	

Table 3.9: Samples object.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

Samples object (continued)	
invertWS(var self)	
returns a new Samples object created by performing waveset inversion to <i>self</i> .	
applyEnv(var self, env)	
returns a new Samples object created by applying the envelope <i>env</i> to <i>self</i> .	
maxAmp(var self)	
returns the maximum amplitude of <i>self</i> .	
fir(var self, coefs)	
returns a new Samples object created by applying FIR filter below with <i>coefs</i> (an array of coefficients, [a0, a1, a2, ... aN]) to <i>self</i> . $y(n) = a_0 * x(n) + a_1 * x(n - 1) + a_2 * x(n - 2) \dots a_N * x(n - N)$	
iir(var self, coefsA, coefsB)	
returns a new Samples object created by applying IIR filter below with <i>coefsA</i> (an array of coefficients, [a0, a1, a2, ... aN]) and <i>coefsB</i> (an array of coefficients, [b1, b2, ... bN]) to <i>self</i> . $y(n) = a_0 * x(n) + a_1 * x(n - 1) + a_2 * x(n - 2) \dots a_N * x(n - N) - b_1 * y(n - 1) - b_2 * y(n - 2) - b_3 * y(n - 3) - \dots - b_N * y(n - N)$	
copyOfRange(var self, start=0, end=-1)	
returns a new Sample object, copying the samples within <i>self</i> . The <i>start</i> and <i>end</i> parameters specify the range to be copied. When giving a negative value for <i>end</i> , it is interpreted as <i>end</i> = size - 1.	
getWaveset(var self, start=0)	
extract a waveset within the Samples object <i>self</i> from the given index <i>start</i> . The return value is a new Sample object which is the first waveset found. If not found, the return value will be 'null'.	
getZeroCrossingIndex(var self, start=0)	
searches the first zero-crossing point within the Samples object <i>self</i> from the given index <i>start</i> . The return value is the index of the first zero-crossing. If no zero-crossing is found, the return value will -1.	
toSampleBuffer(var self, start=0, end=-1)	
returns a new SampleBuffer object, copying the samples within <i>self</i> . The <i>start</i> and <i>end</i> parameters specify the range to be copied. When giving a negative value for <i>end</i> , it is interpreted as <i>end</i> = size - 1.	

Table 3.9: Samples object (continued).

An indexed-access to *Samples* object example

```
01: var win = GenWindow(128::samp, \triangle);
02:
03: //iterate each samples within 'win'.
04: for(var i = 0; i < win.size; i += 1){
05:   println("win[" .. i .. "] = " .. win[i]);
06: }
```

The output from the above example

```
win[0] = 0.000000
win[1] = 0.015625
win[2] = 0.031250
win[3] = 0.046875
win[4] = 0.062500
win[5] = 0.078125
win[6] = 0.093750

...

win[126] = 0.031250
win[127] = 0.015625
```

Figure 3.33: An indexed-access to *Samples* object example.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

SampleBuffer object
size
the <i>size</i> slot retains the number of samples within the SampleBuffer object (read-only).
dur
the <i>dur</i> slot retains the duration in samples of the Samples object (read-only).
toSamples(var self, start=0, end=-1)
returns a new Samples object, copying the samples within <i>self</i> . The <i>start</i> and <i>end</i> parameters specify the range to be copied. When giving a negative value for <i>end</i> , it is interpreted as <i>end</i> = size - 1.

Table 3.10: SampleBuffer object.

calling *toSampleBuffer* or *toSamples* method. Table 3.10 describes the methods and attributes of SampleBuffer. *SampleBuffer* allows indexed-access to the samples both for read and write, as shown in Figure 3.34.

Library functions to manipulate microsounds. As described, a Samples object can be used to represent a microsound, and its methods provide the related manipulations for microsound synthesis, such as enveloping, addition, amplification, etc. Some manipulations are better provided as library functions not as Sample’s methods, if they are not directly associated with an existing Samples object; for instance, a function to create a new Samples object from a sound buffer is not associated to any particular Samples object that is already created. Table 3.11 lists the library functions to create/manipulate Samples in the current prototype of LC.

3.2.3.2 Microsound synthesis in LC

This section provides the code examples for microsound synthesis techniques in LC. While there are lots of synthesis techniques that belong to the family of microsound synthesis, those most widely-used in computer music compositions are described to provide the enough information for the following discussion in the later chapter. The examples include granular synthesis (synchronous, quasi-synchronous and asynchronous), granular sampling, and waveshape synthesis. Time-stretching and cross-synthesis by FFT/IFFT

A SampleBuffer object example

```
01: //create a SampleBuffer object (8 samples).
02: var sbuf1 = new SampleBuffer(8);
03: //generate a square waveform.
04: for(var i = 0; i < sbuf1.size; i += 1){
05:     if (i < sbuf1.size / 2){
06:         sbuf1[i] = 1.0;
07:     }
08:     else {
09:         sbuf1[i] = -1.0;
10:     }
11: }
12: //create a Samples object
13: var smp1 = sbuf1->toSamples();
14: println("a new Samples object");
15: for(var i = 0; i < smp1.size; i += 1){
16:     println("smp1[" .. i .. "] = " .. smp1[i]);
17: }
18: //amplify 'smp1' and convert it back to a SampleBuffer object
19: var tmp = smp1->amplify(2.0);
20: var sbuf2 = tmp->toSampleBuffer();
21: println("a new SampleBuffer object");
22: for(var i = 0; i < sbuf2.size; i += 1){
23:     println("sbuf2[" .. i .. "] = " .. sbuf2[i]);
24: }
```

The output from the above example

```
a new Samples object
smp1[0] = 1.000000
smp1[1] = 1.000000
smp1[2] = 1.000000
smp1[3] = 1.000000
smp1[4] = -1.000000
smp1[5] = -1.000000
smp1[6] = -1.000000
smp1[7] = -1.000000
a new SampleBuffer object
sbuf2[0] = 2.000000
sbuf2[1] = 2.000000
sbuf2[2] = 2.000000
sbuf2[3] = 2.000000
sbuf2[4] = -2.000000
sbuf2[5] = -2.000000
sbuf2[6] = -2.000000
sbuf2[7] = -2.000000
```

Figure 3.34: A *SampleBuf* object example.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

The list of library functions for microsound synthesis	
ReadADC(var dur, channel=0, offset=0::samp)	
Read the input samples from the sound input at the channel <i>channel</i> for the duration <i>dur</i> . Giving <i>offset</i> parameter will give an offset to read. The return value is a Samples object. For instance, ReadDAC(512::samp, 1) will read the last 512 samples from the input channel 1 and ReadADC(128::samp, offset:10::samp) will returns the samples between the last 138th sample to 10th sample.	
WriteDAC(var samples, pan=0, offset=0::samp)	
Write out a <i>Samples</i> object for the sound output with panning (-1.0 - 1.0). To schedule the sound output in the future, the <i>offset</i> argument specify when the output is scheduled (the duration from now).	
PanOut(var samples, pan=0, offset=0::samp)	
Write out a <i>Samples</i> object for the sound output, with panning (<i>panning</i> between -1.0 to 1.0). To schedule the sound output in the future, the <i>offset</i> can be specify the duration from now when the output is scheduled.	
GenWindow(size, type=\hanning, amp=1.0)	
<i>GenWindow</i> generates a window of <i>type</i> with the amplitude <i>amp</i> . The return value is a Samples object that contains the window.	
ReadBuf(bufno, dur, rate=1.0, offset=0::samp, channel=0)	
<i>ReadBuf</i> returns a Sample object, taken from the buffer no. <i>bufno</i> , with the duration of <i>dur</i> . The <i>rate</i> parameter can be given to specify the reading rate (e.g., giving 2.0 for <i>rate</i> will result in octave higher samples and 0.5 for octave lower). The <i>offset</i> parameter specifies the duration from the beginning of the buffer, where the samples should be taken from the buffer and <i>channel</i> can be given when the buffer contains multi-channel audio samples.	
WriteBuf(bufno, samples, offset=0::samp, channel=0, mode=\overwrite)	
<i>WriteBuf</i> write the given Samples object <i>samples</i> , at the position in the buffer given by <i>offset</i> to the channel <i>channel</i> . The writing mode can be specified by <i>mode</i> , which can be either \overwrite or \overlap.add.	
ExtractWavesets(var bufno, channel=0)	
Analyse the sample data at the channel <i>channel</i> of the buffer No <i>bufno</i> and returns an array of Samples, each element of which is a waveset.	

Table 3.11: The list of library functions related to microsound synthesis in the prototype version of LC.

The list of library functions for microsound synthesis (FFT)
CreateBuf(var bufno, dur, numChannels=1)
<i>CreateBuf</i> creates a new buffer with the buffer no <i>bufno</i> , the duration of <i>dur</i> and the number of channels <i>numChannels</i> . If the buffer with the same <i>bufno</i> already exists, the old one is deleted and replaced by new one.
DeleteBuf(var bufno)
<i>DeleteBuf</i> deletes the buffer <i>bufno</i> from the current environment.
LoadSndFile(var bufno, file)
<i>LoadSndFile</i> creates the buffer no <i>bufno</i> and loads the sound file <i>file</i> . If the buffer with the same <i>bufno</i> already exists, the old one is deleted and replaced by a new one.
FFT(var samples)
<i>FFT</i> performs fast Fourier transform to given <i>samples</i> (The current version just performs real FFT only). The size of the Samples object <i>samples</i> must be the power of two. This function returns the FFT result as an array of Samples objects ([<i>real-part</i> , <i>imaginary-part</i>]).
IFFT(var real, imag)
<i>IFFT</i> performs inverse fast Fourier transform to given <i>real</i> and <i>imag</i> . The sizes of the Samples objects <i>real</i> and <i>imag</i> must be a power of two. This function returns the IFFT result as a Samples object (only real part of the signal).
CarToPol(var real, imag)
<i>CarToPol</i> performs cartesian to polar coordinate conversion and returns an array [<i>magnitude</i> , <i>phase</i>].
PolToCar(var mag, phase)
<i>PolToCar</i> performs polar to cartesian coordinate conversion and returns an array [<i>real</i> , <i>imag</i>].
PFFT(var samples, window=\hanning)
<i>PFFT</i> performs FFT, after windowing the window type of <i>window</i> and returns an array of [<i>magnitude</i> , <i>phase</i>]. While the same tasks can be performed by combining the other functions, this function is provided for the utility.
PIFFT(var mag, phase, window=\hanning)
<i>PIFFT</i> performs IFFT, after windowing the window type of <i>window</i> and returns a Samples object (real-part). While the same tasks can be performed by combining the other functions, this function is provided for the utility.

Table 3.12: The list of library functions related to microsound synthesis in the prototype version of LC (continued).

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

are also included as LC's sound synthesis framework can perform sound processing in the frequency domain within the same framework.

ReadADC and WriteDAC. First, we describe a simple example that reads the samples from the sound input and writes them out to the sound output. Figure 3.35 first simply routes the input to the left channel (of a stereo output device) for three seconds without any sound processing. Then, it reads both left and right channels and outputs the samples, applying a one second delay only to the right channel. The last part of the code plays the original input for three seconds, followed by the reversed sound. As the example reads the last 64 samples from ADC in every iteration and then outputs it to DAC, the code would cause the same amount of latency between the input and the output.

Synchronous, quasi-synchronous, and asynchronous granular synthesis. As briefly described in the earlier chapter, *synchronous granular synthesis* is a kind of granular synthesis, in which “sounds result from one or more streams of grains. Within each stream, one grain follows another, with a delay period between the grains. Synchronous means that the grains follow each other at regular intervals” (242, p.93). Quasi-synchronous granular synthesis is another variation of granular synthesis, in which “the grains follow each other at unequal intervals, where a random deviation factor determines the irregularity” (242, p.93). On the other hand, asynchronous granular synthesis “abandons the concept of linear streams of grains” and “instead, it scatters the grain over a specified duration within regions inscribed on the time-frequency plane” (242, p.96).

Figure 3.36 describes an example of simple synchronous granular synthesis in LC. The code first creates a `SampleBuffer` object with 1024 samples and fills it with four cycles of a sine wave. Then, these 1024 samples are applied to a Hanning window to create a grain (lines 01-11). The code between lines 13-18 performs synchronous granular synthesis with this grain, by writing the grain to the sound output and then waiting for the 1/4 duration of the grain so that the four grains can overlap each other. It should be noted that the code reuses the same grain in the main loop. As *Samples* in LC is an immutable object, it can be guaranteed that once microsound is generated,

A ReadADC/WriteDAC example

```

01: //first 3 seconds, perform play-through (to the left channel)
02: within(3::second){
03:     while(true){
04:         //read 64 samples from ADC
05:         var in = ReadADC(64::samp);
06:         WriteDAC(in);
07:         //advance the time for the duration of the input samples.
08:         now += in.dur;
09:     }
10: }
11: //then read/write to both channels, applying 1 sec delay to the left channel.
12: within(3::second){
13:     while(true){
14:         var l = ReadADC(64::samp); //left channel
15:         var r = ReadADC(64::samp, 1); //right channel
16:         WriteDAC(l,0);
17:         WriteDAC(r,1, offset:1::second);
18:         now += l.dur;
19:     }
20: }
21:
22: //play-through for 3 seconds, then play it backward.
23: within(3::second){
24:     var pos = 0::second;
25:     var dur = 512::samp;
26:     var win = GenWindow(512::samp, \hanning);
27:     var ipos= 6::second;
28:     while(true){
29:         //read 512 samples at 'pos' and send it to DAC.
30:         var original= ReadADC(dur);
31:         WriteDAC(original);
32:         //output the reversed version.
33:         //ipos moves from 6 sec to 3 sec since the start
34:         WriteDAC(original->reverse(), offset:ipos);
35:         //advance the reading position and the logical time
36:         now += dur;
37:         //adjust 'ipos' so that the next grain for reversed play-back can be
38:         //should be placed the right before the previous one.
39:         ipos -= dur * 2;
40:     }
41: }

```

Figure 3.35: A ReadADC/WriteDAC example.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

A synchronous granular synthesis example

```
01: //create a SampleBuffer and fill it with 256 samp sinewave * 4 cycles
02: var sbuf = new SampleBuffer(1024);
03: for (var i = 0; i < sbuf.size; i += 1){
04:     sbuf[i] = Sin(3.14159265359 * 2 * (i * 4.0 / sbuf.size));
05: }
06:
07: //create a grain.  apply an envelope to sinewave and resample it to 440 samples
08: var tmp = sbuf->toSamples();
09: var win = GenWindow(1024::samp, \hanning);
10:
11: var grain = tmp->applyEnv(win)->resample(440)->amplify(0.25);
12:
13: within(5::second){
14:     while(true){
15:         WriteDAC(grain);
16:         now += grain.dur / 4;
17:     }
18: }
```

Figure 3.36: A synchronous granular synthesis example.

it will be never changed and be reused even when grains must overlap each other as in this example. Figure 3.37 describes an example of quasi-synchronous granular synthesis in LC. As shown, the only difference from Figure 3.36 in this example is line 16, where the interval to the next grain is modified to give a random value between 1 - 20 ms.

Figure 3.38 describes an example of simple asynchronous granular synthesis. Unlike two previous examples, it randomly distributes 2048 grains, each with a various duration (64 - 512 samples) within the next five seconds. The scheduling of these grains is done synchronously at once (without the advance of logical time).

Granular pitch-shifting and time-stretching. While there exist various phase-vocoding techniques for time-stretching and pitch-shifting, a microsound synthesis technique called *granular-sampling* (189) is also often used in computer music compositions due to its computational efficiency. Generally speaking, both techniques involve the creation of grains by extracting short fragments from the original sound samples (or real-time input) and applying a window function to the fragments. Then, these grains

A quasi-synchronous granular synthesis example

```
01: //create a SampleBuffer and fill it with 256 samp sinewave * 4 cycles
02: var sbuf = new SampleBuffer(1024);
03: for (var i = 0; i < sbuf.size; i += 1){
04:   sbuf[i] = Sin(3.14159265359 * 2 * (i * 4.0 / sbuf.size));
05: }
06:
07: //create a grain.  apply an envelope to sinewave and resample it to 440 samples
08: var tmp = sbuf->toSamples();
09: var win = GenWindow(1024::samp, \hanning);
10:
11: var grain = tmp->applyEnv(win)->resample(440)->amplify(0.25);
12:
13: within(5::second){
14:   while(true){
15:     WriteDAC(grain);
16:     now += Rand(1, 20)::ms;
17:   }
18: }
```

Figure 3.37: A quasi-synchronous granular synthesis example.

An asynchronous granular synthesis example

```
01: //this time white noise is used as the sound source/
02: var sbuf = new SampleBuffer(1024);
03: for (var i = 0; i < sbuf.size; i += 1){
04:   sbuf[i] = Rand(-1.0, 1.0);
05: }
06:
07: var tmp = sbuf->toSamples();
08: var win = GenWindow(1024::samp, \hanning);
09:
10: var grain = tmp->applyEnv(win)->amplify(0.5);
11:
12: //distributing 2048 grains randomly for 5 seconds.
13: for(var i = 0; i < 2048; i += 1){
14:   var offset = Rand(0, 5.0)::second;
15:   var snd = grain->resample(Rand(64, 512));
16:   WriteDAC(snd, offset: offset);
17: }
```

Figure 3.38: An asynchronous granular synthesis example.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

are scheduled for playback, overlapping onto others. For pitch-shifting, each grain is resampled shorter or longer, according to the desired pitch before playing back. For instance, resampling a grain composed of 441 samples to 882 samples creates a new grain whose pitch is one octave lower pitch when both of them are played back in the same sample rate.

On the other hand, time-stretching by granular sampling is made possible by advancing the reading position to extract sound samples from the original material at the different rate than playing back. To give an example, assume such a situation as follows. The first 1024 samples were read from the beginning of the sound buffer and then played it back. After 512 samples in logical time, the next 1024 samples are read at the position of the 256th sample in the buffer and played it back. By repeating such a procedure, the original sound samples can be time-stretched twice as long as the original sound without altering its pitch.

Figure 3.39 and Figure 3.40 describe examples of pitch-shifting and time-stretching by granular-sampling in LC. As shown, even granular sampling of real-time audio input can be done by a simple algorithm. As seen in Figure 3.39 (on line 27), in LC's programming model, it is possible to directly apply a filter to a microsound if there is any need to avoid aliasing caused by resampling. The time-stretching example for real-time audio input doesn't require the sound buffer and can be realized by directly scheduling the grains obtained from the sound input.

Waveset synthesis. As described in Section 2.3, *waveset synthesis techniques* also belong to the family of microsound synthesis as well as granular synthesis techniques. To perform waveset synthesis techniques, it is necessary to analyse the sound material and to extract wavesets, by examining the zero-crossings in the sound¹. Generally speaking, waveset synthesis techniques are performed by transforming and rescheduling such wavesets. See Table 3.13 (reproduced from 2.2).

LC provides the *ExtractWavesets* function, which analyses a sound buffer to extract the wavesets and returns an array of Samples object, each of which represents an extracted waveset. Waveset synthesis techniques can be performed in LC by applying

¹As described in Section 2.3, waveset is defined as “the distance from zero-crossing to a 3rd zero crossing”, whereas wavecycle is defined as “the wavelength of sound, where clearly pitched” (323, p.50).

A granular pitch-shifting example

```

01: //load a sound file to the buffer no.0
02: LoadSndFile(0, "/sound/sample1.aif");
03: //play back the original sound for 2::second
04: within(2::second){
05:     var pos = 0::second;
06:     var dur = 512::samp;
07:     while(true){
08:         //read 512 samples at 'pos' and send it to DAC.
09:         var sample = ReadBuf(0, dur, offset:pos);
10:         WriteDAC(sample);
11:         //advance the reading position and the logical time
12:         pos += dur;
13:         now += dur;
14:     }
15: }
16: //pitch-shifting (octave upper)
17: within(2::second){
18:     var pos = 0::second;
19:     var dur = 512::samp;
20:     var win = GenWindow(512::samp, \hanning);
21:     while(true){
22:         //read 512 samples at 'pos'
23:         var sample = ReadBuf(0, dur * 2, offset:pos);
24:         //resample to 1/2 size and apply an hanning window
25:         var tmp = sample->resample(dur);
26:         //if there is a need to consider aliasing, once can apply a filter as below.
27:         //tmp = tmp->fir([0.5, 0, 0.5]);
28:         var grain= tmp->applyEnv(win);
29:         WriteDAC(grain);
30:         //advance the reading position and the logical time
31:         //but with 2 overlaps
32:         pos += dur / 2;
33:         now += dur / 2;
34:     }
35: }

```

Figure 3.39: A granular sampling example (pitch-shifting).

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
36: //pitch-shifting (real-time input for 5 seconds) (octave down)
37: within(5::second){
38:   var pos = 0::second;
39:   var dur = 512::samp;
40:   var win = GenWindow(512::samp, \hanning);
41:   while(true){
42:     //read 512 samples from ADC
43:     var sample = ReadADC(dur);
44:     //resample to 1/2 size and apply an hanning window
45:     var tmp = sample->resample(sample.size * 2);
46:     var grain= tmp->applyEnv(win);
47:     WriteDAC(grain);
48:     //advance the reading position and the logical time
49:     //but with 2 overlaps
50:     now += dur / 2;
51:   }
52: }
53:
54: //harmonizing major 3rd to the input for 5 seconds
55: within(5::second){
56:   var dur = 512::samp;
57:   var win = GenWindow(dur, \hanning);
58:
59:   while(true){
60:     //read 512 samples at 'pos' and send it to DAC.
61:     var original= ReadADC(dur);
62:
63:     //512 * 1.26 samples -> resample to 512 -> major 3rd upper
64:     var sample = ReadADC(dur * 1.26);
65:     var tmp = sample->resample(original.size);
66:     var grain = tmp->applyEnv(win);
67:
68:     WriteDAC(original->amplify(0.5));
69:     WriteDAC(grain);
70:     //advance the logical time
71:     now += dur / 2;
72:   }
73: }
```

Figure 3.39: A granular sampling example (pitch-shifting) (continued).

A granular time-stretching example

```
01: //load a sound file to the buffer no.0
02: LoadSndFile(0, "/sound/sample1.aif");
03:
04: //time-stretching (twice as long as the original)
05: within(4::second){
06:     var pos = 0::second;
07:     var dur = 512::samp;
08:     var win = GenWindow(512::samp, \hanning);
09:     while(true){
10:         //read 512 samples at 'pos'
11:         var sample = ReadBuf(0, dur, offset:pos);
12:         var grain= sample->applyEnv(win);
13:         WriteDAC(grain);
14:         pos += dur / 4;
15:         now += dur / 2;
16:     }
17: }
18: //time-stretching (half as long as the original)
19: within(4::second){
20:     var pos = 0::second;
21:     var dur = 512::samp;
22:     var win = GenWindow(512::samp, \hanning);
23:     while(true){
24:         //read 512 samples at 'pos'
25:         var sample = ReadBuf(0, dur, offset:pos);
26:         var grain= sample->applyEnv(win);
27:         WriteDAC(grain);
28:         pos += dur / 2;
29:         now += dur / 4;
30:     }
31: }
```

Figure 3.40: A granular sampling example (time-stretching).

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
32: //time-stretching (real-time sound input. the original is played-through
33: //on the left channel and time-stretched (twice as long as the original)
34: //on the right channel)
35: within(4::second){
36:   var wpos = now; //write timing for time-stretched version (in time line)
37:   var dur = 512::samp;
38:   var win = GenWindow(512::samp, \hanning);
39:   while(true){
40:     var sample = ReadADC(dur);
41:     var grain = sample->applyEnv(win);
42:     WriteDAC(grain);
43:     WriteDAC(grain, 1, offset:(wpos - now));
44:     wpos += dur / 2;
45:     now += dur / 4;
46:   }
47: }
```

Figure 3.40: A granular sampling example (time-stretching) (continued).

methods/library functions (or directly manipulating sample values by indexed-access if necessary) to these wavesets. For instance, in Wishart’s Composer’s Desktop Project software, 17 different waveset synthesis techniques are available as seen in Table 3.13. The following code examples describe the examples of waveset inversion, waveset distortion, waveset transposition, waveset substitution, and waveset harmonic distortion in LC.

Figure 3.41 is a simple example to reproduce the original sound by iterating each waveset in the array of wavesets returned by the *ExtractWavesets* function. Waveset inversion can be performed by inverting each waveset by the *invertWS* method, and waveset distortion can be performed by squaring each waveset by the *mul* method¹ as seen in Figure 3.42 and Figure 3.43; The only difference between these two examples is the use of the *invertWS* method (line 10 in Figure 3.42) and the *mul* method (line 10 in Figure 3.43).

Figure 3.44 describes an example of waveset transposition in LC. As shown, waveset transposition can be performed simply by resampling the original wavesets and schedul-

¹While the code will be computationally inefficient, it is also possible to invert and distort a waveset by examining each sample within a Sample object and by using SampleBuffer to generate a new Sample object.

ing it for output. Figure 3.45 describes a waveset substitution example. In this example *SampleBuffer* is used to create a waveform to replace the original wavesets. Then, it is resampled to the same size as each original waveset and written out to the sound output. Figure 3.46 is an example of waveset harmonic distortion. In this example, each original waveset is resampled and amplified. These resampled/weighted wavesets are scheduled so that it can overlap-add onto the original wavesets.

Figure 3.47 and Figure 3.48 are examples to apply waveset synthesis techniques to the real-time sound synthesis input. The former example combines waveset inversion and waveset transposition, and the latter performs waveset harmonic distortion. As presented, the code only has to involve a small modification to extract wavesets from real-time sound input. It should also be noted that the combination of different waveset synthesis techniques as seen in Figure 3.47 can be still easily made without involving much complexity.

Thus, various waveset synthesis techniques can be performed by combining basic manipulations scheduling of wavesets in LC's programming model. While only five of the 17 waveset synthesis techniques in Table 3.13 were shown in these code examples, all 17 waveset synthesis techniques in the list can be performed in LC; as long as a waveset synthesis technique can be described algorithmically, the waveset synthesis technique can be implemented in LC, since LC allows direct access to the samples within a *Samples* object by indexed-access, and *Samples* object can be scheduled with sample-rate accuracy. For instance, waveset enveloping can be performed by using the *append* method to combine N wavesets together and *applyEnv* to applying an envelope to the combined wavesets.

Fourier Transform and Inverse Fourier Transform. LC can also perform Fourier transform and Inverse Fourier transform. The related library functions are listed in Table 3.11. As *Samples* and *SampleBuffer* objects are used to represent the data for FFT/IFFT, various synthesis techniques in the frequency domain can be performed in LC within the same microsound synthesis framework.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

Waveset transposition	substitutes N copies of a waveset in the place of M wavesets, for example 2 in the place of 1, or 1 in the place of 4, for doubling and quartering of frequency, respectively
Waveset reversal	reverses individual wavesets while retaining their order; reversals can be of each individual waveset or collections of N wavesets at a time.
Waveset shaking	alternates between compressing (in time) and expanding (in time) successive wavesets
Waveset inversion	inverts the phase of all wavesets in a signal; in Wishart's diagram half-wavesets are inverted
Waveset omission	deletes every Nth waveset, leaving silence in its place; controlled by initial and final density from 0 to 100 %
Waveset shuffling	permutes collections of wavesets. A simple shuffle of successive wavesets starting with (a, b, c, d) becomes (d, c, a, b)
Waveset distortion	distorts a signal by squaring and cubing the signal; the example given in Wishart does not indicate how it is tied to waveset boundaries
Waveset substitution	replaces wavesets by a stipulated waveform of the same amplitude, frequency, and time span as the original waveset
Waveset harmonic distortion	superimposes N harmonics on the waveset fundamental with a scaling factor M relative to the previous harmonic
Waveset averaging	creates a signal containing N new wavesets that are the average duration and the average amplitude and time function of the N original wavesets; the overall duration of the signal is unchanged
Waveset enveloping	applies an envelope to 1 to N wavesets at a time
Waveset transfer	substitutes the waveform of the wavesets in signal A into the time frames of the wavesets in signal B
Waveset interleaving method 1	substitutes wavesets from signal A into alternate wavesets of signal B; applied either to individual wavesets or groups.
Waveset interleaving method 2	interleaves wavesets from signal A with wavesets of signal B, thus lengthening the output signal; applied either to individual wavesets or groups
Waveset time-stretching	repeats each waveset N times
Waveset time-shrinking	retains only the first of every N wavesets or retains only the loudest of every N wavesets
Waveset normalizing	normalizes every N wavesets above a stipulated amplitude threshold, thus a 10% threshold has a greater effect than a 90% threshold

Table 3.13: Waveset transformations in the Composer's Desktop Project software (242, p.207), reproduced from Table 2.2.


```
01: //load the sound file onto the buffer no.0
02: LoadSndFile(0, "/sound/sample1.aif");
03:
04: //ExtractWavesets returns the array of wavesets extracted from the buffer.
05: var wavesets = ExtractWavesets(0);
06:
07: //output the array of wavesets from the beginning one-by-one
08: //to reconstruct the original sound.
09: for (var i = 0; i < wavesets.size; i +=1){
10:     WriteDAC(wavesets[i]);
11:     now += wavesets[i].dur;
12: }
```

Figure 3.41: A waveset example to reproduce the original sound in LC.

```
01: //load the sound file onto the buffer no.0
02: LoadSndFile(0, "/sound/sample1.aif");
03:
04: //ExtractWavesets returns the array of wavesets extracted from the buffer.
05: var wavesets = ExtractWavesets(0);
06:
07: //iterate through the array of wavesets from the beginning one-by-one
08: //to perform waveset inversion.
09: for (var i = 0; i < wavesets.size; i +=1){
10:     //invert the waveset.
11:     var inverted = wavesets[i]->invertWS();
12:     WriteDAC(inverted);
13:     now += inverted.dur;
14: }
```

Figure 3.42: A waveset inversion example in LC.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
01:  //load the sound file onto the buffer no.0
02:  LoadSndFile(0, "/sound/sample1.aif");
03:
04:  //ExtractWavesets returns the array of wavesets extracted from the buffer.
05:  var wavesets = ExtractWavesets(0);
06:
07:  //iterate through the array of wavesets from the beginning one-by-one
08:  //to perform waveset distortion.
09:  for (var i = 0; i < wavesets.size; i +=1){
10:    //distort the waveset.
11:    var distorted = wavesets[i]->mul(wavesets[i]);
12:    WriteDAC(distorted);
13:    now += distorted.dur;
14:  }
```

Figure 3.43: A waveset distortion example in LC.

Figure 3.49 and Figure 3.50 show simple *cross synthesis* examples in LC. To perform cross synthesis, “two input signals are required: signal A’s spectrum is convolved with the amplitude spectrum of signal B. Thus, the pitch/phase information of signal A and the time varying spectral envelope of signal B are combined to form the output signal” (265). While Figure 3.49 describes the straight forward implementation that uses *FFT/IFFT* functions and includes such tasks as windowing by Hanning window, conversions between Cartesian coordinates and polar coordinates, Figure 3.50 uses *PFFT/PIFFT* functions to simplify the implementation. *PFFT/PIFFT* functions can apply a windowing function and the conversion between Cartesian coordinates and polar coordinates at once, while both examples implement the same algorithm.

Figure 3.51 describes a simple example of time-stretching by phase vocoding (100)(113). As in Figure 3.50, *PFFT/PIFFT* library functions are used to simplify the implementation. While the code uses the *sub* method on line 23 and the *accumulatePhase* method on line 26, a user can also write the code to compute the subtraction and accumulation by using *Samples* and *SamplesBuffer* objects, since each sample in a *Samples* object is directly accessible by indexed-access. Hence, any arithmetic computation on samples can be directly implemented in LC, if necessary to implement a certain synthesis technique.

```

01: //load the sound file onto the buffer no.0
02: LoadSndFile(0, "/sound/sample1.aif");
03:
04: var wavesets = ExtractWavesets(0);
05:
06: //4 wavesets in the space of 1
07: for (var i = 0; i < wavesets.size; i +=1 ){
08:     //resample the wave set to 1/4 of the original size.
09:     var ws = wavesets[i];
10:     var resampled = ws->resample(ws.size / 4);
11:
12:     //repeat 4 times.
13:     for (var j = 0; j < 4; j += 1){
14:         WriteDAC(resampled, offset:resampled.dur * j);
15:     }
16:
17:     //sleep until the next scheduling timing.
18:     now += ws.dur;
19: }
20:
21: //1 wavesets in the space of 2
22: for (var i = 0; i < wavesets.size - 1; i += 2){
23:     //get two wavesets from the array
24:     var ws1 = wavesets[i];
25:     var ws2 = wavesets[i + 1];
26:
27:     //resample the 1st one to the size of (1st + 2nd)
28:     var resampled = ws1->resample(ws1.size + ws2.size);
29:
30:     //write it out and sleep until the next scheduling timing.
31:     WriteDAC(resampled);
32:     now += resampled.dur;
33: }

```

Figure 3.44: A waveset transposition example in LC.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
01:  //create a square wave
02:  var tmp = new SampleBuffer(256);
03:  for (var i = 0; i < 128; i += 1){
04:    tmp[i] = 1.0;
05:  }
06:  for (var i = 128; i < 256; i += 1){
07:    tmp[i] = -1.0;
08:  }
09:  var square = tmp->toSamples();
10:
11:  LoadSndFile(0, "/sound/sample1.aif");
12:  var wavesets = ExtractWavesets(0);
13:  for (var i = 0; i < wavesets.size; i += 1){
14:    var ws = wavesets[i];
15:    var out= square->resample(ws.size)->amplify(ws->maxAmp());
16:    WriteDAC(out);
17:    now += out.dur;
18:  }
```

Figure 3.45: A waveset substitution example in LC.

Figure 3.52 and Figure 3.53 perform time-stretching by phase-vocoding for real-time sound input. The algorithm in Figure 3.52 can be CPU intensive because FFT/IFFT occurs in a short period, depending on the given parameters. On the other hand, the Figure 3.53 example stores the real-time input to the buffer once and then reads the buffer, performing time-stretching. Such a strategy results in less computational cost compared to the Figure 3.52 example, as the number of FFT/IFFT function calls can be reduced even for the same parameters¹.

3.2.3.3 The collaboration between microsounds and unit-generators

While LC's programming model for microsound synthesis is significantly different from the traditional unit-generator concept as described so far. LC provides the mechanism to facilitate the collaboration between these two different concepts. Table 3.14 and Table 3.15 list the related methods Patch object and UGen objects for such collaboration

¹The biggest difference between these two algorithms is the period of FFT/IFFT calls. In Figure 3.52, the example is defined by the hopsize to read the original input, while the period of FFT/IFFT in the Figure 3.53 example is defined by the hopsize to output the time-stretched sound; the FFT/IFFT period of the latter algorithm is longer. Thus, less FFT/IFFT function calls are made in the same duration in Figure 3.53, and it is more suitable for real-time sound processing.

```
01: LoadSndFile(0, "/sound/sample1.aif");
02: var wavesets = ExtractWavesets(0);
03:
04: //weights for 1st and 2nd harmonics
05: var weight1 = 0.5;
06: var weight2 = 1.2;
07:
08: for (var i = 0; i < wavesets.size; i +=1 ){
09:     //create the 2nd and 3rd harmonics from the original waveset.
10:     var ws = wavesets[i];
11:     var harm1 = ws->resample(ws.size / 2)->amplify(weight1);
12:     var harm2 = ws->resample(ws.size / 3)->amplify(weight2);
13:
14:     //schedule each waveset
15:     PanOut(ws, 0.0);
16:     //put 1st harmonics that overlap-add the original
17:     PanOut(harm1, -1.0);
18:     PanOut(harm1, -1.0, offset:harm1.dur);
19:     //2nd harmonics
20:     PanOut(harm2, 1.0);
21:     PanOut(harm2, 1.0, offset:harm2.dur);
22:     PanOut(harm2, 1.0, offset:harm2.dur * 2);
23:
24:     now += ws.dur;
25: }
```

Figure 3.46: A waveset harmonic distortion example in LC.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
01: while(true){
02:     //read ADC for 50 msec, which is the largest waveset size we assume.
03:     var input = ReadADC(50::ms);
04:     //get the first wave set found in the input.
05:     var waveset = input->getWaveset();
06:     //if there is no waveset found getWaveset() returns null.
07:     if (waveset == null){
08:         //then find the first zero-crossing index to discard unnecessary input.
09:         var adv= waveset->getZeroCrossingIndex()::samp;
10:         //if there is no crossing, getZeroCrossingIndex() returns -1.
11:         if (adv < 0::samp){
12:             //then discard the whole input for the last 50 msec.
13:             adv = 50::ms;
14:         }
15:         //advance the time to skip the unnecessary input.
16:         now += adv;
17:         continue;
18:     }
19:     //perform waveset inversion, then waveset transposition.
20:     waveset = waveset->invertWS();
21:     waveset = waveset->resample(waveset.size);
22:     PanOut(waveset);
23:     PanOut(waveset, offset:waveset.dur);
24:
25:     now += waveset.dur;
26: }
```

Figure 3.47: A waveset inversion + waveset transposition example (real-time sound input) in LC.

```

01: //performing waveset harmonic distortion to the real-time input
02: var weight1 = 0.5;
03: var weight2 = 0.5;
04:
05: while(true){
06:     //read ADC for 50 msec, which is the largest waveset size we assume.
07:     var input = ReadADC(50::ms);
08:
09:     //get the first wave set found in the input.
10:     var waveset = input->getWaveset();
11:     //if there is no waveset found getWaveset() returns null.
12:     if (waveset == null){
13:         //then find the first zero-crossing index to discard unnecessary input.
14:         var adv= waveset->getZeroCrossingIndex()::samp;
15:         //if there is no crossing, getZeroCrossingIndex() returns -1.
16:         if (adv < 0::samp){
17:             //then discard the whole input for the last 50 msec.
18:             adv = 50::ms;
19:         }
20:         //advance the time to skip the unnecessary input.
21:         now += adv;
22:         continue;
23:     }
24:
25:     //now we got a waveset from real-time input.
26:     //perform waveset harmonic distortion now.
27:     var harm1 = waveset->resample(waveset.size / 2)->amplify(weight1);
28:     var harm2 = waveset->resample(waveset.size / 3)->amplify(weight2);
29:     PanOut(waveset);
30:     PanOut(harm1, -1.0);
31:     PanOut(harm1, -1.0, offset:harm1.dur);
32:     PanOut(harm2, 1.0);
33:     PanOut(harm2, 1.0, offset:harm2.dur);
34:     PanOut(harm2, 1.0, offset:harm2.dur * 2);
35:
36:     //sleep until the next input.
37:     now += waveset.dur;
38: }

```

Figure 3.48: A waveset harmonic distortion example (real-time sound input) in LC.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
01: //loading the sound files onto the buffers.
02: LoadSndFile(0, "/sound/violin.wav");
03: LoadSndFile(1, "/sound/kill_humans.wav");
04:
05: //the duration of each window, the number of overlapping and the hanning window.
06: var dur = 1024::samp;
07: var overlap = 4;
08: var win = GenWindow(dur, \hanning);
09:
10: //process 800 frames.
11: for(var i = 0; i < 800; i += 1){
12:   //first, extract the sound fragments from the given sound materials.
13:   var src1 = ReadBuf(0, dur, offset:i * dur / overlap);
14:   var src2 = ReadBuf(1, dur, offset:i * dur / overlap);
15:   //apply the hanning window to both of them.
16:   var wsrc1 = src1->applyEnv(win);
17:   var wsrc2 = src2->applyEnv(win);
18:
19:   //now perform FFT. FFT returns an array of Samples objects, [real, imaginary].
20:   var fft1 = FFT(wsrc1);
21:   var fft2 = FFT(wsrc2);
22:   //perform cartesian-to-polar conversion.
23:   var pol1 = CarToPol(fft1[0],fft1[1]);
24:   var pol2 = CarToPol(fft2[0],fft2[1]);
25:
26:   //applying the magnitude of the voice material to the violin sound.
27:   //to perform cross synthesis.
28:   var pved = pol1[0]->mul(pol2[0]);
29:
30:   //convert it back to cartesian form
31:   var car = PolToCar(pved, pol1[1]);
32:
33:   ///perform IFFT and apply the hanning window again.
34:   var ifft = IFFT(car[0], car[1]);
35:   var wifft= ifft->applyEnv(win);
36:
37:   //send the samples to the sound output, sleep until the next frame.
38:   WriteDAC(wifft);
39:   now += src1.dur / overlap;
40: }
```

Figure 3.49: A FFT/IFFT example (cross synthesis).


```
01: //loading the sound files onto the buffers.
02: LoadSndFile(0, "/sound/violin.wav");
03: LoadSndFile(1, "/sound/kill_humans.wav");
04:
05: //the duration of each window, the number of overlapping and the hanning window.
06: var dur = 1024::samp;
07: var overlap = 4;
08:
09: //process 800 frames.
10: for(var i = 0; i < 800; i +=1 ){
11:     //first, extract the sound fragments from the given sound materials.
12:     var src1 = ReadBuf(0, dur, offset:i * dur / overlap);
13:     var src2 = ReadBuf(1, dur, offset:i * dur / overlap);
14:
15:     //now perform FFT. PFFT apply a window function and
16:     //returns returns an array of Samples objects, [magnitude, phase].
17:     var pfft1 = PFFT(src1, \hanning);
18:     var pfft2 = PFFT(src2, \hanning);
19:
20:     //perform cross synthesis.
21:     var ppved = pfft1[0]->mul(pfft2[0]);
22:
23:     //perform IFFT
24:     var pifft = PIFFT(ppved, pfft1[1], \hanning);
25:
26:     //send the samples to the sound output, sleep until the next frame.
27:     WriteDAC(pifft);
28:     now += src1.dur / overlap;
29: }
```

Figure 3.50: A PFFT/PIFFT example (cross synthesis).

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
01: //load the sound files.
02: LoadSndFile(0, "/sound/kill_humans.wav");
03: //perform the time-stretching for 24 seconds, using the time-out feature.
04: within(24::second){
05:     //the parameters (window/frame size, the initial reading position)
06:     var dur = 1024::samp;
07:     var overlap = 4;
08:     var stretch = 4;
09:     var pos = dur / (overlap * stretch);
10:
11:     //extract the first frame and keep the information.
12:     var firstFrame= ReadBuf(0, dur);
13:     var ffted = PFFT(firstFrame);
14:     //take the initial previous phase and accumulated phase from the first frame.
15:     var prevPhase = ffted[1];
16:     var accumPhase= prevPhase;
17:
18:     //main loop for phase vocoder time-stretching.
19:     while(true){
20:         //read the next frame and perform FFT
21:         var sample= ReadBuf(0, dur, offset:pos);
22:         var ffted = PFFT(sample);
23:
24:         //get the phase difference between the previous frame and current frame.
25:         //multiply it by 'stretch' (= resynthesis hop size / analysis hop size)
26:         //to compute the phase increment in resynthesis version.
27:         var dif = ffted[1]->sub(prevPhase)->amplify(stretch);
28:         accumPhase = accumPhase->accumulatePhase(dif);//accumulate it
29:
30:         //update the previous phase info
31:         prevPhase = ffted[1];
32:
33:         //perform IFFT with the accumulated phase and output the sound.
34:         var iffted = PIFFT(ffted[0], accumPhase);
35:         PanOut(iffted, 0);
36:
37:         //move the reading position and sleep until the next frame.
38:         pos += dur / (overlap * stretch);
39:         now += dur / overlap;
40:     }
41: }
```

Figure 3.51: A PFFT/PIFFT example (time-stretching).

```
01: //time-stretching the real-time audio input by phase vocoding.
02: //5 seconds of real-time input is immediately extended to 1.5 times
03: //as long as the original. As this can be very CPU intensive, depending
04: //on the overlap, stretch, dur as they define how often PFFT and PIFFT are
05: //called and how long each call can consume time for computation.
06: within(5::second){
07:     var dur = 4096::samp;
08:     var overlap = 4;
09:     var stretch = 1.5;
10:     take the first frame.
11:     var prevPhase = PFFT(ReadADC(dur))[1];
12:     var accumPhase= prevPhase;
13:     now += dur / (overlap * stretch);
14:     var wpos = now;
15:
16:     while(true){
17:         var sample = ReadADC(dur);
18:
19:         var ffted = PFFT(sample);
20:
21:         var dif = ffted[1]->sub(prevPhase)->amplify(stretch);
22:         prevPhase = ffted[1];
23:
24:         accumPhase = accumPhase->accumulatePhase(dif);
25:         var iffted = PIFFT(ffted[0], accumPhase);
26:         PanOut(iffted, offset:(wpos - now));
27:         wpos+= dur / overlap;
28:         //wait until the next frame to read from ADC
29:         now += dur / (overlap * stretch);
30:     }
31: }
```

Figure 3.52: A PFFT/PIFFT example (time-stretching real-time input).

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
01: //perform recording for 10 second during time-stretching.
02: var recordDur = 10::second;
03: //create the buffer for the recording
04: CreateBuf(0, recordDur);
05: //frame size, overlap, stretch parameters.
06: var framesize = 1024::samp;
07: var overlap = 4;
08: var hopsize = framesize / overlap;
09: var stretch = 4;
10:
11: //write the first frame onto the buffer.
12: var input = ReadADC(framesize);
13: WriteBuf(0, input);
14: var prevPhase = PFFT(input)[1];
15: var accumPhase= prevPhase;
16:
17: //the initial write/read positions on the buffer.
18: var wpos= framesize;
19: var rpos= 0::samp;
20:
21: within(recordDur * stretch){
22:   while(true){
23:     //wait for the hop size and update the read position on the buffer.
24:     now += hopsize;
25:     rpos+= hopsize / stretch;
26:     //write the next input to the buffer.
27:     if (wpos < recordDur){
28:       var input = ReadADC(hopsize);
29:       WriteBuf(0, input, offset:wpos);
30:       wpos += hopsize;
31:     }
32:     //perform time-stretching
33:     var snd = ReadBuf(0, framesize, offset:rpos);
34:     var ffted = PFFT(snd);
35:
36:     var dif = ffted[1]->sub(prevPhase)->amplify(stretch);
37:     prevPhase = ffted[1];
38:
39:     accumPhase = accumPhase->accumulatePhase(dif);
40:     var iffted= PIFFT(ffted[0], accumPhase);
41:
42:     PanOut(iffted, 0.0);
43:   }
44: }
```

Figure 3.53: A PFFT/PIFFT example (time-stretching real-time input, with a buffer).

The list of UGen’s methods for the collaboration between microsounds and unit-generators

retain

The duration of the output samples to keep is stored in this field *retain*. By assigning a duration, the unit-generator will update the size of the internal buffer to keep the output so to store the output samples. The default value is 0::samp. The samples can be taken by calling ‘read’ method.

process(var self, dur)

Processes the patch for *dur*. This function is used for the collaboration between the traditional unit-generator-based sound synthesis and LC’s microsound synthesis framework.

read(var self, dur, outlet=\defout, offset=0::samp)

Reads the output for the last *dur* duration from *outlet*, with *offset*. This function returns a *Samples* object and is used for the collaboration between the unit-generator framework and LC’s microsound synthesis framework.

pread(var self, dur, outlet=\defout)

Processes the patch for *dur* and returns the output from the outlet *outlet*. This function returns a *Samples* object and is used for the collaboration between the unit-generator framework and LC’s microsound synthesis framework.

write(var self, samples, inlet=\defin, offset=0::samp)

Writes a *Samples* object to the inlet *inlet* of the unit-generator, with *offset* to the future. This function is used for the collaboration between the unit-generator framework and LC’s microsound synthesis framework

Table 3.14: The list of UGen’s methods for the collaboration between microsounds and unit-generators in LC.

between microsounds and unit-generators. Every UGen object is equipped with these methods in Table 3.14, regardless of its actual type.

As there are still many sound synthesis/processing algorithms that can be better described by the unit-generator concept (e.g., envelope-shaping, FM-synthesis, amplitude-modulation, reverberation and the like) such a mechanism that can combine two different programming models are of significant importance. By using these methods, the collaboration between these two different programming models can be performed in LC as described in the following examples.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

The list of Patch's methods for the collaboration between microsounds and unit-generators

write(var self, samples, inlet=\defn, offset=0::samp, ugen_inlet=\defn)

Writes a Samples object to the inlet *ugen_inlet* of the unit-generator/subpatch stored at the slot *inlet*, with *offset* to the future. This function is used for the collaboration between the unit-generator framework and LC's microsound synthesis framework.

process(var self, dur)

Processes the patch for *dur*. This function is used for the collaboration between the unit-generator framework and LC's microsound synthesis framework.

read(var self, dur, outlet=\defout, offset=0::samp, ugen_outlet=\defout)

Reads the output for the last *dur* duration from *ugen_outlet* of the unit-generator/subpatch stored at the slot *outlet*. This function returns a *Samples* object and is used for the collaboration between the unit-generator-based framework and LC's microsound synthesis framework.

pread(var self, dur, outlet=\defout, ugen_outlet=\defout)

Processes the patch for *dur* and returns the output from the outlet *ugen_outlet* of the unit-generator/subpatch stored at the slot *outlet*. This function returns a *Samples* object and is used for the collaboration between the unit-generator framework and LC's microsound synthesis framework.

Table 3.15: The list of Patch's methods for the collaboration between microsounds and unit-generators in LC.

Generating a microsound from the output of a unit-generator or a patch.

While the unit-generators in LC can constitute a patch, which can be played along with the advance of logical time, both a patch and a unit-generator has the feature to create a new `Samples` object from its output, regardless of the advance of logical time. Thus, a user can easily generate `Samples` objects by using unit-generators and patches, and then directly apply various manipulations to the generated `Samples` objects to perform microsound synthesis. For instance, in the Figure 3.36 example, `SampleBuffer` and `for-loop` are used to generate a `Samples` object that contains four cycles of a sine wave, with which the example performs synchronous granular synthesis. Yet, this task can also be performed with `Sin~` unit-generator by using *retain* field and *read* method.

Figure 3.54 and Figure 3.55 are examples that generate `Samples` objects from the output of a unit-generator. In Figure 3.54, by assigning a duration value to *retain*, the unit generator is set up to keep the last output samples for the duration, and these samples can be taken out in the form of a `Samples` object as shown on lines 11, 29, and 39. Figure 3.55 performs the same task, but with the *pread* method instead. The difference between these two examples is that *pread* can directly produce *Samples* of the given duration, even when *retain* is not set up¹. As a unit-generator in LC can have multiple numbers of outlets, both *read* and *pread* methods can specify the outlet to obtain the output samples from, as on line 39 of Figure 3.54 and on line 20 of Figure 3.55.

One of the benefits of such a design is that a user can create microsounds and then store it for the later use as seen in Figure 3.57. Even on-demand generation of microsounds is possible as in Figure 3.58. Such a strategy can reduce the required computational cost in microsound synthesis and is hard to realize in a unit-generator language.

Patch objects can also be used to produce `Samples` objects as well as unit-generator objects, as shown in Figure 3.56. As described in Table 3.14 and Table 3.15, the signatures of these methods are defined with the orders and the default values that can

¹However, as the *pread* method is not capable of creating `Samples` objects from multiple outlets in the current version of LC, it is necessary to use *retain/process/read* for this purpose; it is planned to modify the behaviour in the later version so that the *pread* method can also read from multiple outlets at once.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

make the best use of duck-typing. Figure 3.59 is a simple example of such duck-typing. Duck-typing can facilitate the collaboration between the two programming models, also when microsounds are used as input to a unit-generator or a patch as described in the next paragraph.

Giving a microsound as an input to a unit-generator or a patch. Some tasks are easier to achieve if a *Samples* object can be used as the input to a unit-generator or a patch (e.g., applying an envelope or reverberation to the entire sound output of microsound synthesis). Both LC's unit-generator and patch have the *write* method, which allows a user to route a *Samples* object to the inlets.

Figure 3.60 and Figure 3.61 are examples of writing *Samples* objects into the input of a unit-generator or a patch for the reverberation. As shown in these examples, just by using the *write* method, a *Samples* object can be treated as the input signal to a unit-generator (or a patch) and then one can obtain the processed samples by reading from the unit-generator (or the patch).

Figure 3.62 is another reverberation example. In this example, the patch that receives *Samples* objects to its input is already being played. As the patch is active and produces its output synchronizing with the advance of logical time, using the *write* method is enough to play the processed sound unlike the previous two examples, in which a user must explicitly process the output and send it to the sound output device.

Furthermore, as seen in Figure 3.59, thanks to duck-typing, a unit-generator can be easily replaced with a patch without any modification to the existing code in many cases. Figure 3.63 describes an example of waveset harmonic distortion, the output of which is first applied to a triangular envelope by a unit-generator, then to a triangular envelope with reverberation by a patch. As shown, the same function can be reused without any modification; thus, the bidirectional collaboration between the traditional unit-generator concept and LC's microsound synthesis framework can be performed without difficulty.


```
01: //instantiate a white noise generator
02: var src = new Noise~();
03:
04: //we want to keep the latest 1 second output.
05: src.retain = 1::second;
06:
07: //process for 1 second, without the advance of logical time.
08: src->process(1::second);
09:
10: //get the last 1 second output.
11: var out = src->read(1::second);
12:
13: //set it to the sound output.
14: PanOut(out);
15: now += 1::second;
16:
17: //instantiate a sine wave oscillator.
18: src = new Sin~(440);
19:
20: //we want to keep the last 50 msec output.
21: src.retain = 50::ms;
22:
23: //play a sine wave of 440 Hz.
24: within(5::second){
25:   while(true){
26:     //process for 50 msec, without the advance of logical time.
27:     //then read the last 50msec output and write to DAC.
28:     src->process(50::ms);
29:     var out = src->read(50::ms);
30:
31:     //set it to the sound output.
32:     PanOut(out);
33:     now += out.dur;
34:   }
35: }
36: //change the frequency every 50 msec.
37: while(true){
38:   src->process(50::ms);
39:   var out = src->read(50::ms, \out);
40:   PanOut(out);
41:   //update the frequency.
42:   src.freq = Rand(1,8) * 440;
43:   now += out.dur;
44: }
```

Figure 3.54: An example of creating Sample objects from the unit-generator’s output samples (1)

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
01: //using 'pread' instead
02: var src = new Noise~();
03: var out = src->pread(1::second);
04: PanOut(out);
05: now += 1::second;
06:
07: //instantiate a sine wave oscillator.
08: src = new Sin~(440);
09:
10: within(5::second){
11:   while(true){
12:     //generate a Samples object of 50ms duration and output.
13:     var out = src->pread(50::ms);
14:     PanOut(out);
15:     now += out.dur;
16:   }
17: }
18: //change the frequency every 50 msec.
19: while(true){
20:   var out = src->pread(50::ms, \out);
21:   PanOut(out);
22:   src.freq = Rand(1,8) * 440;
23:   now += out.dur;
24: }
```

Figure 3.55: An example of creating Sample objects from the unit-generator's output samples (2).

```
01: //create a patch.  the sine wave oscillator with tremolo/ring modulation.
02: var p = patch {
03:   amp:Sin~(25.0) {\out => \amp} defout:Sin~(440) => DAC~();
04: };
05:
06: //to retain the last 1 sec output from Sin~.
07: p.defout.retain = 1::second;
08:
09: //process the patch for 1 second without the advance of logical time.
10: p->process(1::second);
11:
12: //get the last 1 sec output and send it to the sound output.
13: var out = p->read(1::second);
14: PanOut(out);
15:
16: now += 1::second;
17:
18: within(5::second){
19:   while(true){
20:     //let's hear the phase output of p.amp
21:     out = p->pread(50::ms, \amp, \phase);
22:     PanOut(out);
23:     now += out.dur;
24:   }
25: }
26:
27: //changing the frequency every 50msec
28: while(true){
29:   out = p->pread(50::ms, \defout);
30:   p.defout.freq = Rand(1, 8) * 440;
31:   PanOut(out);
32:   now += out.dur;
33: }
```

Figure 3.56: An example to create Sample objects from the patch's output samples.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
01: //create an array to store pregenerated grains
02: var grains = new Array(100);
03:
04: //generate grains with 400Hz - 500Hz sine waves.
05: var win = GenWindow(512::samp, \hanning);
06: for (var i = 0; i < grains.size; i += 1){
07:   var src = new Sin~(i + 400);
08:   var tmp = src->pread(win.dur);
09:   var grn = tmp->applyEnv(win);
10:   grains[i] = grn;
11: }
12:
13: within(5::second){
14:   while(true){
15:     var idx = Rand(0, grains.size - 1);
16:     PanOut(grains[idx]);
17:     now += grains[idx].dur;
18:   }
19: }
```

Figure 3.57: A granular synthesis example with the pregenerated grains.

```
01: //an array to store grains.
02: var grains = new Array(100);
03: var win = GenWindow(512::samp, \hanning);
04: //perform granular synthesis for 5 second.
05: within(5::second){
06:   while(true){
07:     //pick up random index and check if a grain is already stored.
08:     var idx = Rand(0, grains.size - 1);
09:     var grn = grains[idx];
10:     //if no grain is stored at the index, generate one and store it.
11:     if (grn == null){
12:       var src = new Sin~(idx + 400);
13:       var tmp = src->pread(win.dur);
14:       grn = tmp->applyEnv(win);
15:       grains[idx] = grn;
16:     }
17:     //output the grain and sleep until the next scheduling timing.
18:     PanOut(grn);
19:     now += grn.dur / Rand(0.5, 2);
20:   }
21: }
```

Figure 3.58: A granular synthesis example with on-demand generation of the grains.

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
01:  //a simple function that plays given sound source 'src' for duration 'dur'
02:  var f = function(var src, dur)
03:  {
04:    within(dur){
05:      while(true){
06:        var out = src->pread(50::ms);
07:        PanOut(out, 0.0);
08:        now += out.dur;
09:      }
10:    }
11:  };
12:
13:  //src1 is a unit-generator while src2 is a patch.
14:  var src1 = new Phasor~(880);
15:  //set Freeverb~ as the default output of the patch ('defout').
16:  var src2 = patch {
17:    Sin~(440) => defout:Freeverb~(damp:1, fb1:0.58, fb2:0.2, spread:200)
18:    => DAC~();
19:  };
20:
21:  f(src1, 3::second);
22:  f(src2, 5::second);
```

Figure 3.59: A duck-typing example of 'pread'.

```
01: //loading the sound file onto the buffer.
02: LoadSndFile(0, "/sound/sample1.aif");
03:
04: //Freeverb is a reverbrator unit-generator
05: var reverb = new Freeverb~(damp:1, fb1:0.58, fb2:0.2, spread:200);
06:
07: //play the sound for 20 second
08:
09: var entireDur = 20::second;
10: within(entireDur){
11:
12:     //perform granular-sampling time-stretching with reverberation.
13:     var pos = 0::second;
14:     var dur = 512::samp;
15:     var win = GenWindow(512::samp, \hanning);
16:
17:     var overlap = 2;
18:     var adv = dur / overlap; //the amount of the time to advance in each iteration.
19:
20:     while(true){
21:         //generate a grain.
22:         var sample = ReadBuf(0, dur, offset:pos);
23:         var grain= sample->applyEnv(win);
24:
25:         //write it to the reverbrator's default input (defin).
26:         reverb->write(grain);
27:         //read the output samples from the reverbrator's default output.
28:         var out = reverb->pread(adv ); //read from 'defout' outlet.
29:         //send it out to the sound output.
30:         PanOut(out);
31:
32:         //update the reading position and advance the time.
33:         pos += dur / 4;
34:         now += adv;
35:     }
36: }
```

Figure 3.60: A reverberation example (1).

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
01: //loading the sound file onto the buffer.
02: LoadSndFile(0, "/sound/sample1.aif");
03:
04: //Freeverb is a reverbrator unit-generator.
05: var reverb = patch {
06:   defin: Freeverb(damp:0.2, fb1:0.58, fb2:0.2, spread:200) => defout: DAC();
07: };
08:
09: //play the sound for 20 seconds.
10: var entireDur = 20::second;
11: within(entireDur){
12:   //perform granular-sampling time-stretching.
13:   var pos = 0::second;
14:   var dur = 512::samp;
15:   var win = GenWindow(512::samp, \hanning);
16:
17:   var overlap = 2;
18:   var advance = dur / overlap;
19:
20:   while(true){
21:
22:     var sample = ReadBuf(0, dur, offset:pos);
23:     var grain = sample->applyEnv(win);
24:     //write it to the reverbrator's default input (defin).
25:     reverb->write(grain);
26:     var out = reverb->pread(advance); //read from 'defout' outlet.
27:     WriteDAC(out);
28:
29:     pos += dur / 4;
30:     now += advance;
31:   }
32: }
```

Figure 3.61: A reverberation example (2).


```
01: //loading the sound file onto the buffer.
02: LoadSndFile(0, "/sound/sample1.aif");
03: //Freeverb is a reverbrator unit-generator.
04: var reverb = patch {
05:   reverb:Freeverb~(damp:0.2, fb1:0.58, fb2:0.2, spread:200) => DAC~();
06: };
07: //start this patch immediately. keep on playing regardless of below loop.
08: reverb->start();
09:
10: //play the sound for 20 seconds.
11: var entireDur = 20::second;
12: within(entireDur){
13:   //perform granular-sampling time-stretching.
14:   var pos = 0::second;
15:   var dur = 512::samp;
16:   var win = GenWindow(512::samp, \hanning);
17:
18:   while(true){
19:     var sample = ReadBuf(0, dur, offset:pos);
20:     var grain = sample->applyEnv(win);
21:     //this time we only have to write to the patch. As the patch is being played
22:     //by the system, it automatically sends the samples to DAC (see line 08).
23:     reverb->write(grain, \reverb);
24:
25:     pos += dur / 4;
26:     now += dur / 2;
27:   }
28: }
```

Figure 3.62: A reverberation example (3).

3. DESIGN: LC, A MOSTLY-STRONGLY-TIMED PROTOTYPE-BASED COMPUTER MUSIC PROGRAMMING LANGUAGE THAT INTEGRATES OBJECTS AND MANIPULATIONS FOR MICRO SOUND SYNTHESIS

```
01: //load a sound file and extract wavesets.
02: LoadSndFile(0, "/sound/sample1.aif");
03: var wavesets = ExtractWavesets(0);
04: //below function performs waveset harmonic distortion.
05: //and write out the output to the given 'stream'
06: var f = function(var stream, entireDur){
07:   var weight1 = 0.5;
08:   var weight2 = 0.5;
09:   //perform waveset harmonic distortion.
10:   within(entireDur){
11:     for (var i = 0; i < wavesets.size; i+= 1){
12:       var ws = wavesets[i];
13:       var harm1 = ws->resample(ws.size / 2)->amplify(weight1);
14:       var harm2 = ws->resample(ws.size / 3)->amplify(weight2);
15:
16:       stream->write(ws);
17:       stream->write(harm1);
18:       stream->write(harm1, offset:harm1.dur);
19:       stream->write(harm2);
20:       stream->write(harm2, offset:harm2.dur);
21:       stream->write(harm2, offset:harm2.dur * 2);
22:
23:       var out = stream->pread(ws.dur);
24:       PanOut(out);
25:       now += ws.dur;
26:     }
27:   }
28: };
29: //apply an triangular envelope.
30: var entireDur = 2::second;
31: var envelope = new Line~([\reset, 0, 1.0, entireDur / 2, 0.0, entireDur / 2]);
32: envelope->trigger();
33: //pass it to the waveset harmonic distortion function.
34: f(envelope, entireDur);
35: //apply an triangular envelope + reverberation.
36: var p = patch {
37:   defin:Line~([\reset, 0, 1.0, entireDur / 2, 0.0, entireDur / 2]) =>
38:   Freeverb~(damp:0.2, fb1:0.58, fb2:0.2, spread:200) => defout:DAC~();
39: };
40: p.defin->trigger();
41: f(p, entireDur);
```

Figure 3.63: A duck-typing example to apply an envelope (by a unit-generator) and an envelope + reverberation (by a patch) to the output of waveset harmonic distortion.

4

Discussion: the Necessity for the Development of LC as a New Language and the Benefits of Its Language Design

This chapter first describes why it was necessary to develop LC as an entirely new language with its own compiler and virtual machine, not just as a sound synthesis framework or internal DSL built upon some other language. Then, the benefits of LC's language design are discussed by comparing LC to the existing computer music languages from the perspective of the three issues discussed in Chapter 2: (1) the insufficient support for dynamic modification of a computer music program, (2) the insufficient support for precise timing behaviour and other features with respect to time, and (3) the difficulty in microsound synthesis programming caused by the anti-pattern of abstraction inversion. The last section summarises the discussion in this chapter to clarify the contribution of this thesis.

4.1 The justification of the development of LC as a new computer music programming language

As described in previous chapters, LC is developed as a computer music programming language with its own syntax, compiler and virtual machine, and does not depend on

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

any other general-purpose programming language or domain-specific language. Some may suggest designing a language with the same features as an internal DSL or using software libraries for some general purpose programming language, rather than developing one from scratch.

However, there are several reasons that require LC to be developed as a new computer music language: (1) LC's design intends to provide more suitable syntaxes for frequently performed tasks (e.g., patch-creation and message-communication), (2) the nested execution time constraints must support a different behaviour than the normal exception-handling mechanism when nested, and LC also requires performing execution time constraints with logical synchronous time, not real-time, (3) mostly-strongly-timed programming cannot be implemented as library functions; thus, it was necessary to develop LC as a new computer music programming languages with its own syntax, compiler, and virtual machine.

4.1.1 The necessity to provide more suitable syntaxes for frequently performed tasks

Some features of LC may be simply implemented as library functions or software frameworks. For instance, the 'Patch' object (Figure 4.1) and the message-passing feature (Figure 4.2) in LC can be implemented as library functions. However, since these features are often used by users, it is better to provide more suitable syntaxes for programming and comprehension. LC provides syntax sugars for such features as shown in Figure 4.3 and Figure 4.4.

To provide such syntaxes, a programming environment must translate the code to one that the base language can accept and requires support at the level of micro definition or pre-compiler; then, the language is already considered an *internal DSL* at least.

4.1.2 Execution time constraints

While execution time constraint is one of the most essential features that real-time programming languages must support, many general-purpose programming languages lack this feature and require some extension to the language specifications. The main reason for this is that the execution time constraint is an *asynchronous transfer of control* (ATC) and the behaviour of the nested constraints differ significantly from

4.1 The justification of the development of LC as a new computer music programming language

A simple sine wave oscillator example

```
01: //create a Patch object.
02: var p = new Patch();
03: //store unit-generator objects to its slots.
04: p.src = new Sin~(440); //u-gen names always start with a tilde.
05: p.dac = new DAC~();
06: //connect the default output of a sine wave osc to DAC's default input.
07: p->connect(\src, \defout, \dac, \defin);
08: //update the unit-generator graph.
09: p->compile();
10: //start playing the patch immediately.
11: p->start();
12:
13: //wait for 1 second and change the frequency.
14: now += 1::second;
15: p.src.freq = 880;
16:
17: //wait for 0.5 second and disconnect sin and dac.
18: now += 0.5::second;
19: p->disconnect(\src, \defout, \dac, \defin);
20: //it is necessary to update the unit-generator graph again.
21: p->compile();
22:
23: //wait for 0.5 second again, connect to DAC's right channel(ch1).
24: now += 0.5::second;
25: p->connect(\src, \defout, \dac, \ch1);
26: p->compile();
27: //wait for 1 second again, deactivate the patch.
28: now += 1::second;
29: p->stop();
30:
31: //wait for 0.5 seconds, activate the patch again.
32: now += 0.5::second;
33: p->start();
34:
35: //swap a sine wave osc with a phasor.
36: var tmp = p.src; //store a sinewave osc to tmp.
37: p.src = new Phasor~(440);
38: p->compile();
39: //restore a sinewave osc after 1 sec.
40: now += 1::second;
41: p.src = tmp;
42: p->compile();
```

Figure 4.1: A simple sine wave oscillator example in LC (reproduced from Figure 3.17).

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

A timed interthread messaging example in LC(1)

```
01: //create a function object.
02: var f = function(){
03:     var thread = GetCurrentThread();
04:     while(true){
05:         //receive a message in the blocking mode.
06:         var message = thread->recv(\blocking);
07:         if (message == \quit){
08:             break;
09:         }
10:         println("received :" .. message);
11:     }
12:
13:     println("bye!");
14:     return;
15: };
16:
17: //starting the function in a new thread
18: var thread = f@();
19: thread->start();
20:
21: //queue a message to the thread's mailbox
22: thread->queueMessage("Hello!");
23: now += 1::second;
24:
25: //since the return value of queueMessage() is the thread itself,
26: //it can be cascaded as below.
27: //it is also possible to specify the message delivery timing.
28: thread->queueMessage("Sending the first message")
29:     ->queueMessage("This should be received after 1 sec", now + 1::second)
30:     ->queueMessage("This should be received after 2 sec", now + 2::second);
31:
32: if duration value is used as timing, it is interpreted as 'now + duration'.
33: thread->queueMessage(\quit, 3::second);
```

The output from the above example

```
received :Hello!
received :Sending the first message
received :This should be received after 1 sec
received :This should be received after 2 sec
bye!
```

Figure 4.2: A timed interthread messaging example in LC(1) (reproduced from Figure 3.24).

4.1 The justification of the development of LC as a new computer music programming language

Another simple sine wave oscillator example

```
01: //create a Patch object.
02: //patch { ... } expression is a syntax sugar to create a new patch.
03: var p = patch {
04:   //=> operator builds a connection.
05:   //if no inlet/outlet is specified it connects defout to defin.
06:   src:Sin~(440) => dac:DAC~();
07: };
08: p->start();
09: now += 1::second;
10: p.src.freq = 880;
11:
12: now += 0.5::second;
13: //update_patch { ... } expression is a syntax sugar to update a patch.
14: update_patch {
15:   //|= operator removes an existing connection.
16:   //when no inlet/outlet is specified, it disconnects defout and defin.
17:   src =| dac;
18: };
19:
20: update_patch(p){
21:   //=> operator can also specify inlet/outlet explicitly.
22:   src { \out => \ch1 } dac;
23: };
24: now += 1::second;
25: p->stop();
26:
27: now += 0.5::second;
28: p->start();
29:
30: var tmp = p.src;
31: update_patch(p){
32:   src:Phasor~(440);
33: };
34: now += 1::second;
35: update_patch(p){
36:   src:tmp;
37: };
```

Figure 4.3: Another sine wave oscillator example in LC (reproduced from 3.18).

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

A timed interthread messaging example in LC

```
01: //create a function object.
02: var f = function(){
03:   exit:
04:   while(true){
05:     receive(var message){
06:       case \quit:
07:         break exit;
08:       default:
09:         println("received :" .. message);
10:     }
11:   }
12:   println("bye!");
13:   return;
14: };
15:
16: var thread = f@();
17: thread->start();
18:
19: thread <- "Hello!";
20: now += 1::second;
21:
22: thread <- "Sending the first message"
23:     <- @1::second, "This should be received after 1 sec"
24:     <- @2::second, "This should be received after 2 sec";
25:
26: thread <- @3::second, \quit;
```

Figure 4.4: A timed interthread messaging example in LC.

4.1 The justification of the development of LC as a new computer music programming language

normal exception handling, as “nested ATC’s (for example, for timeouts) must work properly. A timeout from an outer timer must be handled in the outer scope, even if the control is in the scope of an inner (longer) timer” (60).

Thus, to support the nested execution time constraints, a language must provide the correct behaviour in handling asynchronous transfer of control. However, many general-purpose programming languages do not provide such a feature. Moreover, since LC is strongly-timed and executes interactive programs in internal logical synchronous time, existing real-time programming languages are not particularly beneficial for LC to be built upon the languages, as they provide such features with respect to time in *real time*; supporting execution-time constraints in logical synchronous time with sample-rate accuracy is another significant reason for development of a new language from scratch.

4.1.3 Mostly-strongly-timed programming cannot be implemented as library functions

The mostly-strongly-timed programming concept is one of the core features of LC, and this feature cannot be implemented as just a library, because it requires the support from the underlying task scheduler at least at the virtual machine level. As described in Chapter 2.2, synchronizing the timing behaviour with sample-rate accuracy in multithreading with native threads can be almost impossible under many operating systems. Instead, many computer music languages implement the feature of coroutines (as in LuaAV) or software threads (as in ChuckK) to achieve sample-rate accurate synchronization in logical synchronous time.

Mostly-strongly-timed programming extends such synchronous behaviour with explicit switching between synchronous/non-preemptive context and asynchronous/preemptive context **within the same thread**. While many languages support both collaborative/non-preemptive multithreading by coroutines and preemptive multithreading by native or software threads, these are independent from each other and the context switching between two different models as seen in mostly-strongly-timed programming is not supported in the existing programming languages; such context switching must be dealt with within the underlying task scheduler, and the runtime environment may require a significant degree of modification in its implementation. when implementing

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

the mostly-strongly-timed programming concept in an existing programming language.

Moreover, even when the underlying scheduler is modified to support context switching between synchronous/non-preemptive context and asynchronous/preemptive context, the mostly-strongly-timed programming cannot be supported in the form of library function calls. Otherwise, mostly-strongly-timed programs cannot provide the precise timing behaviour with sample-rate accuracy, when an execution time constraint causes time-out or an exception is thrown at runtime.

For example, see the Figure 4.5 example. It may appear that this example can be translated to the Figure 4.6 example, which involves library function calls for context switching, in the pre-compilation phase. However, in the presence of an execution time constraints or exception-handling as in these two examples, the behaviour of the translated program cannot be equivalent.

In Figure 4.5, the statement on line 08 violates the execution time constraints given on line 04. This results in the code jumping to the ‘timeout’ statement, recovering the original context. Thus, in this example, the thread’s context switches back to the synchronous context when the time-out occurs. However, in the Figure 4.6 example, which uses the library functions for context switching, it exhibits a different behaviour. When the time-out occurs on line 14, the code jumps to the line 18, where the timeout block starts, but since the switch back to the synchronous context is not performed until line 24, the underlying scheduler may suspend the thread and advance the logical synchronous time. Hence, while the original code can guarantee that the logical time at the beginning of ‘timeout’ block is not advanced right after the time-out occurred, the translated code, which uses the library function call for context switching, may advance logical synchronous time; thus, the context switching needs to be handled at the virtual machine level and can’t be replaced by the library function call.

4.1.4 The necessity for LC’s own compiler and virtual machine

As described in this section, to provide more appropriate syntaxes for certain features, it is necessary to use macro definitions or a precompiler, if LC needs to be implemented upon an existing programming language. To provide execution time constraints, the base languages must be equipped with the nested execution time constraints properly, as its behaviour differs from the normal exception-handling mechanism. Moreover, context

4.1 The justification of the development of LC as a new computer music programming language

```
01: //switch to sync context
02: sync {
03:     //give an execution time constraint of 5 sec.
04:     within(5::second){
05:         //switch to async context.
06:         async{
07:             //intentionally causing the time out.
08:             now += 10::second;
09:         }
10:     }
11:     //the whole 'timeout' block below must be executed
12:     //in the synchronous/non-preemptive context.
13:     //in other words, the whole timeout block below must be
14:     //executed right when the execution time constraint is violated
15:     //without any progress of logical synchronous time.
16:     timeout{
17:         //the code must reach here exactly when the above
18:         //execution time constraint is violated.
19:         println("timeout!");
20:     }
21: }
```

Figure 4.5: An example of context switching between synchronous/non-preemptive context and asynchronous/preemptive context with an execution time constraint.

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

```
01: var prevCtx1;
02: {
03:     //store the current sync context to recover it later.
04:     prevCtx1 = getCurrentSyncContext();
05:     //then switch to the sync context.
06:     switchToSyncContext();
07:     {
08:         var prevCtx2;
09:         within(5::second){
10:             //save the previous context before switching to the async context.
11:             prevCtx2 = getCurrentSyncContext();
12:             //now switch to the async context.
13:             switchToAsyncContext();
14:             now += 10::second;
15:             //recover the previous context when no timeout occurs
16:             setCurrentSyncContext(prevCtx2);
17:         }
18:         timeout {
19:             //before processing 'timeout' block, the previous context must be
20:             //recovered. however, the underlying scheduler may suspend this thread
21:             //before the below function call to recover the previous 'sync' context,
22:             //as the thread is still in the 'async' context.
23:             //thus, the advance of logical synchronous time can occur here.
24:             setCurrentSyncContext(prevCtx2);
25:             println("timeout!");
26:         }
27:     }
28: }
29: //recover the original context.
30: setCurrentSyncContext(prevCtx1);
```

Figure 4.6: An example of context switching by library function calls between the synchronous/non-preemptive context and the asynchronous/preemptive context with an execution time constraint.

4.2 Comparing LC with the existing computer music languages

switching between synchronous and asynchronous contexts in the mostly-strongly-timed programming concept cannot be correctly implemented by the use of library functions and must be handled at the level of the virtual machine or runtime environment.

Therefore, it is more appropriate to develop LC as an independent programming language with its own compiler and virtual machine than as an internal DSL or software library/framework for an existing programming language; even in the latter case, there would be a substantial amount of software development involved at many different levels of language implementation. To make matters worse, it would be necessary to extend the language specification of the original language.

4.2 Comparing LC with the existing computer music languages

In this section, the comparison between LC and existing computer music languages is described to evaluate the benefits of LC's language design in the three issues addressed as design opportunities for a new computer music language: the insufficient support for dynamic modification of a computer music system in run-time, the lack of precise timing behaviour with other features with respect to time, and the difficulty in microsound synthesis programming.

The following subsections provides descriptions related to these three issues in existing computer music languages, such as SuperCollider (210) (320), ChucK (312)(314), Impromptu (61)(274)(276), Max (234)(327), PureData (233), and similar languages, to clarify the benefits of LC's language design regarding the three issues.

4.2.1 The support for dynamic modification of a computer music system at runtime

As described in Section 2.1, recent computer music practices, such as live-coding and dynamic-patching, often require a significant degree of dynamism at runtime at both levels of compositional algorithms and sound synthesis, and such dynamism in programming is also considered beneficial for rapid-prototyping of a computer music system.

While it has become an the important design criterion to integrate sound synthesis and compositional algorithms seamlessly into one environment around the late 90s as seen in Nyquist (89)(90), SuperCollider, or Max (231)(327), even recent computer music

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

languages have certain drawbacks in their language designs in order to fully support the dynamic modification at both levels of compositional algorithms and sound synthesis, especially when we consider interactive music applications¹; this is especially true when considering whether a certain computer music language provides a terse and consistent programming model that can be applied to these two levels. The existing languages often provide such a dynamic feature as an extension as discussed below.

4.2.1.1 Dynamic modification of a computer music system in the existing computer music languages

SuperCollider. As SuperCollider was designed under a large influence (211) from Smalltalk (126), which is a dynamic object-oriented programming language, it can support the dynamic modification of a computer music system at the compositional algorithm level. One can easily mimic prototype-based programming with *Dictionary* or *Event* (an associative collection that stores key-value pairs) and first class functions in SuperCollider, as shown in Figure 4.7. The extension called *chucklib* is also available to facilitate prototype-based programming in SuperCollider (320, Chapter 20) (Figure 4.8). As shown in these examples, SuperCollider can support a certain degree of dynamic modification in runtime by prototype-based programming at the level of compositional algorithms.

However, largely due to SuperCollider’s software architecture that divides its runtime system into *scsynth*, which is a sound synthesis server, and *sclang*, which is an interpreter where compositional algorithms are performed, the programming models for dynamic modification significantly differ between the compositional algorithms and the sound synthesis level; SuperCollider extended its original language specification with *Just-in-Time programming* (247) (320, Chapter 7) for dynamic modification at the sound synthesis level.

Generally speaking, one must utilize a *Synth* object so that sound synthesis can be performed on *scserver*. Even when one uses such expression as ‘{ SinOsc.ar(440) }.play’ in SuperCollider, it is using the *play* method of a function object (a block ‘{ ... }’ in SuperCollider is a function object), which creates and plays a *Synth* object. Figure

¹As Nyquist is not designed for interactive music applications, this subsection does not further discuss Nyquist.

4.2 Comparing LC with the existing computer music languages

```
01: //using Dictionary for prototyped-based programming
02: d = Dictionary.new;
03: d[\say] = {
04:   arg self, message;
05:   ("message :" + message).postln;
06: };
07:
08: d[\performAddition] = {
09:   arg self, a,b;
10:   a + b;
11: };
12:
13: d[\say].value(d, "Hello, world");
14: d[\performAddition].value(d, 1,2);
15:
16: //using Event instead.
17: e = (); //create an Event object.
18:
19: e.say = {
20:   arg self, message;
21:   ("message :" + message).postln;
22: };
23: e.performAddition = {
24:   arg self, a, b;
25:   a + b;
26: };
27: //An Event object implicitly passes the reference to itself as a first argument.
28: e.say("Hello, world");
29: e.performAddition(1,2);
```

Figure 4.7: A prototype-based programming example by Dictionary and Event in Super-Collider.

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

```
01: (
02:   ~greeter = Proto({
03:     ~sayhi = { |name|
04:       "%, %\n".postf(~greeting.value, name ? ~name);
05:     };
06:     ~name = "Monique"; //default name
07:     ~greeting = "Hello"; //default greeting
08:   });
09:
10:   ~frenchGreeter = ~greeter.clone({
11:     ~greeting = "Bonjour";
12:   });
13:
14:   ~timeAwareFrenchGreeter = ~frenchGreeter.clone({
15:     ~greeting = {
16:       var hour = Date.getDate.hour;
17:       if (hour < 18) { ~dayGreeting } { ~eveningGreeting };
18:     };
19:     ~dayGreeting = "Bonjour";
20:     ~eveningGreeting = "Bon soir";
21:   });
22: )
23:
24: ~greeter.sayhi;
25: ~greeter.sayhi("Bob");
26: ~frenchGreeter.sayhi("Isabelle");
27: ~timeAwareFrenchGreeter.sayhi("Eric");
```

Figure 4.8: A prototype-based programming example with chuchklib in SuperCollider (320, p.600).

4.2 Comparing LC with the existing computer music languages

4.9 describes how to play a Synth object in SuperCollider.

As mentioned above, the dynamic modification of a unit-generator graph must involve *Just-in-Time programming* extensions in SuperCollider to the original language. Figure 4.10 describes several examples of Just-in-Time programming. Figure 4.10 (above) describes a simple example, which does not involve *Just-in-Time programming*. The code between line 04-08 repeatedly send an message to play a note number x for the duration 0.125 second to *scserver*. As the whole program is executed on *sclang*, the update of the variables (x and y) on line 13 and 14 can be reflected to the messages sent to *scserver*.

However, since the dynamic modification of a synthesis graph must involve the sound synthesis server *scserver*, which is a different program than the interpreter *sclang*, the code cannot be as simple as the above example any more. The Figure 4.10 example (middle) describes a wrong way of dynamic modification of a sound synthesis graph, which is ineffective. To make such a dynamic modification of a synthesis graph in SuperCollider, one needs to use *proxies* as in Figure 4.10 (bottom). While this example implicitly creates *proxy objects*, it is also possible to explicitly create a proxy object. Figure 4.11 (above) describes an example of such explicit creation and use of a proxy object.

As shown, a proxy object play a role of *place holder* for the other objects and it is required to involve *NodeProxy* object for objects on the *scserver* side, explicitly or implicitly. As seen in Figure 4.10 (bottom), the use of *ProxySpace* simplifies the creation of *NodeProxy* objects in the current environment. However, it also makes it unclear if the variable contains the reference to a server side object as a *NodeProxy*, or if it contains a value on the *sclang* side. This can make it harder for users to comprehend what the program does, especially when one must take care in distinguishing if an object is on the side of *scserver* or within the *sclang* interpreter.

In such a programming model, since the modification of a synthesis graph in Just-in-Time programming is possible only at the point where proxy objects are used, one must use proxy objects before which part of a synthesis graph must be dynamically modified before writing the code, otherwise it can require a significant amount of refactoring in runtime as shown in Figure 4.12. Furthermore, many proxy objects can be involved for more fine-grained modification, as shown in Figure 4.13.

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

```
01: //the simplest way to perform synthesis.
02: //the code below implicitly instantiates a BinaryOpUGen that holds
03: //the references to two SinOsc unit-generators and adds the input from them.
04: //as {...} in SuperCollider is indeed a function object and calling 'play' method
05: //of a function object will creates an Synth object and then send it to server
06: //and then performs sound synthesis immediately.
07: { SinOsc.ar(440) + SinOsc.ar(880) }.play;
08:
09: //store an instrument definition named \inst1 to the server.
10: //this makes the reuse of the definition of Synth objects a lot easier.
11: (
12:   SynthDef(\inst1, {
13:     arg freq = 440;
14:     var a = SinOsc.ar(freq) + SinOsc.ar(freq * 2);
15:     Out.ar(0, a);
16:   }
17: ).add;
18:
19: //play the instrument \inst1.
20: Synth(\inst1);
21: Synth(\inst1, [\freq, 880]);
```

Figure 4.9: Playing Synth objects in SuperCollider.

As above, SuperCollider adopts different programming models for the dynamic modification at the levels of compositional algorithms and sound synthesis and the dynamic modification at the sound synthesis level can often exhibit such problems as *premature commitment*¹ and *viscosity*². These problems can be attributed not just to SuperCollider’s language design and programming models, but also to its software architecture that separate the sound synthesis software from the language interpreter.

ChuckK. As ChuckK is a statically-typed class-based language, dynamic modification at the compositional algorithm level can be a lot constrained by types and class-

¹Green and Blackwell describes the problems of *premature commitment* “arise when the target notation contains many internal constraints or dependencies and when the order constraints force the user to make a decision before full information is available (premature commitment) or to look ahead in a way that is cognitively expensive (enforced lookahead)” (129).

²Green and Blackwell defines *viscosity* as “resistance to change: the cost of making small changes” and viscosity “becomes a problem in opportunistic planning when the user/planner changes the plan” (129).

4.2 Comparing LC with the existing computer music languages

a modulo algorithm that operates over states of variables (from (320, p.208))

```
01: (  
02:   Task {  
03:     x = 4; y = 13;  
04:     loop {  
05:       x = (x * y) % 5;  
06:       (note: x.postln, dur: 0.125).play;  
07:       0.125.wait;  
08:     };  
09:   }.play;  
10: ); // creates a loop of values;  
11:  
12: // change x and y;  
13: x = 5; // new initial value  
14: y = 4; // new multiplication factor
```

a wrong way to modify a synthesis graph (from (320, p.209))

```
01: (  
02: {  
03:   x = SinOsc.kr(4);  
04:   y = SinOsc.kr(13);  
05:   SinOsc.ar(x * y % 0.4 * 500 + 600) * 0.2  
06: }.play;  
07: )  
08:  
09: //change x and y?  
10: x = SinOsc.kr(4); // no effect.  
11: y = SinOsc.kr(4); // no effect either.
```

dynamic synthesis graph (from (320, p.210))

```
01: p = ProxySpace.push;  
02: ~x = { SinOsc.kr(4) };  
03: ~y = { SinOsc.kr(13) };  
04: ~z = { SinOsc.ar(~x * ~y % 0.4 * 500 + 600) * 0.2 };  
06: ~z.play;  
07:  
08: // now ~x and ~y can be replaced  
09: ~x = { SinOsc.kr(0.4) };  
10: ~y = { SinOsc.kr(1.3) };  
11:  
12: p.clear(2).pop; // release environment (2 sec fadeout)
```

Figure 4.10: Just-in-Time programming example in SuperCollider (320, pp.208-210).

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

```
01: n = NodeProxy.new;
02: x = { SinOsc.ar(n.kr * 200 + 300) * 0.1 }.play;
03: n.source = { LFPulse.kr([1.3,2.1,3.2]).sum };
04: n.clear; x.free;
```

Figure 4.11: Creating a proxy object explicitly and changing its source (from (320, p.215)).

```
01: p = ProxySpace.push; // if needed
02:
03: ~a = Lag.ar(LFClipNoise.ar(2 ! 2, 0.5, 0.5), 0.2) ;
04: (
05: ~b = {
06:   var c, d;
07:   c = Dust.ar(20 ! 2);
08:   d = Decay2.ar(c, 0.01, 0.02, SinOsc.ar(11300));
09:   d + BPF.ar(c * 5, ~a.ar * 3000 + 1000, 0.1)
10: }
11: );
12:
13: ~b.play;
14:
15: // the refactored code from above
16:
17: (
18: ~a = {
19:   var a;
20:   a = Lag.ar(LFClipNoise.ar(2 ! 2, 0.5, 0.5), 0.2);
21:   BPF.ar(~c.ar * 5, a * 3000 + 1000, 0.1)
22: }
23: );
24: ~c = { Dust.ar(20 ! 2) };
25: ~d = { Decay2.ar(~c.ar, 0.01, 0.02, SinOsc.ar(11300) )};
26: ~b = ~a + ~b;
27:
28: ~b.play;
```

Figure 4.12: Refactoring a synthesis graph at runtime (from (320, p.212)).

4.2 Comparing LC with the existing computer music languages

```
01: ~out.play; ~out.fadeTime = 3;
02: (
03: // name with a_ represents audio rate argument
04: ~out = { | freq = 440, mod=0.4, detune=0.1, a_in = #[1,1] |
05:   freq = freq * ([0, detune] + 1);
06:   LFTri.ar(LFTri.ar(mod * freq).range(freq * mod, freq)) * a_in * 0.2
07: }
08: );
09:
10: (
11: ~mod2 = { LFNoise1.kr(1).range(0,1) };
12: ~mod1 = { LFPulse.kr(~mod2.kr * 30 + 1, 0, 0.3) };
13: ~freq1 = { ~mod1.kr * 13100 + 100 };
14: ~freq2 = { LFTri.kr(30) * 200 + 300 };
15: ~audio1 = { BrownNoise.ar(LFClipNoise.kr(10. dup), 1) };
16: ~audio2 = { SinOsc.ar(LFNoise2.kr(1.dup).exprange(4, 1000)) };
17: );
18:
19: ~out.map(\freq, ~freq2, \mod, ~mod1);
20: ~out.set(\detune, 0.01);
21: ~out.map(\freq, ~freq1, \mod, ~mod1);
22: //xmap crossfades over fade time to new value.
23: ~out.xmap(\freq, ~freq1, \mod, ~mod2);
24: ~out.xmap(\freq, ~freq2, \mod, ~mod1, \a_in, ~audio2);
25: ~out.map(\a_in, ~audio1);
```

Figure 4.13: Parameter mapping and setting (from (320, p.216)).

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

hierarchy. While ChuckK allows overloading of functions, the current version¹ simply neglects the redefinition of a function. Chuck also lacks other desirable features, such as first class functions and duck-typing. This makes a ChuckK program a lot less flexible against dynamic modification in comparison with other computer music languages with dynamic features. On the other hand, at the sound synthesis level, Chuck's ' $=>$ ' operator (for connection) and ' $=<$ ' (for disconnection) makes it significantly easier to modify a unit-generator graph dynamically. Figure 4.14 describes an example of the dynamic modification of the unit-generator graph in ChuckK.

However, unlike the dynamic modification of the connections in a synthesis graph, ChuckK has a significant obstacle in replacing a unit-generator with another unit-generator between the different types in runtime. Figure 4.15 example describes such an issue. Two problems can be observed. The first problem is that as ChuckK is a class-based programming language, the replacement of a unit-generator with another unit-generator can involve an issue of typing as seen in Figure 4.15 (1). While one can use the reference type of the parent class instead in Figure 4.15 (2), as the connection between unit-generators is built **between instances of the unit-generators**, just assigning a new unit-generator to some variable does not cause the modification of a synthesis graph. To make an effective change in a synthesis graph, as seen in Figure 4.15 (3), one must disconnect the old instance from the synthesis graph first, and then connect the new instance. If the old instance was connected to many different unit-generators, one must rebuild all the existing connections in such a manner. Moreover, as the reference type of the parent class is involved in typing, the methods and attributes in the child class cannot be directly accessed (see line 15 in Figure 4.15 (3) example).

Thus, generally speaking, ChuckK has a drawback in dynamic modification at both levels of compositional algorithms and sound synthesis and it is due to as it is its nature as a statically-typed class-based language. Although the lack of the notion of 'subpatch' may make it harder to build and modify more complicated synthesis graphs in ChuckK.

Furthermore, the current implementation of ChuckK hardly allows interaction with a program that is already being executed. One cannot modify any part of the code or directly change parameters from outside of a program once after it is launched, while many other recent computer music languages can perform such interaction with

¹version 1.2.1.4-beta-1 (dracula).

4.2 Comparing LC with the existing computer music languages

```
01: //connect a sine wave oscillator to a gain controller and DAC.
02: SinOsc sin => Gain gain => dac;
03:
04: //set the frequency of 's' to 440Hz.
05: 440 => sin.freq;
06:
07: //when '3' is set to 'op' field of a unit-generator,
08: //the unit-generator will multiply all inputs.
09: 3 => gain.op;
10:
11: //connect another sine wave oscillator to the gain controller
12: //and set the frequency to 5Hz for tremolo effect.
13: SinOsc amp => gain;
14: 5 => amp.freq;
15:
16: //play the sound for 3 seconds.
17: 3::second +=> now;
18: <<< "3 sec passed. no tremolo effect from now" >>>;
19:
20: //kill the tremolo effect by disconnecting 'amp' from 'gain'.
21: amp =< gain;
22:
23: 3::second +=> now;
```

Figure 4.14: A simple example to connect/disconnect the connections in a synthesis graph in ChuckK.

the current environment. This can significantly limit the dynamic modification at any level.

Other textual computer music languages. As Impromptu (276) is an internal domain-specific language built on Scheme (278), which is very dynamic in its nature as one of the LISP family languages, it offers a substantial flexibility against dynamic modification at the level of compositional algorithm. Withal, as its sound synthesis functionality fully depends on Apple's Audio Unit framework, Impromptu's flexibility at the level of sound synthesis is constrained by the underlying Audio Unit framework; the Audio Unit framework provides a similar programming model to the unit-generator languages in that it builds the graph of the sound synthesis modules to perform the entire sound synthesis, yet the granularity of sound synthesis algorithms that can be

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

A typing issue in dynamic modification in ChuckK (1)

```
01: //‘@’ denotes the reference type.
02: //so ‘SinOsc@’ is a reference to ‘SinOsc’
03: new SinOsc @=> SinOsc@ osc;
04: osc => dac;
05: 3::second +=> now;
06: //the statement below results in a type error,
07: //as one cannot assign Phasor@ to SinOsc@.
08: new Phasor @=> osc;
09: 3::second +=> now;
```

A typing issue in dynamic modification in ChuckK (2)

```
01: //‘@’ denotes the reference type.
02: //use ‘UGen@’ instead, as UGen is a parent class of all ugens.
03: new SinOsc @=> UGen@ osc;
04: osc => dac;
05: 3::second +=> now;
06: //the statement below do not result in a type error,
07: //however, still ineffective, since ChuckK builds the connection
08: //between the instances of the unit-generators*
09: //so simply assigning a new unit-generator to ‘osc’ doesn’t
10: //cause the update of the unit-generator.
11: new Phasor @=> osc;
12: 3::second +=> now;
```

A typing issue in dynamic modification in ChuckK (3)

```
01: //‘@’ denotes the reference type.
02: //use ‘UGen@’ instead, as UGen is a parent class of all ugens.
03: new SinOsc @=> UGen@ osc;
04: osc => dac;
05: 3::second +=> now;
06: //first we disconnect the old connection.
07: osc =< dac;
08: //then instantiate a new object and reconnect.
09: new Phasor @=> osc;
10: osc => dac;
11: 3::second +=> now;
12:
13: //however the statement below results in a syntax error,
14: //as UGen class doesn’t have ‘freq’.
15: 880 => osc.freq;
```

Figure 4.15: A typing issue in dynamic modification at the sound synthesis level in ChuckK.

4.2 Comparing LC with the existing computer music languages

described differs a lot.

While the unit-generator languages provides very simple sound synthesis modules such as sine wave oscillators and filters as components to build sound objects (or note-level objects), the Audio Unit framework provides higher-level components as AU Instrument, a software synthesizer modules that “take MIDI and soundbank data as input and provide audio data as output — letting a user play a virtual instrument” (19), and AU Effect, software modules that apply various sound effects to the given input. Thus, while it still allows dynamic modification between these AU components, the exploration of the sound synthesis algorithms can be significantly limited compared to the unit-generator languages.

Extempore (277), the successor language of Impromptu recently under development, has a language design that takes Impromptu’s limitation into account and it can describe sound synthesis algorithms at two levels. Extempore allows a user to write the DSP function called back from the underlying sound driver, in which a user can directly describe the algorithms to compute the output samples while it is also possible to write virtual instruments that plays notes together with audio effects. The code at both levels can be dynamically compiled into native machine code at runtime by LLVM’s just-in-time compiler (172) for better performance efficiency.

However, while the capability to write the callback function makes it virtually possible to describe any sound synthesis algorithms, it is too low level for many sound synthesis algorithms as the user must write an excessive amount of the code themselves, unlike unit-generator languages. On the other hand, while an instrument-like abstraction can be useful in many cases, it may not be beneficial to some application domains like dynamic-patching (as seen in the performance by reacTable (159)(160)); the latter requires the dynamic modification of the sound synthesis algorithms **while the sound is being generated**; thus, Extempore still leaves a design issue when considering the generality of its programming model, though it would be useful for a certain kind of tasks.

LuaAV is an internal DSL built upon Lua, a dynamically-typed programming language and it can be flexible for dynamic modification at the compositional algorithm level. At the sound synthesis level, LuaAV provides a programming model similar to

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

SuperCollider in that it builds a *hierarchical directed acyclic graph* (DAC) (or simply, a *tree structure*) of unit-generators. A graph shown in Figure 4.16 can derive from such a statement ‘local dag = sin(time() * 500) * sin(time() * 0.1)’ as described in (273). The recent version of LuaAV translates a synthesis graph first into an equivalent C language program and then into a native machine code by just-in-time compiler, *clang* (171), a C compiler for LLVM and thus achieve a better efficiency in DSP.

In the current version of LuaAV, it does not seem possible to modify a synthesis graph for dynamic-patching, and may require further extension as seen in SuperCollider’s Just-in-Time programming. It may also require further consideration on how to realize such dynamism when just-in-time compilation to the native machine code is performed.

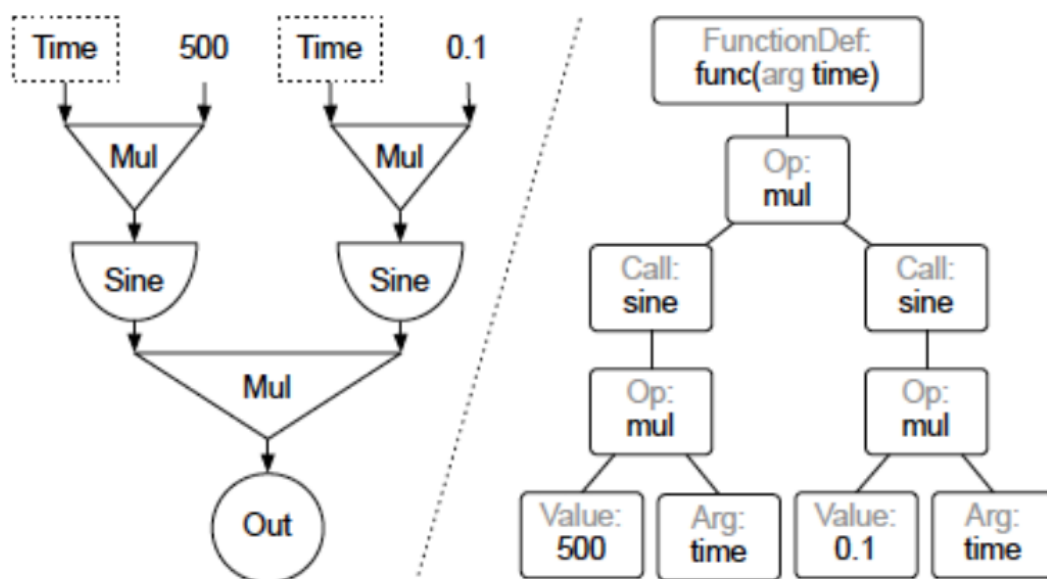


Figure 4.16: An oscillator with amplitude modulation: synthesis graph (left), and equivalent abstract syntax tree (right) - taken from Smith’s publication ‘Augmenting Computer Music with Just-in-time Compilation’ (273).

Visual computer music languages. Generally speaking, in visual computer music programming environments such as Max and PureData, a user can dynamically instantiate and delete unit-generators and the modification of a synthesis graph can be easily performed. Even putting aside the argument that graphical programming languages

4.2 Comparing LC with the existing computer music languages

can be much harder to comprehend in certain situations (131)(132), graphical programming with direct manipulation leaves some problems in computer music practices; as the manipulation of graphical objects can take effect immediately and often take more time than textual languages to make modifications with the keyboard and mouse, one may need to take extra care in dealing with the nature of direct manipulation in a live-coding performance.

While both PureData and Max offer some interfaces, so that lots of dynamic modifications can be performed immediately and automatically (e.g., by external programs or by an embedded scripting language), this means that such tasks can be difficult to achieve solely within the languages. In addition, if users are required to learn other programming languages, it is clearly not a very ideal situation.

4.2.1.2 The benefits of LC’s language design for dynamic modification of a computer music system

The programming concepts and models in the existing computer music languages described so far, propose several perspectives to view how and how much a computer music programming language can support dynamic modification of a computer music system at runtime. The following paragraphs discuss the benefits of the language design of LC, based on these perspectives.

Capability. Some languages are not capable of dynamic modification due to the language design or runtime environment. ChuckK has a significant drawback for dynamic modification because of its statically-typed class-based language design, and its runtime environment, which does not allow modification of a program interactively during execution, is also a significant obstacle. While LuaAV allows the execution of new code in the same runtime environment for live-coding, it is not capable of modifying a sound synthesis algorithm dynamically during its sound generation.

Granularity. It is also important to consider the granularity of dynamic modification, especially at the sound synthesis level. While dynamically-typed languages provide lots of flexibility at runtime, as the sound synthesis features are based on the underlying software framework or library, to what level a language supports dynamism at the sound synthesis level can differ a lot.

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

Since Impromptu depends on Apple’s Audio Unit framework, its granularity is not so fine-grained as the abstractions applied to Audio Units are more similar to virtual software synthesizer instruments; a user cannot redefine or modify sound synthesis algorithms with the same granularity as in other unit-generator languages, which are capable of describing much lower level sound synthesis algorithms. While Extempore provides the lowest level granularity to describe the callback function for DAC output, this is likely too low level for many tasks in computer music. While it is possible to redefine note-level objects and sound effect algorithms for virtual instruments, such a model is not appropriate for some tasks that require dynamic modification of a sound synthesis graph (e.g. dynamic patching as seen in `reactTable`).

Premature commitment. Even when dynamic modification is possible, if the language design enforces a user to consider where and what to modify before making the actual modification, such a situation may not be ideal to support creative exploration by computer musicians.

The problems of premature commitment can frequently occur in Just-in-Time programming in SuperCollider, since proxy objects must be placed at the points where there can be dynamic modification of a synthesis graph. ChuckK also can exhibit a premature commitment problem when replacing unit-generators, as a reference to the parent class must be used before making the actual replacement. On the other hand, in Impromptu, it is only necessary to update the synthesis graph after performing dynamic instantiation of Audio Units and modification to the connections.

Viscosity. It is important, not just whether the dynamic modification is possible, but also to see how much effort must be involved to make a dynamic modification. Graphical computer music languages clearly exhibit such problems of viscosity. While a simple modification such as connecting/disconnecting two unit-generators can be easily performed in graphical languages, the direct manipulation in graphical programming environments normally involves more user actions and takes longer time for more complex modification at both levels of compositional algorithms and sound synthesis.

Textual computer music languages can also exhibit the viscosity problems. As seen in the Figure 4.12 example, SuperCollider’s Just-in-Time programming can be considered viscous when one needs to refactor the code by inserting proxy objects for

4.2 Comparing LC with the existing computer music languages

more dynamic modification in a synthesis graph. ChuckK also exhibits the problem of viscosity when replacing unit-generators. As in Figure 4.15 (below), it is necessary to disconnect the old unit-generator from the graph and then connect the new one to perform the replacement. If a unit-generator is connected to many other unit-generators, one must perform the same procedure to all the connections. Impromptu's programming model also exhibits the same problem as ChuckK.

On the other hand, in Max and PureData, one can simply change the type of an object directly while keeping the existing connections with the other objects and the viscosity with the object replacement is not exhibited. The viscosity problem in the replacement of the unit-generators seen in SuperCollider, ChuckK, and Impromptu seem mostly due to that these languages **build the connections directly between the inlets and outlets of the unit-generator instances**; it is better to manage the connections in a synthesis graph separately from the instances.

The benefits of LC's language design. LC was designed to take such problems in the existing languages into consideration, and its design and implementation offer better support for the dynamic modification of a computer music program to a considerable degree in comparison with predecessors; LC adopts prototype-based programming at both levels of compositional algorithms and sound synthesis and provide a terse and consistent programming model, which is beneficial for dynamic modification of a computer music system. As described in Chapter 8, LC's virtual machine dynamically loads new bytecode and executes it within the same name space and the same memory space. This makes on-the-fly redefinition and modification easier.

LC is capable of dynamic modification at both levels of compositional algorithms as well as other languages as SuperCollider, Impromptu, Max/MSP, and PureData. All these languages are designed as a dynamically-typed textual language or a graphical language with direct manipulation. Even the programming environment developed only to test the current proof-of-concept prototype of LC allows such dynamic modification even during a program being executed. For instance, suppose there is a loop that keeps on calling a certain function assigned to the global variable *a* every 200 msec. If a user writes a new function and assigns it to the global variable *a*, the loop will then call the

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

new function from the next iteration. Thus, the language design of LC is very suitable for an interactive programming environment¹.

The granularity of dynamic modification in LC is fine-grained. It supports the dynamic modification at the level of the unit-generator graph. Moreover, it can directly compute output samples, even without depending on unit-generators; thus, LC's sound synthesis framework supports dynamism at various granularity levels.

Dynamic modification in LC does involve less premature commitment in comparison with other languages. It provides a similar programming model to Impromptu, as it can dynamically instantiate new unit-generators/subpatches and reconnect synthesis graphs without any preparation before actual modification. Calling *compile* method can update the patch to reflect the modification.

LC is less viscous in the modification of a synthesis graph. As in Max/MSP or PureData, its Patch object also manages the connections between unit-generators (and subpatches) separately from the instances. As previously describe in Section 3.2.1.2, LC does not directly connect the unit-generator instances as in other textual languages above and instead Patch object manages the connections between its slots rather than the instances.

The cause of the obstacles against dynamic modification seems rooted in many different aspects of computer music languages (e.g. what kind of programming paradigms and concepts are adopted, how sound synthesis is abstracted, and how the runtime environment such as interpreter, compiler, virtual machine, and the software synthesis framework, are implemented). The design of LC is considerably flexible in dynamic modification and also beneficial for supporting creative exploration by users, for example, in rapid-prototyping and live-coding activity.

4.2.2 The support for precise timing behaviour and other features with respect to time

Supporting precise timing behaviour is still an important criterion in computer music language. While the desirable precision was much lower in the hybrid computer music systems with external MIDI synthesizers in the 1980s, today's computer music

¹Sorensen and Gardner calls such an interactive programming activity as seen in live-coding as *cyber-physical programming* in (276), considering a programmer as one of the agents involved in the entire system.

4.2 Comparing LC with the existing computer music languages

systems require more precise timing behaviour. Microsound synthesis techniques require sample-rate accuracy in scheduling microsounds to precisely render its output. Also, even at the rhythmic level, it is argued that “a pulsation may feel not quite right when there are a few 10s of milliseconds of inaccuracy in the timing from beat to beat” (194); thus, today’s computer music practices require much higher precision in timing behaviour than before.

While many computer music systems and languages still have some problem in precise timing behaviour, some of the recent computer music languages have achieved precise timing behaviour, with sample-rate accuracy by introducing the concept of synchronous programming. Yet, such languages can suffer from the temporal suspension of real-time sound synthesis in the presence of time-consuming tasks.

Furthermore, many desirable features with respect to time, such as execution time constraints, which are almost indispensable in real-time programming languages, are still not available in many recent computer music languages, while some of such features were implemented in the hybrid computer music systems of the previous decades. The following sections discuss such issues in the existing languages and describes the benefits of LC’s language design.

4.2.2.1 Timing behaviour in the existing computer music languages

This section mainly focuses on recent programming languages and discuss the precision of timing behaviour, as today’s computer music systems require much higher precision in timing behaviour than the hybrid computer music systems with external MIDI synthesizers. However, FORMULA (Forth Music Language) (16) should be worthy of reviewing in that its *virtual system time* concept is a precursor to the recent synchronous approach as seen in ChuckK and LuaAV as described in Section 2.2.

ChuckK and LuaAV. Among recent computer music languages, ChuckK and LuaAv are of significant interest as these languages achieve sample-rate accuracy in timing behaviour. While ChuckK proposes the concept of strongly-timed programming, the behaviour of LuaAV is based on a similar synchronous approach. ChuckK is also interesting in that it proposes the deterministic behaviour in scheduling and order of its threads. However, as discussed above, these languages based on such synchronous

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

approaches can suffer from the temporal suspension of real-time DSP in the presence of time-consuming tasks.

SuperCollider and Impromptu. Both SuperCollider and Impromptu (and Extempore) separate the sound synthesis servers from the interpreters and are not based on the synchronous approach as in ChuckK and LuaAV. Hence, while both adopt the concept of logical time in the design, there is still the necessity to take the passage of real time into account to some degree when programming. Moreover, the separation between interpreters and synthesis servers in these languages makes it impossible to precisely synchronize compositional algorithms and sound synthesis algorithms with sample-rate accuracy. Such a drawback is considered problematic to certain application domains.

Other languages. While PureData is also implemented based on the synchronous approach, as it computes output samples by audio vectors, and compositional algorithms are executed in the same thread with audio computation between DSP cycles, the precision of timing behaviour is determined by the size of audio vectors. As in ChuckK and LuaAV, time consuming-tasks can temporarily suspend real-time DSP in PureData.

On the other hand, Max/MSP normally processes compositional algorithms in a separate thread to avoid such real-time DSP suspension; this leads to imprecise timing behaviour as in SuperCollider and Impromptu, and users must take extra care when timing precision matters.

4.2.2.2 Other features with respect to time in the existing computer music languages

As already mentioned, many recent computer music programming languages still lack other desirable features with respect to time.

In ‘Motivating time as a first class entity’, Lee et al. (179) discuss six desirable features for high-level real-time programming: (1) expression of timing, (2) timed communication¹, (3) enforcement of timing constraints, (4) time fault tolerance, (5) maintaining consistency in distributed real-time systems, and (6) static timing verification.

¹‘Timed communication’ should not be confused with time-tagged messages as seen in Section 3.2.2.2.

4.2 Comparing LC with the existing computer music languages

Table 4.1 describes these six features with more detail.

While many computer music languages rarely refer to such features of real-time programming languages, these six desirable features can also be considered criteria for computer music language design, as seen in Sorensen’s discussion on Impromptu(276). The following discussion on the features with respect to time also refers to these six features.

Impromptu is particularly interesting, as the developers considered these features in the language design. FORMULA (15)(16) is another interesting work as a predecessor language of this kind. Even though FORMULA itself is a language/system developed around early the 1990s and only targeting hybrid computer music systems with the external synthesizer hardware, and such hybrid computer music systems may seem already outdated today, the concepts to integrate these time-related features in computer music systems are still worthy of reviewing.

FORMULA. Since FORMULA is designed for hybrid computer music systems with external MIDI synthesizer, its precision of timing behaviour is assumed to be much lower than what is required for real-time sound synthesis today. Yet, the features of FORMULA described in (16) seem to implement some of the six desirable features as above.

- **expression of timing** — Start-time constraint is implemented as library functions for task scheduling and execution-time constraint is implemented as *time-control structures*: ‘maxtime(n) statement’, ‘mintime(n) statement’, and ‘min-loop(n) statement’. For instance, when the time-control structure ‘maxtime(n)’ executes its following statement, which may be compound statement ($\{\dots\}$), if the ‘time_advance’ function call advances FORMULA’s *virtual system time* over the given execution constraint, it immediately stops the execution of the current statement and jumps to the next statement right after ‘maxtime(n) statement’. Yet, as the only point to check if the violation occurred or not is the *time_advance* function call, it is uncertain how an execution time constraint can be realized within FORMULA’s background process, which may not call this function.

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

expression of timing	it is desirable for language to support timing constraints (e.g., start-time constraint and execution-time constraint). The feature of variable time constraints is required as "timing constraints may be variable with their value depending of the state of the environment" (179). As timing constraints may be nested, the support for nested time constraints is also required.
timed communication	is described as predictable communication and time constrained communication. For the former, it is emphasized that the communication is predictably fast enough and should support asynchronous messages and concurrent shared data so that "the forms of communication are low overhead and thus physically fast", since "neither form requires waiting in a queue or buffer (or at most a small buffer for asynchronous messages)" (179). For the later, as "messages often have a specific interval when they are valid", "for the <code>send</code> primitive the ability to stamp a message with a validity time interval is required" and "for the <code>receive</code> primitive the ability to specify a deadline for waiting for a message is required" (179).
enforcement of timing constraints	"It is necessary to provide system support to enforce timing constraints. System support includes: The underlying system detecting violations of timing constraints and invoking appropriate action when they are violated; The run-time system changing timing constraints due to nesting of constraints, propagation of constraints form a sending process based on the urgency of messages, and the setting of values for variable constraints; The run-time scheduler scheduling processes based on their current timing constraints so that all system constraints can be met" (179).

Table 4.1: The six required features for high-level real-time programming as Lee et al. discuss in (179)

4.2 Comparing LC with the existing computer music languages

time fault tolerance	"Violation of timing constraints are called time faults. In order to provide complete control, the system must be time fault tolerant by handling missed timing constraints" (179).
maintaining consistency in distributed real-time systems	It is necessary "to guarantee a consistent global state, some changes to the system state must be carried out completely or not carried out at all so as not to leave the system in an inconsistent state" and "the traditional solution is to provide atomic actions with the property that the actions either complete entirely or have no effect". "Adding atomicity with respect to time faults leads to the notion of timed atomic action". "A timed atomic action either performs completely within its timing constraints or appears as if it never executed" (179).
static timing verification	is "offline static verification of real-time programs". "Static timing verification tools examine the timing that was expressed and determine if the timing constraints can be met" (179).

Table 4.1: The six required features for high-level real-time programming as Lee et al. discuss in (179) (continued).

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

- **timed communication** — It does not seem considered in the language features of FORMULA. However, as FORMULA is equipped with the mutual exclusion mechanism, concurrent shared data can be used for fast communication between processes, while the asynchronous message communication mechanism does not seem supported.
- **enforcement of timing constraints** — It is described that FORMULA's operating systems can enforce timing constraints by the underlying scheduler in (16). FORMULA is also capable of handling the nested time-control structures.
- **time fault tolerance** — FORMULA's time-control structure does not appear to have a handler when any violation has occurred. Using a flag that is set to 'false' in the end of a block that is given a time constraint may be helpful to check if any violation has occurred in order to handle the violation.
- **maintaining consistency in distributed real-time systems** — This issue is not discussed in the related publications (15)(16). Yet, as FORMULA has the mutual exclusion mechanism, it can be used to maintain consistency as is seen in general-purpose programming languages.
- **static timing verification** — This is another issue that is not discussed in their publications.

Impromptu.

- **expression of timing** — Impromptu can express both start-time constraints and execution-time constraints by its underlying scheduler (276). Figure 4.17 describes an example of execution-time constraints in Impromptu (276). As shown, an execution-time constraint in Impromptu can be given when passing a first-class function to a scheduler. However, this programming model exhibits a serious problem as it cannot express the nested timing constraints as described later.
- **timed communication** — Impromptu's primary communication mechanism is remote-procedure calls (RPCs) (276) and both synchronous and asynchronous

4.2 Comparing LC with the existing computer music languages

RPCs are supported¹. Time constrained communication can be realized by giving an execution-time constraint to RPCs.

- **enforcement of timing constraints** — Both start-time and execution-time constraints are available in Impromptu. Impromptu’s scheduler is responsible for dispatching events to the requested process. Each process is responsible for execution-time constraints (276).
- **time fault tolerance** — When Impromptu missed start-time constraints, the underlying the scheduler simply culls the missed events and prints out an error message. The violation of execution time constraints is not handled at all.
- **maintaining consistency in distributed real-time systems** — Sorensen describes the use of Tuple space (275)(276) for the maintenance of consistency in distributed real-time systems in Impromptu.
- **static timing verification** — Static timing verification is not available in Impromptu. However, such a feature is almost unrealisable as live-coding activity involves the creation and modification of a new program on-the-fly. It is impossible to estimate what kind of tasks a user may program and execute beforehand.

Other languages. Many other computer music languages still lack these six desirable features. Even among the widely-used computer music languages, neither SuperCollider, ChuckK, LuaAV, Max/MSP or PureData supports execution-time constraints. While most of these languages support Open Sound Control (OSC) (324) for asynchronous communications, the lack of execution-time constraints in these languages may make it harder to realize time constrained message communications. The enforcement of timing constraints and tolerance constrains with respect to time seem not considered in the design of these languages.

The maintenance of the consistency in distributed real-time systems may be realizable as these languages do not provide preemptive threads and atomicity can be expected; as the requests can be serialized and processed without preemption. Static

¹“Asynchronous RPC calls do not block the caller (client) and the replies can be received as and when they are needed, thus allowing the client execution to proceed locally in parallel with the callee (server) invocation” (14) whereas on the contrary synchronous RPCs block until it receives the result.

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

```
01: ;; video player temporal recursion looping
02: ;; at a rate of 1/24th of one second
03: ;; maximum-execution time of 1/32nd of one second
04: (define video-player
05:   (lambda (time mov position)
06:     (let ((frame (gfx:get-movie-frame mov position)))
07:       (gfx:draw-image time *canvas* frame 1)
08:       (schedule (cons (+ (now) (/ *second* 24))
09:                       (/ *second* 32))
10:                 video-player
11:                 (+ time (/ *second* 24))
12:                 mov (+ position 1/24))))))
13: ;; this call starts the temporal recursion
14: ;; the execution deadline for this first call
15: ;; is the default execution duration of the process
16: (video-player (now) (gfx:load-movie "/tmp/myfilm.mp4") 0.0)
```

Figure 4.17: An execution-time constraint example in Impromptu (276).

timing verification is not provided, yet this is also unrealizable in live-coding as discussed in the previous paragraph on Impromptu.

4.2.2.3 The benefits of LC’s language design

Precise Timing Behaviour. LC is a mostly-strongly-timed programming language and its behaviour is with the precision of sample-rate accuracy. The mostly-strongly-timed programming concept extends the strongly-timed programming concept by integrating the explicit context switching between the synchronous context and the asynchronous context and time-consuming tasks can be preempted when necessary.

Other features with respect to time. LC has also taken these six desirable features with respect to time into account in the language design as below.

- **expression of timing** — LC can express both start-time constraints and execution-time constraints. The former is supported both for threads and patches. Patches can be given the duration as execution-time constraints for sound synthesis. Threads can be given execution time constraints with the *within* statement.

4.2 Comparing LC with the existing computer music languages

- **timed communication** — LC's inter-thread communication is based on a message passing mechanism. Message passing is performed asynchronously for send primitives and both synchronous (blocking) and asynchronous (non blocking) behaviour is supported for receive primitives. As LC provides the execution-time constraint feature, it can be used for timed communication. Moreover, When LC receives a message, it comes with the timestamp of the actual delivery time (in logical time), thus it is easy to check the validity of the received message if the message can be invalid after a certain duration. Thus, LC meets the criteria for timed communication. Additionally, LC's send primitive can specify the delivery time.
- **enforcement of timing constraints** — The enforcement of timing constraints is performed by LC's virtual machine. As LC is based on logical synchronous time as it is a mostly-strongly-timed programming language with sample-rate accurate timing behaviour, these constraints are always enforced with precise timing.
- **time fault tolerance** — As LC is based on logical-time, start-time constraints are always performed with precise timing, which means start-time constraints will never be violated. Execution-time constraints can be handled by the *timeout* statement with sample-rate accuracy in timing.
- **maintaining consistency in distributed real-time systems** — LC can provide atomicity required for this issue by using the *sync* statement, as no other thread can be executed while one thread is in 'sync' context¹.
- **static timing verification** — LC is assumed to be utilized in an interactive programming environment. It is impossible to realize this feature as one can write/modify a program at runtime and execute it interactively.

The benefits of LC's language design. LC's mostly-strongly-timed programming concept provides precise timing behaviour with sample-rate accuracy, as seen in ChuckK and LuaAV. These three languages also provide multi-tasking by coroutines (LuaAV) and by lightweight concurrency by software thread (ChuckK and LC). Such a language

¹This behaviour in 'sync' context is guaranteed because the current prototype of LC is concurrent but not parallel.

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

design also contributes to sample-rate accurate synchronization not just between compositional algorithms and sound synthesis but also between compositional algorithms that are separated into several different threads (or coroutines).

However, while ChuckK and LuaAV also support sample-rate accuracy in timing behaviour, real-time DSP can be temporarily suspended in the presence of time-consuming tasks, as they are only based on the synchronous approach. Fragmenting a time-consuming task by inserting the explicit advance of logical synchronous time cannot always solve the problem, as there are some tasks that involve an I/O block and unpredictable timing. Such tasks cannot be fragmented despite being time-consuming; file access is a typical example of this kind.

LC's *mostly-strongly-timed programming* extends strongly-timed programming by integrating the explicit context switching between the synchronous context and the asynchronous context, and time-consuming tasks can be preempted when necessary. Such time-consuming tasks can be explicitly enclosed in an 'async' block and be executed regardless of the progress of logical time as a background task.

While FORMULA allows such tasks to run in background, it targets a hybrid computer music system with an external synthesizer and is outdated, as today's computer music systems require sample-rate accuracy to perform certain synthesis techniques, such as microsound synthesis techniques. Moreover, even when enclosing some time-consuming part of a task as a background process, a computer music program often involves the collaboration between a foreground process and a background process. This can result in unnecessary complication of the resulting code, involving inter-process communication even for a simple task; as the simple explicit context switching in LC can describe the time-consuming part just as a compound statement ('{' ... '}') to be executed in the background, but within the same thread of execution, it doesn't involve much complication.

Thus, LC provides sample-rate accuracy in timing behaviour while allowing time-consuming tasks to run in the background to ward off the suspension of real-time DSP, and only simple explicit switching between the synchronous context and the asynchronous context is required. Such a feature does not exist in any other computer

4.2 Comparing LC with the existing computer music languages

music language¹.

When considering these six desirable features with respect to time, both FORMULA and Impromptu seem to support these features to a certain degree. FORMULA’s ‘max-time(n)’ time-control structure for execution-time constraints (in its virtual system time) does not support the handler for the violation. The support for tolerance to violations and constraints seems insufficient and the violation of execution time-constraints are checked only at the *time_advance* function call. Such a design can be problematic if an execution-time constraint must be given to a background task; many real-time programming languages check the violation by a watchdog thread, since execution-time constraints should be treated as *asynchronous transfer of control* (ATC), which “is a transfer of control within a thread, triggered not by the thread itself but rather from some action by another thread or an event handler” (60). As a violation is examined at the *time_advance*’ function call in FORMULA, it can lead to a problem in a background process, in which no ‘time_advance’ function call may be invoked. Moreover, it is targeting event level communications with the external hardware (such as MIDI synthesizers), and its precision in timing is far from sample-rate accuracy, which is required for recent computer music practices.

Impromptu is another interesting language that considers the features with respect to time in its design. However, its programming model for execution-time constraints has a problem in expressing nested execution-time constraints. As seen in the Figure 4.17 example, execution-time constraints are specified when passing functions to Impromptu’s scheduler. However, such a design makes it harder to realize the nested execution time constraints. Even the inner constraint must be given, the only entry point of the execution timing constraints in Impromptu is *schedule* function call and it immediately returns just after scheduling a function, without waiting until the scheduled function is finished. Thus, the nested timing constraints can be hardly expressed

¹Yet, it should be noted that there has been some effort to integrate asynchronous behaviour into synchronous programming languages as seen in the previous works such as (39) and (29). Nevertheless, these works are motivated by such issues as the extension of reactive systems, for *communicating reactive processes*, “where a set of individual reactive synchronous processes is linked by asynchronous communication channels” (39), or better performance efficiency by asynchronous concurrency in reactive systems. Thus, there is a substantial difference in the context, the motivation, the target applications and the programming language concept between these works and LC; LC’s mostly-strongly-timed programming is focused on precise timing behaviour in an imperative programming language designed for interactive systems, with a significant focus on interactive music systems.

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

in Impromptu’s programming model, whereas it is only necessary to nest the *within* statements in LC. Moreover, Impromptu does not provide the feature to handle the violation of execution time constraints.

On the contrary, LC provides five of the six desirable features and the only feature that is not supported is static timing verification. Yet, generally speaking, static timing verification involves the analysis of worst case execution time (WCET) and also needs to collaborate with the compiler/interpreter and requires detailed information about the environment. As described, such an assumption cannot be made in LC’s target application domains; for instance, in live-coding, programmers write and modify a program on-the-fly, and the information on programs that may be launched cannot be given beforehand, while static time verification requires such information.

Thus, as above, LC provides sample-rate accuracy in timing behaviour with background processes, and it also meets the requirements of five out of six desirable features proposed by Lee et al. (179), except static timing verification, which is unrealizable in LC’s target application domain.

4.2.3 The difficulty in programming microsound synthesis techniques

In Section 2.3, the difficulty in programming microsound synthesis techniques was described and assessed as a software anti-pattern of *abstraction inversion*. Based on such a view, LC integrates the objects and library functions for microsound synthesis techniques, expecting to reduce the difficulty by removing the problem of abstraction inversion. In the following sections, the benefit of such sound synthesis framework design is discussed.

4.2.3.1 Abstraction inversion in the unit-generator languages

As discussed in Section 2.3, abstraction inversion is a software anti-pattern, which occurs “when a programmer is forced to employ a combination of higher-level abstractions to express a lower-level abstraction” (28). The lack of bitwise operators in the early versions of Lua provide a typical example of abstraction inversion, in that a user has to write such a code as in Figure 4.18 just to perform a very simple bitwise operation.

4.2 Comparing LC with the existing computer music languages

Such a problem of abstraction inversion can be seen in the implementation of microsound synthesis techniques in the existing unit-generator languages, as a user has to combine higher-level abstractions by modelling microsounds as a note-level object that consist of unit-generators and scheduling them. Even when implementing a microsound synthesis technique, which is very simple in concept, the resulting code can be a lot more complicated. Such examples as Figure 4.19 (waveset harmonic distortion in SuperCollider) and Figure 4.20 (synchronous granular synthesis in ChuckK) are briefly described.

To make matters worse, as each microsound is modelled as a note-level object, applying the effects to the entire output of microsound synthesis techniques can involve more complexity; Figure 4.21 shows such an example. As shown, to apply a triangle envelope to the output of synchronous granular synthesis, one must use a ‘bus’ to route the output from each grain to the overall envelope instrument and then start them altogether and thus introduce more complexity in the implementation. The equivalent code example in ChuckK in Figure 4.22 exhibits another kind of problem in envelope-shaping. In this example, while it is only required to connect the output of the threads to play grains to the input of the unit-generator that applies the entire envelope, it must involve concurrency by multi-threading as shown¹.

4.2.3.2 When black-box abstractions do not benefit

One may argue that the abstraction mechanism in programming languages can reduce the difficulty in microsound synthesis programming. For instance, Figure 4.23 shows such an example² of waveset harmonic distortion in Nyquist (89)(90). While Nyquist is originally designed as an internal DSP built on LISP, this example is written in the SAL programming language³. .

¹Such a problem in ChuckK is likely due to both its lack of garbage collection and the software design of unit-generators. The introduction of garbage collection and the redesign of the envelope unit-generators may reduce the complexity. For instance, the complexity of the code may be reduced by reimplementing the SndBuf unit-generator to access the shared sound buffer rather than loading its own sound from the disk and by giving the overall envelope shape of the envelope shaper unit-generator at its instantiation. However, it is still required to involve multi-threading, and explicit disconnection of each grain from the entire envelop must be performed, even after such improvement; each grain must be still modelled as a thread in this programming model and the garbage collector won’t be able to collect the garbage unit-generator objects if it is still connected to the other live unit-generators.

²Thanks to an anonymous programmer from the Nyquist community for providing this example.

³The SAL languages is built on LISP, using LISP’s powerful macro definition, as a part of Nyquist’s programming environment. The designers of Nyquist describes that users “who are put off by LISP

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

```
01: function bit_and(x, y)
02:   local digit = 1
03:   local ret = 0
04:   local limit = x > y and x or y
05:   while digit <= limit do
06:     if (x % (digit * 2)) >= digit and (y % (digit * 2)) >= digit then
07:       ret = ret + digit
08:     end
09:     digit = digit * 2
10:   end
11:   return ret
12: end
```

Figure 4.18: A bitwise operation (bitwise-and) example in Lua (reproduced from 2.18).

While Nyquist itself is not a computer music language for interactive computer music systems and is basically a language for non real-time sound synthesis, the code example may seem fairly terse on the surface as the equivalent example in LC (Figure 4.32), yet it should be noted such a black box abstraction is not beneficial for creative exploration in computer music.

In Figure 4.23, the code first uses a library function, *extract-wavesets*¹. The *extract-wavesets* call is assumed to return an array of sound objects, each of which can be used to play a single waveset extracted from the given sound file. Then, the code performs waveset harmonic distortion using Nyquist’s *seqrep* function². The *seqrep* function combines the iteration with an index variable and scheduling of sound objects to perform sequential repetition. In the example, the index value *i* first starts from 0 then is incremented to the given limit, the length of *wavesets* array by *seqrep* macro on line 4. In each iteration, a waveset at the index *i* is given as an argument to the *harm-dist* function call and *harm-dist* plays it together with its 1st harmonic, which repeats twice while the original waveset is being played. After finishing playing these wavesets, *seqrep* proceeds to the next iteration to play another waveset and its harmonics; thus, waveset harmonic distortion is performed.

syntax may find Nyquist more accessible and easier to learn.” (91)

¹This *extract-wavesets* seems not provided as a part of Nyquist. Yet, the code suggested by the anonymous Nyquist programmer assume such a function is provided in this situation.

²As Dannenberg describes in the Nyquist Reference Manual (92), “*technically, seqrep is not really a function but abbreviation for a special kind of loop construct*”. Yet, the manual often calls it a “function”; we follow the same tradition among Nyquist programmers.

4.2 Comparing LC with the existing computer music languages

```
01: Server.default = s = Server.internal;
02: s.boot;
03: w = Wavesets.from("sound.aif");
04: (
05:   b = w.buffer;
06:   SynthDef(\wvst0, {
07:     arg out = 0, buf = 0, start = 0, length = 441,
08:     playRate = 1, sustain = 1, amp = 1;
09:     var phasor = Phasor.ar(rate:playRate, start:0, end:length) + start;
10:     var env = EnvGen.ar(Env([amp, amp, 0], [sustain, 0]), doneAction:2);
11:     var snd = BufRd.ar(1, buf, phasor) * env;
12:     OffsetOut.ar(out, snd);
13:   }).add;
14: )
15: (
16:   var numOfWavesets = w.lengths.size;
17:   var original = Pbind(
18:     \instrument, \wvst0,
19:     \startWs, Pseries(0, 1, numOfWavesets),
20:     \numWs, 1,
21:     \playRate, 1,
22:     \bufnum, b.bufnum,
23:     \repeats, 1,
24:     \amp, 1,
25:     [\start, \length, \sustain], Pfunc( { |ev|
26:       var start, length, wsDur;
27:       #start, length, wsDur = w.frameFor(ev[\startWs], ev[\numWs]);
28:       [start, length, wsDur * ev[\repeats] / ev[\playRate].abs]
29:     }),
30:     \delta, Pkey(\sustain)
31:   );
```

Figure 4.19: A waveset harmonic distortion example in SuperCollider (reproduced from Figure 2.20).

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

```
32: var octup = Pbind(  
33:   \instrument, \wvst0,  
34:   \startWs, Pseries(0, 1, numOfWavesets),  
35:   \numWs, 1,  
36:   \playRate, 2,  
37:   \bufnum, b.bufnum,  
38:   \repeats, 2,  
39:   \amp, 1,  
40:   [\start, \length, \sustain], Pfunc( {|ev|  
41:     var start, length, wsDur;  
42:     #start, length, wsDur = w.frameFor(ev[\startWs], ev[\numWs]);  
43:     [start, length, wsDur * ev[\repeats] / ev[\playRate].abs  
44:   ]},  
45:   \delta, Pkey(\sustain)  
46: );  
47:  
48: Ppar([original, octup]).play(SystemClock);  
49: )
```

Figure 4.19: A waveset harmonic distortion example in SuperCollider (continued) (reproduced from Figure 2.20).

However, the brevity of this Nyquist example is superficial, and it is questionable whether such a solution by black-box abstraction of the detail can be beneficial to users when considering the purpose for which computer musicians write their own programs. It should be emphasized with a significant remark that the reason why computer musicians program in computer music programming languages is mainly for *exploratory design* and *exploratory understanding*¹ for their creative practices.

For such activities, abstracting the complex detail within a function as a black-box do not benefit the users, if what they want to explore is the algorithms hidden within the function; in such a case, users have to access the hidden details to modify or to understand the algorithms and they face the complexity of the implementation anyway. To make matters worse, due to the abstraction barriers, which “isolate different levels of the system” (5, p.88), there can be more difficulty in modification. For instance, to

¹Blackwell and Green list such activities as *sketching; design of typography, software, etc.; other cases where the final product cannot be envisaged and has to be ‘discovered’* as the examples of exploratory design and *discovering structure of algorithm, or discovering the basis of classification* as the examples of exploratory understanding (41).

4.2 Comparing LC with the existing computer music languages

```
01: "a11w1k01-44_1.aif" => string filename;
02:
03: 22050 => int startPos;
04: 0.1 => float sustain;
05: 40 => int rep;
06: 0.05 => float interval;
07: 0 => int cnt;
08:
09: (sustain / interval) $ int => int numOfThreads;
10: if (sustain / interval > numOfThreads){
11:   numOfThreads + 1 => numOfThreads;
12: }
13: fun void grain()
14: {
15:   SndBuf buf => Envelope env => dac.left;
16:   filename => buf.read;
17:
18:   while(cnt < rep){
19:     cnt + 1 => cnt;
20:
21:     now + numOfThreads * interval::second => time nextGrainStartTime;
22:
23:     startPos => buf.pos;
24:     sustain::second / 2 => env.duration;
25:
26:     env.keyOn();
27:     (sustain * 0.5)::second +=> now;
28:     env.keyOff(); 29:     (sustain * 0.5)::second +=> now;
30:
31:     nextGrainStartTime =>now;
32:   }
33:   return;
34: }
35:
36: for(0 => int i; i < numOfThreads; i + 1 => i){
37:   spork ~ grain();
38:   interval::second +=> now;
39: }
40: (interval * (rep - numOfThreads) + sustain)::second +=> now;
```

Figure 4.20: Another synchronous granular synthesis example in ChuckK with less memory-leak (reproduced from 2.23).

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

```
01: //booting the sound synthesis server
02: Server.default = Server.internal;
03: s.boot;
04: //read the sound file to extract grains.
05: b = Buffer.read(s, "/sound/sample1.aif");
06:
07: //to define instruments, evaluate below.
08: (
09: //the instrument to apply a triangle envelope to the grains,
10: //which are received from the bus input.
11: SynthDef(\envelope, {
12:   arg out = 0, inbuf = 127, sustain = 1.0;
13:   var env = EnvGen.ar(
14:     Env.triangle(sustain),
15:     doneAction:2
16:   );
17:   var snd = In.ar(inbuf) * env;
18:   OffsetOut.ar(out, snd);
19: }).add;
20:
21: //the instrument that plays a single grain, output of which
22: //is routed to the bus input of the envelope instrument above.
23: SynthDef(\grain, {
24:   arg out = 0, buf = 0, start = 0, sustain = 1.0, rate = 1;
25:   var snd = PlayBuf.ar(1, buf, rate, startPos:start);
26:   var env = EnvGen.ar(Env.triangle(sustain), doneAction:2);
27:
28:   var grain = snd * env;
29:   OffsetOut.ar(out, grain);
30: }).add;
31: )
32: //to perform synchronous granular synthesis, evaluate below.
33: (
34: //these are parameters for synchronous granular synthesis.
35: var out = 0;
36: var bufnum = b.bufnum;
37: var startPos = 22050;
38: var sus = 0.1;
39: var interval = 0.05;
40: var repeat = 20;
41: var route = 127;
42: var entireDur = interval * repeat + sus;
```

Figure 4.21: A synchronous granular synthesis example with a triangle envelope applied to the entire sound output in SuperCollider.


```
43: //this generates an 'event stream' to play an envelope instrument once.
44: var env = Pbind(
45:   \instrument, \envelope,
46:   \out, out,
47:   \inbuf, route,
48:   \sustain, Pn(entireDur, 1),
49:   \delta, 0
50: );
51: //this generates an 'event stream' to play grains.
52: var grains = Pbind(
53:   \instrument, \grain,
54:   \out, route,
55:   \buf, bufnum,
56:   \sustain, sus,
57:   \start, Pn(startPos, repeat),
58:   \delta, interval
59: );
60: //playing the above instrument with the same timing.
61: Ppar([env, grains]).play(SystemClock);
62: )
```

Figure 4.21: A synchronous granular synthesis example with a triangle envelope applied to the entire sound output in SuperCollider (continued).

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

```
01: "a11w1k01-44_1.aif" => string filename;
02: 22050 => int startPos;
03: 0.1 => float sustain;
04: 40 => int rep;
05: 0.05 => float interval;
06: 0 => int cnt;
07: (sustain / interval) $ int => int numOfThreads;
08: if (sustain / interval > numOfThreads){
09:     numOfThreads + 1 => numOfThreads;
10: }
11:
12: Envelope globalEnv => dac.left;
13:
14: fun void overAllEnvelope(float entireDuration)
15: {
16:     entireDuration::second / 2 => globalEnv.duration;
17:     globalEnv.keyOn();
18:     (entireDuration * 0.5)::second +=> now;
19:     globalEnv.keyOff();
20:     (entireDuration * 0.5)::second +=> now;
21:     return;
22: }
23: fun void grain()
24: {
25:     SndBuf buf => Envelope env => globalEnv;
26:     filename => buf.read;
27:     while(cnt < rep){
28:         cnt + 1 => cnt;
29:         now + numOfThreads * interval::second => time nextGrainStartTime;
30:         startPos => buf.pos;
31:         sustain::second / 2 => env.duration;
32:         env.keyOn();
33:         (sustain * 0.5)::second +=> now;
34:         env.keyOff(); 35:         (sustain * 0.5)::second +=> now;
36:         nextGrainStartTime =>now;
37:     }
38:     return;
39: }
40:
41: for(0 => int i; i < numOfThreads; i + 1 => i){
42:     spork ~ grain();
43:     interval::second +=> now;
44: }
45: (interval * (rep - numOfThreads) + sustain)::second +=> now;
```

Figure 4.22: Another synchronous granular synthesis example in ChuckK with a triangle envelope applied to the entire sound output.

4.2 Comparing LC with the existing computer music languages

perform waveset substitution, it is necessary to obtain the maximum amplitude of each waveset to scale the substituting wavesets to the same amplitude as the original. Yet, both the definition of the sound objects and the analysis of the given sound data are encapsulated within the *extract-wavesets* function in the Figure 4.23 example. These details are hidden under the abstraction barriers and a user must investigate the implementation of the *extract-waveset* function for this purpose; this seems contradictory that users have to face the complexity of the implementation while the motivation to abstract the implementation within a function is to hide the complexity itself from the users.

Furthermore, while Nyquist provides several different methods for scheduling, an end-user may also want to investigate what the *seqrep* function does inside when experimenting with more complex algorithms in scheduling microsounds. While *seqrep* is a basic control structure in a programming language as well as a for-loop is, it is not as primitive as other control structures, as Nyquist’s *seqrep* combines two different features, iteration and scheduling¹. As shown in Figure 4.24, the actual implementation of *seqrep* requires the further understanding of LISP and Nyquist’s sound synthesis framework; this is another problem that black box abstractions do not benefit much, not just because one has to face the complexity of the detailed implementation, but also because the detail is indeed a macro definition in LISP and not implemented in the SAL programming language, which is provided for users who may have difficulty with LISP programming.

Thus, even though the Figure 4.23 example in Nyquist seems much terser and simpler in its appearance, such black-box abstraction can be helpful when the functions can provide exactly what users want, but may not be beneficial for exploratory design and exploratory understanding. Considering the accessibility to the detail of the algorithms hidden inside the black boxes, black-box abstractions may be more harmful for creative exploration and lead to the similar problems exhibited when microsound

¹Nyquist documents by Dannenberg clearly state that *seqrep* combines iteration and scheduling. In (92), Dannenberg describes “the *seq* function is used to invoke a sequence of behaviours. Each note is started at the time the previous note finishes” and “the *seqrep* “function” works like *seq* except that it creates copies of a sound by evaluating an expression multiple times”. Thus, the *seqrep* function integrates both features of looping and scheduling.

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

```
01: function ws-harm-dist()
02: begin
03:   with wavesets = extract-wavesets("sound.wav")
04:   return seqrep(i, length(wavesets), harm-dist(wavesets[i]))
05: end
06: function harm-dist(grain) ; return grain + 2 copies up 1 octave
07:   return cue(grain) + seqrep(j, 2, sound(grain) ~ 0.5)
08: play ws-harm-dist()
```

Figure 4.23: A waveset harmonic distortion example in Nyquist’s SAL programming language.

synthesis techniques are encapsulated within built-in unit-generators. It is also important in end-user programming to provide the language/library design that requires less environment-specific knowledge and less cognitive resources in programming and comprehension; thus, it is more desirable to provide a better abstraction of the sound synthesis framework, which can appropriately express microsound synthesis techniques without involving such complexity.

4.2.3.3 Microsound objects and manipulations in the existing computer music languages

The previous sections briefly reviewed the difficulty with microsound synthesis programming in the unit-generator languages and the reasons the black-box abstraction of the complexity of the implementation does not benefit users for exploratory design and exploratory understanding in computer music programming.

However, while the language design of LC intends to reduce this difficulty by providing a sound synthesis framework that directly integrates objects and manipulations for microsound synthesis, some previous works also seem to provide programming models that are different from the unit-generator concept. For example, in Section 2.3, the software synthesis framework design by Bencina (36) and Chronic computer music language by Brandt (56) were discussed as the examples that consider the abstraction of microsounds in the framework design. While Bencina’s work is about the software design mainly targeting granular synthesizer software design and not appropriate to be discussed in the context of computer music language design, Brandt’s work clearly targets the language design. Some other works also exhibit alternative programming

4.2 Comparing LC with the existing computer music languages

```

01: (defmacro seqrep (pair sound)
02:   '(let ((,(car pair) 0)
03:         (loop%count ,(cadr pair))
04:         (nyq%environment (nyq:the-environment))
05:         seqrep%closure first%sound s%rate)
06:     ; note: s%rate will tell whether we want a single or multichannel
07:     ; sound, and what the sample rates should be.
08:     (cond ((not (integerp loop%count))
09:            (error "bad argument type" loop%count))
10:           (t
11:            (setf seqrep%closure #'(lambda (t0)
12: ;      (display "SEQREP" loop%count ,(car pair))
13:              (cond ((< ,(car pair) loop%count)
14:                    (setf first%sound
15:                          (with%environment nyq%environment
16:                            (at-abs t0 ,sound)))
17:                    ; (display "seqrep" s%rate nyq%environment ,(car pair))
18:                    ;      loop%count)
19:                    (if s%rate
20:                      (setf first%sound (force-srates s%rate first%sound))
21:                      (setf s%rate (get-srates first%sound)))
22:                    (setf ,(car pair) (1+ ,(car pair)))
23:                    ; note the following test is AFTER the counter increment
24:                    (cond ((= ,(car pair) loop%count)
25:                          ;      (display "seqrep:  computed the last sound at"
26:                          ;      ,(car pair) loop%count
27:                          ;      (local-to-global 0))
28:                          first%sound) ;last sound
29:                          ((arrayp s%rate)
30:                          (display "seqrep:  calling snd-multiseq at"
31:                          ,(car pair) loop%count (local-to-global 0)
32:                          (snd-t0 (aref first%sound 0)))
33:                          (snd-multiseq (prog1 first%sound
34:                                          (setf first%sound nil))
35:                                          seqrep%closure))
36:                          (t
37:                          ;      (display "seqrep:  calling snd-seq at"
38:                          ;      ,(car pair) loop%count (local-to-global 0)
39:                          ;      (snd-t0 first%sound))
40:                          (snd-seq (prog1 first%sound
41:                                      (setf first%sound nil))
42:                                      seqrep%closure))))
43:            (t (snd-zero (warp-time *WARP*) *sound-srate*))))))
44:     (funcall seqrep%closure (local-to-global 0))))))

```

Figure 4.24: A macro definition of *seqrep* in Nyquist (open source distribution) - Copyright (c) 2000-2002, by Roger B. Dannenberg.

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

models, even though they are neither clearly targeting microsound synthesis nor interactive computer music applications. The following paragraphs briefly review such languages for better comparison with LC and other computer music languages and systems.

Chronic. As already described in Section 2.3, Chronic clearly takes microsound synthesis techniques into account in its design and tries to provide better flexibility by introducing what Brandt calls *temporal type constructors*. Brandt also argues for the concept of ‘unit-generators’ as “black-box primitives” in computer music language design and that “if a desired operation is not present, and cannot be represented as a composition of primitives, it cannot be realized within the language” and one of the benefits in Chronic is to provide expressibility without black box abstraction (56, pp.4-5).

However, Brandt also clearly states that in Chronic’s programming model, the use of *temporal type constructors* for microsound synthesis only targets non real-time sound synthesis and requires further consideration for real-time sound synthesis, as it has ‘an open problem’ in causality, in that Chronic’s programming model accepts both forward dependency and backward dependency (56, p.77).; While Chronic is an interesting example to integrate objects for microsound in its sound synthesis framework design, it fails to provide an appropriate abstraction for interactive music systems.

Chuck’s unit analyzer. While Chuck’s unit analyzer objects are not designed for microsound synthesis in general and the target domain is limited for audio analysis and spectral processing, we briefly review Chuck’s unit analyzer objects as readers may find some similarity with the design of FFT/IFFT objects in LC’s sound synthesis framework. Figure 4.25 describes a simple FFT-based cross synthesizer example in Chuck. The unit analyzer, *FFT* and *IFFT* are used in this example. As shown in this example, the *FFT* unit-analyzer can perform FFT for the given audio data and the resulting spectral data can be extracted as an array of complex numbers, which can be processed to perform the desired operation (as seen on line 27). The resulting data after processing can be converted back to the audio signal by performing IFFT by the *IFFT* unit-analyser.

4.2 Comparing LC with the existing computer music languages

The buffering of the sound data, windowing, overlap-add are performed internally within the unit-analyzer object. Figure 4.27 pictorially describes the underlying mechanism in ChuckK’s unit analyzer framework.

As in Brandt’s Chronic, Wang et al. also argue that the avoidance of black-box abstraction of the low level details as a feature of ChuckK’s unit analyzer design. They describe that “the high-level abstractions in the system should expose essential low-level parameters while doing away with syntactic overhead, thereby providing a highly flexible and open framework that can be easily used for a variety of tasks” in (316).

While such a problem of accessibility with less abstraction barriers seem to be successful to a certain degree, however, there appears to be a design problem in the unit analyzers, as they must work within ChuckK’s sound synthesis framework; in ChuckK’s sound synthesis framework, no unit generators can process inputs or produces outputs without the advance of logical synchronous time, which is globally shared within the system. Likewise, the behaviour of the unit analyzers is also synchronized with the logical time. This makes it harder to apply the different hop-sizes between FFT and IFFT objects.

Figure 4.26 describes a simple example of the cross synthesis to apply different hop-size to the source for the formant, which seems to be a little more complicated than the Figure 4.25 example. As the hop size for the output is half the frame size, it is necessary to involve two pairs of a SndBuf unit-generator and an FFT unit-analyzer under ChuckK’s sound synthesis framework, in which an FFT object can be fed its input samples only when the logical time advances. Moreover, the reading position from the sound buffers must be updated so to provide the samples from the formant source with the correct hop size. If the hop size of the output is set to 1/4 of the frame size, it is required to modify the code to involve four pairs of a SndBuf unit-generator and an FFT unit-analyzer, since four output frames will overlap. To experiment with various input/output hopsizes, one may need to write more complicated code for generalization; such a problem can be a significant obstacle to express more sophisticate audio analysis and spectral processing algorithms in ChuckK.

Matlab and Octave. Matlab (138) and Octave (105)¹ are designed for numerical computing in general, yet they can also perform sound synthesis and analysis. Both

¹Octave is open source software, which has a considerable compatibility with Matlab.

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

```
01: //loading the sound files to the buffer objects.
02: "/sound/violin.wav" => string filename1;
03: "/sound/kill_humans.wav" => string filename2;
04: SndBuf source1;
05: SndBuf source2;
06: filename1 => source1.read;
07: filename2 => source2.read;
08:
09: //build a synthesis graph.
10: source1 => FFT fft1 => blackhole;
11: source2 => FFT fft2 => blackhole;
12: IFFT ifft => dac;
13:
14: //set up FFT parameters.
15: 1024 => fft1.size => fft2.size => ifft.size => int FFT_SIZE;
16: FFT_SIZE / 2 => int HOP_SIZE;
17:
18: Windowing.hann(FFT_SIZE) => fft1.window;
19: Windowing.hann(FFT_SIZE) => fft2.window;
20: Windowing.hann(FFT_SIZE) => ifft.window;
21: //to store the cross synthesis result.
22: complex Z[FFT_SIZE / 2];
23:
24: //main loop.
25: while(true){
26:     //perform FFT for two inputs.
27:     fft1.upchuck();
28:     fft2.upchuck();
29:     //cross synthesis.
30:     for (int i; i < fft1.size() / 2; i++){
31:         fft1.cval(i) $ polar => polar a;
32:         fft2.cval(i) $ polar => polar b;
33:         %(a.mag * b.mag, a.phase) => polar c;
34:         c $ complex => Z[i];
35:     }
36:     //perform IFFT.
37:     ifft.transform(Z);
38:     //sleep until the next frame.
39:     HOP_SIZE::samp => now;
40: }
```

Figure 4.25: A simple FFT-based cross synthesizer example.

4.2 Comparing LC with the existing computer music languages

```
01: //loading the sound files to the buffer objects.
02: "/sound/violin.wav" => string filename1;
03: "/sound/kill_humans.wav" => string filename2;
04: SndBuf source1;
05: SndBuf source2a;
06: SndBuf source2b;
07: filename1 => source1.read;
08: filename2 => source2a.read;
09: filename2 => source2b.read;
10:
11: //build a synthesis graph.
12: source1 => FFT fft1 => blackhole;
13: source2a => FFT fft2a => blackhole;
14: source2b => FFT fft2b => blackhole;
15: IFFT ifft => dac;
16:
17: //set up FFT parameters.
18: 1024 => fft1.size => fft2a.size => fft2b.size => ifft.size => int FFT_SIZE;
19: FFT_SIZE / 2 => int HOP_SIZE_OUT;
20: FFT_SIZE / 4 => int HOP_SIZE_IN ;
21: Windowing.hann(1024) => fft1.window;
22: Windowing.hann(1024) => fft2a.window;
23: Windowing.hann(1024) => fft2b.window;
24: Windowing.hann(1024) => ifft.window;
25: //to store the cross synthesis result.
26: complex Z[FFT_SIZE / 2];
27:
28: //feed the input samples for the first two frames.
29: 0 => int posA;
30: HOP_SIZE_IN => int posB;
31: ///after 512 samples, we need to reset the read position for source2B
32: FFT_SIZE::samp / 2 +=> now;
33: posB => source2b.pos;
34: FFT_SIZE::samp / 2 +=> now;
35:
36: //now, we are ready to start cross synthesis for the first frame.
37: while(true){
38:     //first, process source2a
39:     fft1.upchuck();
40:     fft2a.upchuck();
```

Figure 4.26: A simple FFT-based cross synthesizer example (using the different hop sizes for the source a and source b).

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

```
41:   for(int i; i < fft1.size() / 2; i++){
42:       fft1.cval(i) $ polar => polar a;
43:       fft2a.cval(i)$ polar => polar b;
44:       %(a.mag * b.mag, a.phase) => polar c;
45:       c $ complex => Z[i];
46:   }
47:
48:   ifft.transform(Z);
49:
50:   //update the reading position for the next frame from source 2a.
51:   HOP_SIZE.IN * 2 +=> posA;
52:   posA => source2a.pos;
53:
54:   //sleep until the next output frame.
55:   HOP_SIZE.OUT::samp => now;
56:
57:   //now the source 2b is ready.
58:   fft1.upchuck();
59:   fft2b.upchuck();
60:
61:   for (int i; i < fft1.size() / 2; i++){
62:       fft1.cval(i) $ polar => polar a;
63:       fft2b.cval(i) $ polar => polar b;
64:       %(a.mag * b.mag, a.phase) => polar c;
65:       c $ complex => Z[i];
66:   }
67:
68:   ifft.transform(Z);
69:
70:   //update the reading position for the next frame from source 2b.
71:   HOP_SIZE.IN * 2 +=> posB;
72:   posB => source2b.pos;
73:
74:   HOP_SIZE.OUT::samp => now;
75: }
```

Figure 4.26: A simple FFT-based cross synthesizer example (using the different hop sizes for the source a and source b) (continued).

4.2 Comparing LC with the existing computer music languages

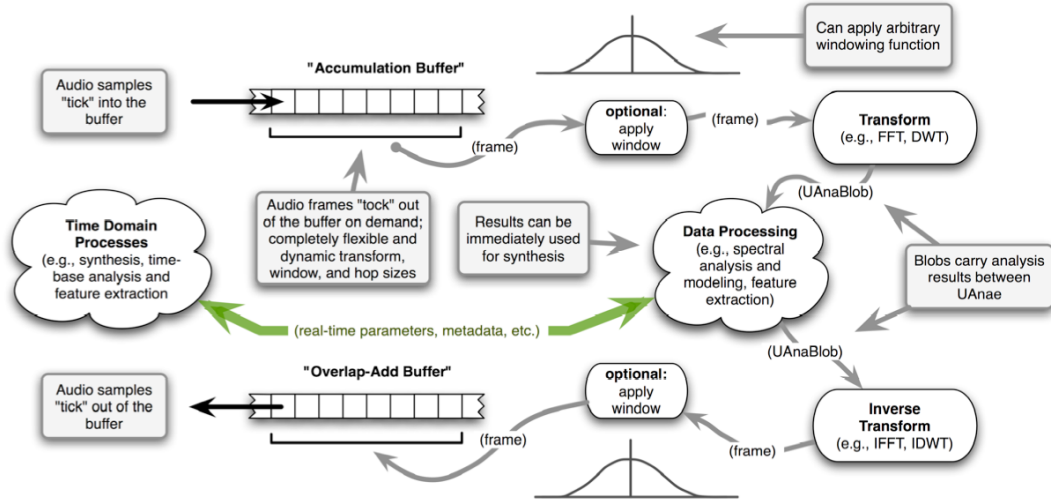


Figure 4.27: Underlying pipeline of a generic hybrid synthesis/analysis system in Chuck audio programming language - (taken from the publication ‘Combining Analysis and Synthesis in the Chuck Programming Language’ (316)).

provide a number of library functions for signal processing, which are useful for the research and experiments in sound synthesis and analysis. As the sample data is expressed as a matrix, it permits direct access to each sample in the data. Such language design may satisfy the capability of accessing to low-level data as Wang and Brandt discussed in the design criteria for Chuck’s unit analyzer and Chronic. Figure 4.28 shows the examples in Matlab of simple sound synthesis and low pass filtering (205).

However, as Matlab and Octave are not designed specially for computer music programming, they do not offer the high-level abstractions or software frameworks/libraries for domain-specific needs of computer music as seen in many computer music languages, such as the unit-generator concept and task-scheduling, real-time sound synthesis and interactivity; hence, these languages are not particularly beneficial to computer music programming when developing real-time interactive computer music systems, while both are useful for the research and experiments in audio processing algorithms.

4.2.3.4 The benefits of LC’s language design

Abstraction inversion. Based on the assessment that the difficulty in microsound synthesis programming in the existing computer music languages is caused by abstraction inversion due to the lack of the objects and functions that directly represent mi-

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

Generating 1.5 sec of a 50 Hz sawtooth and square wave with a sample-rate of 10k Hz

```
01: fs = 10000;
02: t = 0:1/fs:1.5;
03: x1 = sawtooth(2*pi*50*t);
04: x2 = square(2*pi*50*t);
05: subplot(211),plot(t,x1), axis([0 0.2 -1.2 1.2])
06: xlabel('Time (sec)');ylabel('Amplitude'); title('Sawtooth Periodic Wave')
07: subplot(212),plot(t,x2), axis([0 0.2 -1.2 1.2])
08: xlabel('Time (sec)');ylabel('Amplitude'); title('Square Periodic Wave')
```

applying a lowpass FIR filter to a noise waveform

```
01: Fs = 500; % sample rate in Hz
02: N = 500; % number of signal samples
03: rng default;
04: x = ecg(N)'+0.25*randn(N,1); % noisy waveform
05: t = (0:N-1)/Fs; % time vector
06:
07: % Design a 70th order lowpass FIR filter with cutoff frequency of 75 Hz.
08: b = fir1(70,75/(Fs/2));
09:
10: grpdelay(b,1,2048,Fs) % plot group delay
11: D = mean(grpdelay(b,1)) % filter delay in samples
12:
13: y = filter(b,1,[x; zeros(D,1)]); % Append D zeros to the input data
14: y = y(D+1:end); % Shift data to compensate for delay
15:
16: figure
17: plot(t,x,t,y,'r','linewidth',1.5);
18: title('Filtered Waveforms');
19: xlabel('Time (s)')
20: legend('Original Noisy Signal','Filtered Signal');
21: grid on
22: axis tight
```

Figure 4.28: A sound synthesis and lowpass filter example in Matlab (205).

4.2 Comparing LC with the existing computer music languages

crossounds and the related manipulations, LC's sound synthesis framework is designed to directly integrate such objects and functions. LC also provides simple and means to schedule microsounds with sample-rate accuracy. While strongly-timed programming makes it easy to control the advance of logical time, the feature of start-time constraint makes it a lot easy to schedule microsounds in the future without any advance of the time and other library functions such as *WriteDAC* and *PanOut* also has the offset to the timing when the *Samples* object is actually played. Thus, abstraction inversion can be avoided. Such features in LC contribute to describe certain microsound synthesis techniques, e.g. probabilistic scheduling of grains, tersely, compared to other languages.

For instance, the examples given in Section 3.2.3 are simple compared to the above examples in the other languages. Waveset harmonic distortion (Figure 4.29) and synchronous granular synthesis (Figure 4.30) in LC are much terse and simple in comparison with the examples in SuperCollider and ChuckK (Figure 4.19 and Figure 4.20). Applying sound effects to the entire output of microsound synthesis can be also easily expressed as shown in Figure 4.31 compared to the similar example in SuperCollider 4.21.

Less black box abstractions. In the previous section, it was discussed that black box abstractions do not benefit when one intends to explore what is abstracted. Abstracting the implementation detail of a microsound synthesis technique within a function is not much meaningful for users as it is nothing but the detailed implementations that they want to explore. As users also must face the abstraction barriers between the layers of abstractions, it can be some more harmful for exploratory design and exploratory understanding as discussed with the waveset harmonic distortion example in Nyquist.

On the other hand, the Figure 4.32 example in LC, such black-box abstraction is avoided as possible. While the *ExtractWavesets* function in this example seems alike to *extract-wavesets* in the Figure 4.23 example, the array returned from the function consists of *Samples* objects, each of which contain sample values for a single waveset. Every single sample within a *Samples* object can be directly accessed, together with

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

```
01: LoadSndFile(0, "/sound/sample1.aif");
02: var wavesets = ExtractWavesets(0);
03:
04: //weights for 2nd and 3rd harmonics
05: var weight1 = 0.5;
06: var weight2 = 0.5;
07:
08: for (var i = 0; i < wavesets.size; i +=1 ){
09:     //create the 2nd and 3rd harmonics from the original waveset.
10:     var ws = wavesets[i];
11:     var harm1 = ws->resample(ws.size / 2)->amplify(weight1);
12:     var harm2 = ws->resample(ws.size / 3)->amplify(weight2);
13:
14:     //schedule each waveset
15:     PanOut(ws, 0.0);
16:     //put 2nd harmonics that overlap-add the original
17:     PanOut(harm1, -1.0);
18:     PanOut(harm1, -1.0, offset:harm1.dur);
19:     //3rd harmonics
20:     PanOut(harm2, 1.0);
21:     PanOut(harm2, 1.0, offset:harm2.dur);
22:     PanOut(harm2, 1.0, offset:harm2.dur * 2);
23:
24:     now += ws.dur;
25: }
```

Figure 4.29: A waveset harmonic distortion example in LC (reproduced from Figure 3.46).

4.2 Comparing LC with the existing computer music languages

A synchronous granular synthesis example

```
01: //create a SampleBuffer and fill it with 256 samp sinewave * 4 cycles
02: var sbuf = new SampleBuffer(1024);
03: for (var i = 0; i < sbuf.size; i += 1){
04:   sbuf[i] = Sin(3.14159265359 * 2 * (i * 4.0 / sbuf.size));
05: }
06:
07: //create a grain. apply an envelope to sinewave and resample it to 440 samples
08: var tmp = sbuf->toSamples();
09: var win = GenWindow(1024::samp, \hanning);
10:
11: var grain = tmp->applyEnv(win)->resample(440)->amplify(0.25);
12:
13: within(5::second){
14:   while(true){
15:     WriteDAC(grain);
16:     now += win.dur / 4;
17:   }
18: }
```

Figure 4.30: A synchronous granular synthesis example (reproduced from Figure 3.36).

useful methods to perform typical manipulations on the samples¹.

As scheduling of each waveset for the output can be performed quite simply in LC and the explicit control of logical time is provided in the strongly-timed programming concept, the entire sound synthesis algorithm of waveset harmonic distortion is described without hiding the details under the abstraction barrier. Hence, modification of a program can be easier; for instance, to perform waveset distortion, one only has to multiply a *Samples* object with itself² and the maximum amplitude and length of each waveset, which are required to perform waveset substitution can be easily obtained by accessing *size* of a *Samples* object and examining the samples within it directly³.

Microsound objects and manipulations. It was also reviewed several computer music languages (and non computer music languages) with the language design that

¹On the contrary, in the Figure 4.23 example in Nyquist, the elements in the array returned from *extract-wavesets* function encapsulates the sound synthesis algorithms by box abstraction.

²For instance, the waveset distortion can be performed by ‘var distort = ws- >mul(ws);’ where ‘ws’ is the original waveset.

³A method ‘maxAmp()’ is also provided to obtain the maximum amplitude, while one can iterate each sample by for-loop to obtain the maximum amplitude.

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

```
01: //loading the sound file onto the buffer.
02: LoadSndFile(0, "/sound/sample1.aif");
03:
04: //Freeverb is a reverbrator unit-generator.
05: var reverb = patch {
06:   defin: Freeverb~(damp:0.2, fb1:0.58, fb2:0.2, spread:200) => defout: DAC();
07: };
08:
09: //play the sound for 20 second.
10: var entireDur = 20::second;
11: within(entireDur){
12:   //perform granular-sampling time-stretching.
13:   var pos = 0::second;
14:   var dur = 512::samp;
15:   var win = GenWindow(512::samp, \hanning);
16:
17:   var overlap = 2;
18:   var advance = dur / overlap;
19:
20:   while(true){
21:
22:     var sample = ReadBuf(0, dur, offset:pos);
23:     var grain = sample->applyEnv(win);
24:     //write it to the reverbrator's default input (defin).
25:     reverb->write(grain);
26:     var out = reverb->pread(advance); //read from 'defout' outlet.
27:     WriteDAC(out);
28:
29:     pos += dur / 4;
30:     now += advance;
31:   }
32: }
```

Figure 4.31: A reverberation example (2) (reproduced from Figure 3.61).

4.2 Comparing LC with the existing computer music languages

```
01: LoadSndFile(0, "sound.wav");
02: var wavesets = ExtractWavesets(0);
03: for (var i = 0; i < wavesets.length; i+= 1){
04:     var orig = wavesets[i];
05:     var octup = orig->resample(orig.size / 2);
06:     WriteDAC(orig);
07:     WriteDAC(octup);
08:     WriteDAC(octup, offset:octup.dur);
09:     now += orig.dur;
10: }
```

Figure 4.32: A waveset harmonic distortion example in LC (equivalent to Figure 4.23 Nyquist exmaple).

can model microsounds without the unit-generators. Yet, none of Chronic, Matlab and Octave provide an appropriate abstraction for real-time sound synthesis in interactive music systems. While ChuckK is designed for interactive music applications, its unit-analyzers are designed solely for audio analysis and processing in frequency domain and lacks generality to apply to microsound synthesis. ChuckK also exhibits the problem that the analysis and processing must be synchronized with the advance of logical time and it is made a lot difficult to use the different hopsizes for FFT and IFFT objects.

To the contrary, LC's microsound synthesis framework is designed for real-time sound synthesis and interactive music applications and exhibits no problem with causality as seen in Chronic and LC's synthesis framework can easily collaborate with the other features in LC (such as the unit-generators, scheduling functions, light-weight concurrency and the like). While Matlab and Octave are useful to explore non real-time sound synthesis program, generally speaking, they do not provide such domain-specific features for interactive computer music.

In ChuckK, the FFT/IFFT objects must be employed inside the unit-generator graph and it is necessary to advance the logical time, to feed the input/output samples to perform FFT/IFFT. This leads to the unnecessary complexity in the implementation when the hopsizes differ between the sound source and the output as seen in Figure 4.26. In contrast, LC's *Samples* objects and library functions are basically independent from the unit-generators and the advance of the logical time. This leads to much simpler implementation even when the hopsizes differ between the sound source and the output

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

as shown in Figure 4.33. As shown, simply changing the parameters on line 07 and 08 is enough to change the hopsizes both for the input and the output. In addition, as the input samples for FFT can be directly obtained without the advance of logical time, there is no latency in the output as is seen in ChuckK’s example, in which the advance of logical time must be involved to feed the input samples to FFT unit-analyzers.

Even when a user needs to handle real-time input, it can easily be handled by changing the *ReadBuf* function call to *ReadADC* function call. Moreover, unlike ChuckK’s unit-analyzers, which are designed only for spectral processing and audio analysis in frequency domain, LC’s sound synthesis framework is more general in that its programming model integrates both spectral processing and microsound synthesis techniques within the same framework.

Moreover, as LC is still equipped with traditional unit-generators, the sound synthesis framework in LC also provides the capability for the collaboration between unit-generators and microsound objects in LC. As seen in Figure 4.34 and Figure 4.35, the output from a unit-generator (and from a patch) can be obtained as a *Samples* object, even without the advance of logical time, to represent a microsound in LC. Any *Samples* object can be given as an input signal to a unit-generator (and a patch); thus, LC’s sound synthesis framework design allows mutual collaboration between two different abstractions for sound synthesis.

The benefits of LC’s language design. As described above, LC’s sound synthesis framework directly integrates objects and manipulations for microsound synthesis. Such a design contributes to reducing the difficulty in microsound synthesis programming, which is caused by the software anti-pattern of abstraction inversion as seen in many other computer music languages that solely depend on the unit-generator concept.

The reasons black-box abstraction of the detail of the complicated implementations do not reduce the difficulty in microsound synthesis programming are also discussed. As it is the detail of the implementation that users need to investigate for exploratory design and exploratory understanding in microsound synthesis, providing a ready-made black-box abstraction (e.g., a library function), for a certain microsound synthesis tech-

4.2 Comparing LC with the existing computer music languages

```
01: //load sound files
02: LoadSndFile(0, "/LCSynth/violin.wav");
03: LoadSndFile(1, "/LCSynth/kill_humans.wav");
04:
05: //FFT/IFFT parameters.
06: var dur = 1024::samp;
07: var hop_size_in = dur / 4;
08: var hop_size_out = dur / 2;
09:
10: //perform cross synthesis
11: var i = 0;
12: while(true){
13:   var src1 = ReadBuf(0, dur, offset:i * hop_size_out);
14:   var src2 = ReadBuf(1, dur, offset:i * hop_size_in);
15:
16:   var pfft1 = PFFT(src1, \hanning);
17:   var pfft2 = PFFT(src2, \hanning);
18:
19:   var ppved = pfft1[0]->mul(pfft2[0]);
20:   var pifft = PIFFT(ppved, pfft1[1], \hanning);
21:
22:   PanOut(pifft);
23:   i += 1;
24:   now += hop_size_out;
25: }
```

Figure 4.33: Another cross synthesis example in LC (with the variable hopsizes).

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

```
01: //instantiate a white noise generator
02: var src = new Noise~();
03:
04: //we want to keep the latest 1 second output.
05: src.retain = 1::second;
06:
07: //process for 1 second, without the advance of logical time.
08: src->process(1::second);
09:
10: //get the last 1 second output.
11: var out = src->read(1::second);
12:
13: //set it to the sound output.
14: PanOut(out);
15: now += 1::second;
16:
17: //instantiate a sine wave oscillator.
18: src = new Sin~(440);
19:
20: //we want to keep the last 50 msec output.
21: src.retain = 50::ms;
22:
23: //play a sine wave of 440 Hz.
24: within(5::second){
25:   while(true){
26:     //process for 50 msec, without the advance of logical time.
27:     //then read the last 50msec output and write to DAC.
28:     src->process(50::ms);
29:     var out = src->read(50::ms);
30:
31:     //set it to the sound output.
32:     PanOut(out);
33:     now += out.dur;
34:   }
35: }
36: //change the frequency every 50 msec.
37: while(true){
38:   src->process(50::ms);
39:   var out = src->read(50::ms, \out);
40:   PanOut(out);
41:   //update the frequency.
42:   src.freq = Rand(1,8) * 440;
43:   now += out.dur;
44: }
```

Figure 4.34: An example to create Sample objects from the unit-generator's output samples (1) (reproduced from Figure 3.54).

4.2 Comparing LC with the existing computer music languages

```
01: //load a sound file and extract wavesets.
02: LoadSndFile(0, "/sound/sample1.aif");
03: var wavesets = ExtractWavesets(0);
04: //below function performs waveset harmonic distortion.
05: //and write out the output to the given 'stream'
06: var f = function(var stream, entireDur){
07:   var weight1 = 0.5;
08:   var weight2 = 0.5;
09:   //perform waveset harmonic distortion.
10:   within(entireDur){
11:     for (var i = 0; i < wavesets.size; i+= 1){
12:       var ws = wavesets[i];
13:       var harm1 = ws->resample(ws.size / 2)->amplify(weight1);
14:       var harm2 = ws->resample(ws.size / 3)->amplify(weight2);
15:
16:       stream->write(ws);
17:       stream->write(harm1);
18:       stream->write(harm1, offset:harm1.dur);
19:       stream->write(harm2);
20:       stream->write(harm2, offset:harm2.dur);
21:       stream->write(harm2, offset:harm2.dur * 2);
22:
23:       var out = stream->pread(ws.dur);
24:       PanOut(out);
25:       now += ws.dur;
26:     }
27:   }
28: };
29: //apply a triangular envelope.
30: var entireDur = 2::second;
31: var envelope = new Line~([\reset, 0, 1.0, entireDur / 2, 0.0, entireDur / 2]);
32: envelope->trigger();
33: //pass it to the waveset harmonic distortion function.
34: f(envelope, entireDur);
35: //apply an triangular envelope + reverberation.
36: var p = patch {
37:   defin:Line~([\reset, 0, 1.0, entireDur / 2, 0.0, entireDur / 2]) =>
38:   Freeverb~(damp:0.2, fb1:0.58, fb2:0.2, spread:200) => defout:DAC~();
39: };
40: p.defin->trigger();
41: f(p, entireDur);
```

Figure 4.35: A duck-typing example to apply an envelope (by a unit-generator) and an envelope + reverberation (by a patch) to the output of waveset harmonic distortion (reproduced from Figure 3.63).

4. DISCUSSION: THE NECESSITY FOR THE DEVELOPMENT OF LC AS A NEW LANGUAGE AND THE BENEFITS OF ITS LANGUAGE DESIGN

nique is not helpful in such a situation and can even be an obstacle since the detail can be inaccessible in the presence of an abstraction barrier.

LC's sound synthesis framework provides fine-grained manipulations of microsound objects and low-level access to samples within a microsound object to facilitate exploratory design and exploratory understanding.

While LC is not the first language that argues such an issue of the integration of microsound objects and low-level accessibility, and there exists the predecessor languages, which considered this issue as seen in Chronic and ChucK, none of the languages discussed above provides an appropriate programming model required for interactive music systems with real-time sound synthesis capability, which is applicable to microsound synthesis techniques with more generalization. Moreover, LC's sound synthesis framework also considers the mutual collaboration between the traditional unit-generator concept and its microsound synthesis abstraction.

Such features of LC's abstraction for microsound synthesis can provide a terse and expressive programming model for various microsound synthesis techniques, allowing users to explore the domain of microsounds with considerable flexibility, which is not supported by other existing computer music languages to the same degree.

5

Conclusion and Future Work

This chapter first summarises this thesis and then clarifies the contributions made through the development of LC in the conclusion section. Then, the direction of further research, based on the knowledge and observation acquired through this work, is discussed in the future work section.

5.1 Conclusion

5.1.1 Problems

This thesis first described the three problems in today's computer music language design, which hinder creative exploration in artistic practices of our time. These were discussed with the detail in Chapter 2. The following is summary of these three problems.

The insufficient support for dynamic modification of a computer music system. As the process of musical creation is essentially exploratory and experimental, a computer music language should be appropriately designed for rapid-prototyping. At the same time, recent computer music practices involve a considerable degree of dynamism in run-time as seen in live-coding and dynamic-patching. To facilitate such programming activities, it is ideal for a computer music language to support dynamic modification of a computer program at both levels of compositional algorithms and sound synthesis. Yet the existing computer music languages still exhibit problems in supporting dynamic modification and further improvement is still demanded.

5. CONCLUSION AND FUTURE WORK

The insufficient support for precise timing behaviour and other features with respect to time. Especially after the emergence of real-time computer music systems, timing precision has been a traditional issue in computer music. Yet, the timing precision required for today's computer music practices is significantly higher than that of the earlier decades. Generally speaking, microsound synthesis techniques even require sample-rate accuracy in scheduling each microsound for precise sound output, and such sample-rate accuracy is considered desirable in musical practices not just at the acoustic level, but also at the rhythmic level. While some of the recent computer music languages achieved sample-rate accuracy in timing behaviour by introducing the concept of synchronous programming, the temporary suspension of real-time DSP can be easily caused by a time-consuming task in these languages, as they are solely depending on the synchronous approach. Yet, computer music programs often involve such a time-consuming tasks. Furthermore, many recent computer music languages still lack even the features with respect to time that were supported in the earlier era when a computer music system was composed of a computer and external synthesizer hardware; computer music languages of our time should support such features with respect to time with sample-rate accurate timing behaviour.

The difficulty in microsound synthesis programming case by the anti-pattern of abstraction inversion. The implementation of a microsound synthesis technique in a unit-generator language often exhibits a considerable degree of complexity, even when the microsound synthesis technique to be performed is conceptually simple. It is desirable to reduce such complexity, to facilitate creative exploration in microsound synthesis by computer musicians. While some previous works also discuss this difficulty as a problem in the software architecture and language design, these works are targeting stand-alone software synthesizer applications and non real-time computer music language design and are hardly applicable to the design of an interactive computer music language with real-time sound synthesis. Thus, further research is still required to investigate more appropriate software design for computer music languages, which can reduce the difficulty in microsound synthesis programming.

5.1.2 Contribution

This thesis discusses the design and development of LC, a new computer music programming language, by addressing these three problems as a significant design opportunity¹. LC is designed as a mostly-strongly-timed prototype-based programming language that integrates objects and manipulations for microsound synthesis. The following describes the contributions made through the design and development of LC.

Better support for dynamic modification of a computer program. LC adopts the concept of prototype-based programming at both levels of compositional algorithms and sound synthesis for better support for dynamic modification. Generally speaking, prototype-based programming is beneficial to support the dynamic modification of compositional algorithms, as its programming concept is considerably flexible against the runtime modification of a program. While even recent computer music languages exhibit such problems as premature commitment and viscosity in modification, LC adopts prototype-based programming at both level of compositional algorithms and sound synthesis, and such a language design makes LC capable of the fine-grained dynamic modification of a unit-generator graph with considerable flexibility; thus, LC supports a considerable degree of dynamic modification with the terse and consistent programming model at both levels of compositional algorithms and sound synthesis.

Better support for precise timing behaviour and other features with respect to time. LC proposes and implements the mostly-strongly-timed programming concept, which extends the strongly-timed programming concept with the explicit switch between synchronous/non-preemptive context and asynchronous/preemptive context. Such context switching makes it possible to describe a time-consuming task as a background task, and the problem of temporary suspension of real-time DSP can be avoided while maintaining the feature of precise timing behaviour in strongly-timed programming. LC also takes desirable features with respect to time (e.g., execution time constraints and time fault tolerance) into consideration in its language design. These desirable features are integrated into the language design with sample-rate accuracy in timing behaviour in logical synchronous time.

¹Appendix II describes a brief history of computer music languages and systems with an emphasis on how computer music languages and systems have been developed through the synergy between artistic creativity and technological advancement, with more detail.

5. CONCLUSION AND FUTURE WORK

The reduction of the difficulty in microsound synthesis programming. This thesis addressed the difficulty in microsound synthesis programming as an issue of abstraction inversion, which occurs when the higher-level abstractions must be combined to express lower-level abstractions, and also discussed that the black box abstraction that hides the implementation details may reduce the complexity only at the surface level and do not benefit users for exploratory design and exploratory understanding, since it is such details that users have to comprehend and modify for further exploration. LC's underlying sound synthesis framework integrates the objects that directly represent microsounds and related manipulations on microsound objects, based on the assumption that the difficulty in microsound synthesis programming can be reduced by the removal of the abstraction inversion. The interoperability between the traditional unit-generator concept and LC's microsound abstractions is also considered. The resulting sound synthesis framework design makes LC highly expressive in microsound synthesis, and various techniques can be implemented in a terse and simple manner.

5.1.3 Conclusion

While design is an ill-defined problem, which “addresses complex issues and thus cannot easily be described in a concise, complete manner” (282), and many different solutions can be proposed for one problem, LC took the three problems as described above as the problem rooted in the design of a computer music language and its underlying sound synthesis framework.

Upon such perspective, through the design and development of LC, this thesis contributes to the solution to three problems in computer music programming. The means of the contributions are: (1) the adoption of the concept for general-purpose programming languages to a domain-specific problem (the adoption of prototype-based programming to a computer music language at both levels of compositional algorithm and sound synthesis), (2) the proposition of a new programming language concept (the mostly-strongly-timed programming concept, which extends the strongly-timed programming concept with explicit switch between synchronous context and asynchronous context) and (3) the novel approach to the sound synthesis framework design (the integration of the objects and manipulation for microsound synthesis in the sound synthesis framework).

As the problems found in creative practices lead to the contributions in computer music research, this thesis could be also viewed as an example of the synergy between creativity and technology, as is often observed in the history of computer music. Such an approach fostered through computer music history may have even more significance than before, since many domain-specific programming languages and environments are being developed to enhance artistic creativity in other digital art forms today.

5.2 Future Work

In this thesis, a new computer music programming language was designed and the proof-of-concept prototype was also implemented. While the prototype is fully functional and supports all the features described in this thesis, it is desirable to consider a more stable and efficient implementation, possibly with additional language features. While this thesis entirely focuses on the language design issues, the development of the proof-of-concept prototype also suggests that research on computer music language implementation is likely to lead to further research contributions. The following sections discuss such topics of interests for future work.

5.2.1 Language features.

While the current version of LC is already expressive, further improvement may be still be beneficial. For instance, LC still lacks such statements as the *switch-case* statement. The prototype version has no suffix increment/decrement (`'i++'` or `'i--'`), prefix increment/decrement (`'++i'` or `'--i'`), or ternary conditional (`'x > y ? x : y'`). The data type such as *tuple* as seen in Python (301) may be beneficial to describe a program a little more tersely. Pattern-matching as in Erlang (21) may also be desirable for the *receive* statement in LC.

Moreover, it is also desirable to develop the libraries for MIDI (23) and Open Sound Control protocol (116). Yet, these communication protocols can likely fit in LC's message passing model without much difficulty.

5.2.2 Performance efficiency.

The current prototype of LC was developed only to prove that its language design and concept are implementable and can run in real-time. While the resulting implemen-

5. CONCLUSION AND FUTURE WORK

tation is fully interactive and can perform real-time sound synthesis as expected, it is desirable to improve the performance efficiency. Generally speaking, the research on just-in-time compilation to native machine code at runtime may benefit all the issues in following paragraphs. The previous work on the application of just-in-time compilation to computer music languages can be seen in such works as Extempore's xtlang (277), LuaAV (273), and Kronos (217).

Performance efficiency of LC Virtual Machine. As the execution time of a computer music program is spent mostly for digital sound processing, the bytecode interpreter on LC Virtual Machine does not exhibit a significant problem in performance efficiency, even though the proof-of-concept version is not efficient. The techniques to improve the virtual machine performance are a traditional topic, and there would be not be much concern for the improvement. Sasada's thesis on the performance efficiency of his Ruby virtual machine provides a good survey of the related techniques (167).

Performance efficiency in sound synthesis: the unit-generators. The digital sound processing can be CPU intensive, and thus the improvement of DSP performance is a topic of interest. While the proof-of-concept prototype is capable of real-time sound synthesis by the unit-generators, as the implementation does not consider such performance issues, the addition of the features during the design and development process of the language seem to worsen the overall performance of the sound synthesis by the unit-generators, in comparison with the earlier versions of LC. However, as many unit-generator languages have been developed and some of them are released as open-source software, refactoring the existing code or developing a new version of LC can be easily improved, at least to the same degree by applying the implementation techniques in these predecessor languages.

Performance efficiency in the sound synthesis: LC's microsound objects and functions. While LC's microsound objects and functions are also implemented without consideration for performance efficiency, it was also observed that such an abstraction of microsound synthesis can benefit performance efficiency in some cases. As LC's *Samples* object is immutable, it can be easily reused without concern as to whether the samples within a *Samples* object can be changed, and the same object can

be scheduled multiple times with overlapping. Such overlapping of the same instance of the unit-generator graph is not possible as each unit-generator has its own running state (e.g., the current phase of a sine wave oscillator). Moreover, when the *Samples* objects are sent to the input of a unit-generator or to the sound output, it is only necessary to read the samples one-by-one from the internal buffer within a *Samples* object, whereas the unit-generator graph involves the computation of its output samples. Such characteristics seem to lead to better performance efficiency in microsound synthesis techniques, in which microsounds can be pre-generated and reused. For instance, synchronous granular synthesis technique is a synthesis technique of this kind. Other microsound synthesis techniques can also receive the same benefits if microsounds can be pre-generated.

However, a temporary suspension of real-time DSP was observed in the current prototype when the duration of a *Samples* object was too large, while such suspension has not been observed within the threshold of microsound time-scale¹, and microsound synthesis can be safely performed in real-time. For instance, if the duration of a *Samples* object is 30 seconds (= 1,323,000 samples under 44.1k Hz sample-rate), applying an envelope to this object can be time-consuming and the deadline for real-time DSP may be missed. One of the solutions is to apply the mostly-strongly-timed programming concept so that this computation can be performed as a background task; in this case, the logical time may be advanced during computation.

Yet, as LC's *Samples* object is immutable, it is possible to apply the concept of *lazy evaluation* for microsound synthesis without involving any modification to LC's language design. With lazy evaluation, a program "will not evaluate any expression unless its value is demanded by some other part of the computation" (18). By applying lazy evaluation, the manipulations on microsounds in LC on audio samples can be deferred until the values are used, for instance, until when the sound device requires the output samples. As the computation is performed only for the amount of the samples required, the computation can be divided into fairly small fragments, even

¹Roads describes the duration of each microsound extends between "the threshold of timbre perception (several hundred microseconds) up to the duration of short sound objects (~ 100 ms)" (242, p.21).

5. CONCLUSION AND FUTURE WORK

when processing a large number of samples. For instance, when the size of the audio vector for the sound output is set to 64 samples, it is necessary to compute only 64 samples in a *Samples* object at each DSP cycle and the rest of the samples in the *Samples* object can remain unevaluated; thus, the introduction of lazy evaluation for microsound synthesis is an interesting topic for the further research.

5.2.3 Garbage collection

The current version of LC implements a simple incremental mark-and-sweep garbage collection algorithm, known as Yuasa's snapshot-at-the-beginning algorithm (326). While it seems the pause time by the garbage collector in the prototype is fairly small, the further research would be desirable to investigate more appropriate garbage collection algorithms for computer music systems. While the significant body of the previous research has been done in the theory and implementation of garbage collection mechanism (156)(157), the research on the real-time garbage collectors still attracts significant interests. As computer music languages can require both less pause time and better throughput in garbage collection, the research on the real-time garbage collection mechanism in computer music also may have a potential to benefit the research area in general. While the recent progress of the garbage collection mechanism made even a hard real-time garbage collector realized, as seen in (267), yet, generally speaking, the behaviour of computer programs can significantly differ with application domains; there is still a necessity to investigate garbage collection mechanisms for what kind of garbage collection algorithms can be beneficial to each application domain.

Computer music programs may be particularly interesting in this sense, as a garbage collector should not block real-time DSP and audio output. The discussion on such issues of timing behaviour of a computer music program can be seen in previous works such as Metronome garbage collector (24) and ROLLENDURCHMESSERZEITSAMMLER garbage collector (200). Furthermore, as the behaviour of a strongly-timed program may exhibit its own characteristic in timing behaviour, it is desirable to investigate more suitable garbage collection algorithms for strongly-timed programs and mostly-strongly-timed programs. As some other multimedia applications can exhibit similar timing behaviour to a strongly-timed program, in which a program sleeps until a certain timing and must process the scheduled tasks as fast as possible to meet the deadline for

the output, such an investigation of the characteristics of computer music programs may benefit more general application domains of multimedia software; thus, the research on garbage collection in computer music applications is an interesting topic.

5. CONCLUSION AND FUTURE WORK

References

- [1] Pd documentation. 37
- [2] Max 8 api documentation. 2011. 37
- [3] Samuel Aaron, Alan F Blackwell, Richard Hoadley, and Tim Regan. A principled approach to developing new languages for live coding. In *Proceedings of New Interfaces for Musical Expression*, volume 2011, pages 381–386, 2011. 290
- [4] Curtis Abbott. The 4ced program. *Computer Music Journal*, 5(1):13–33, 1981. 275
- [5] Harold Abelson, G Sussman, and Julie Sussman. Structure and implementation of computer programs. *Cambridge, MA: MIT press*, 21:257–261, 1985. 210
- [6] Harold Abelson, RK Dybvig, CT Haynes, GJ Rozas, NI Adams IV, DP Friedman, E Kohlbecker, GL Steele Jr, DH Bartley, R Halstead, et al. Revised report on the algorithmic language scheme. *ACM SIGPLAN Lisp Pointers*, 4(3):1–55, 1991. 75
- [7] Chris Adamson, Mike Lee, and Kevin Avila. *Learning core audio: A hands-on guide to audio programming for Mac and iOS*. Addison-Wesley Professional, 2012. 287
- [8] HG Alles and Pepino di Giugno. A one-card 64 channel digital synthesizer. *Computer Music Journal*, 1(4):7–9, 1977. 12, 274
- [9] Jesse Allison and Christian Dell. Aural: A mobile interactive system for geo-locative audio synthesis. 2012. 291
- [10] Adam Alpern. Techniques for algorithmic composition of music. *On the web: <http://hamp.hampshire.edu/~adaF92/algocomp/algocomp95.html>*, 1995. 272
- [11] Xavier Amatriain. Clam: A framework for audio and music application development. *Software, IEEE*, 24(1): 82–85, 2007. 285
- [12] Xavier Amatriain, Pau Arumi, and David Garcia. A framework for efficient and rapid development of cross-platform audio applications. *Multimedia Systems*, 14 (1):15–32, 2008. 14
- [13] S.W. Amber. *Process patterns: Building large-scale systems using object technology*. Cambridge University Press, 1998. 42, 45
- [14] Akkihebbal L Ananda, BH Tay, and Eng-Kiat Koh. A survey of asynchronous remote procedure calls. *ACM SIGOPS Operating Systems Review*, 26(2):92–109, 1992. 201
- [15] David P. Anderson and R Kiuvila. Formula: A programming language for expressive computer music. *Computer*, 24(7):12–21, 1991. 197, 200, 282
- [16] David P Anderson and Ron Kuivila. A system for computer music performance. *ACM Transactions on Computer Systems (TOCS)*, 8(1):56–82, 1990. 20, 23, 40, 195, 197, 200, 282
- [17] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’89, pages 293–302, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75303. URL <http://doi.acm.org/10.1145/75277.75303>. 30
- [18] A.W. Appel. *Modern compiler implementation in JAVA*. Cambridge University Press, 2002. 241
- [19] Apple. Audio unit programming guide: Introduction. 2007. 189
- [20] Fabio Armani, Lorenza Bizzarri, E Favreau, and Andrea Paladin. Mars: Dsp environment and applications. In *Proceedings of the International Computer Music Conference*, pages 344–344. International Computer Music Association, 1992. 12, 279
- [21] Joe Armstrong. *Programming Erlang*. Pragmatic Bookshelf, 2007. 239, 315
- [22] Gérard Assayag and Andrew Gerzso. *New computational paradigms for computer music*. Delatour, 2009. 17
- [23] MIDI Manufacturers Association. The complete midi 1.0 detailed specification: Incorporating all recommended practices, 1996. 12, 23, 239, 274
- [24] Joshua Auerbach, David F Bacon, Florian Bömers, and Perry Cheng. Real-time music synthesis in java using the metronome garbage collector. In *Proceedings of the International Computer Music Conference, Copenhagen, Denmark*, 2007. 242
- [25] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003. 317
- [26] J. Backus. The history of fortran i, ii and iii. volume 1, pages 21–37, Piscataway, NJ, USA, July 1979. IEEE Educational Activities Department. doi: 10.1109/MAHC.1979.10013. URL <http://dx.doi.org/10.1109/MAHC.1979.10013>. 267
- [27] Henry G Baker. Critique of din kernel lisp definition version 1.2. *Lisp and Symbolic Computation*, 4(4):371–398, 1992. 306
- [28] T. Baker. Opening up ada-tasking. In *Proceedings of the 4th International Workshop on Real-time Ada Issues*, pages 60–64, 1990. xii, 5, 42, 45, 46, 59, 206, 303, 304, 305, 306, 307

REFERENCES

- [29] Michael Baldamus and Klaus Schneider. Extending estereel by asynchronous concurrency. Technical report, GI/GMM/ITG Fachtagung zum Entwurf Integrierter Schaltungen, 1999. 205
- [30] Jerry Banks and John S Carson. *Discrete-event system simulation*. Pearson Education India, 1984. 20
- [31] Daniel Bardou. Delegation as a sharing relation: Characterization and interpretation. In *Position Paper at the Workshop on Prototype-based Object-oriented Programming, ECOOP*, volume 96. Citeseer, 1990. 91
- [32] C. Barlow. Autobusk: An algorithmic real-time pitch and rhythm improvisation programme. In *Proceedings of the International Computer Music Conference*, pages 166–168. International Computer Music Association, 1990. 281
- [33] C. Barlow. *AUTOBUSK: a real-time pitch & rhythm generator*. University of Mainz, 2000. 281, 282
- [34] J Beauchamp. Music 4c introduction. *Computer Music Project, School of Music, University of Illinois at Urbana-Champaign*, 1(99):3, 1993. 267
- [35] James W Beauchamp. Music 4c, a multi-voiced synthesis program with instruments defined in c. *The Journal of the Acoustical Society of America*, 113:2215, 2003. 267
- [36] R. Bencina. *Audio Anecdotes III*, chapter Implementing real-time granular synthesis, pages 55–83. A.K Peters, 2006. 59, 60, 61, 64, 216
- [37] G. Berry, P. Couronne, and G. Gonthier. Synchronous programming of reactive systems: An introduction to estereel. In *Proceedings of the 1st Franco-Japanese Symposium on Programming of future generation computers*, pages 35–56, Amsterdam, The Netherlands, The Netherlands, 1988. Elsevier Science Publishers B. V. ISBN 0-444-70410-8. URL <http://dl.acm.org/citation.cfm?id=60661.60664>. xi, 33
- [38] Gérard Berry and Georges Gonthier. The estereel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2): 87–152, November 1992. ISSN 0167-6423. doi: 10.1016/0167-6423(92)90005-V. URL [http://dx.doi.org/10.1016/0167-6423\(92\)90005-V](http://dx.doi.org/10.1016/0167-6423(92)90005-V). 33
- [39] Gérard Berry, S Ramesh, and RK Shyamasundar. Communicating reactive processes. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 85–98. ACM, 1993. 205
- [40] Hugh Beyer and Karen Holtzblatt. *Contextual design: Defining customer-centered systems*. Access Online via Elsevier, 1997. 312
- [41] A.F. Blackwell and T.R.G. Green. *HCI models, theories and frameworks: Toward a multidisciplinary science*, chapter Notational systems - the cognitive dimensions of notation framework, pages 103–134. Morgan Kaufmann, 2003. 210, 259
- [42] A.F. Blackwell, L. Church, and T.R.G. Green. The abstract is an enemy: Alternative perspectives to computational thinking. In *Proceedings of the 2008 Psychology of Programming Interest Group Workshop*, 2008. 312
- [43] Alan Blackwell and Nick Collins. The programming language as a musical instrument. *Proceedings of PPIG05 (Psychology of Programming Interest Group)*, 2005. 3, 284, 289, 309
- [44] A. Blandford, T.R.G. Green, D. Furniss, and S. Makri. Evaluating system utility and conceptual fit using cassm. *International Journal of Human-Computer Studies*, 66:393–400, 2008. 259, 312
- [45] Ann Blandford and Thomas Green. From tasks to conceptual structures: Misfit analysis. 2001. 312
- [46] Ann Blandford, Thomas RG Green, and Iain Connell. Formalising an understanding of user-system misfits. In *Engineering Human Computer Interaction and Interactive Systems*, pages 253–270. Springer, 2005. 312
- [47] G Blaschek. Type-safe oop with prototypes: The concepts of omega. *Structured Programming*, 12(12):1–9, 1991. 86
- [48] Günther Blaschek. *Object-oriented programming with prototypes*. Springer-Verlag New York, Inc., 1994. 86
- [49] Daniel G Bobrow, Linda G DeMichiel, Richard P Gabriel, Sonya E Keene, Gregor Kiczales, and David A Moon. Common lisp object system specification. *ACM Sigplan Notices*, 23(SI):1–142, 1988. 290
- [50] Gregory Bollella and James Gosling. The real-time specification for java. *Computer*, 33(6):47–54, 2000. 33
- [51] Bert Bongers. An interview with sensorband. *Computer Music Journal*, 22(1):13–24, 1998. 284
- [52] R.C. Boulanger. *The Csound book: Perspectives in software synthesis, sound design, signal processing, and programming*. The MIT Press, 2000. xvii, 44, 267, 307, 310, 311
- [53] R.C. Boulanger and V. Lazzarini. *The Audio programming book*. The MIT Press, 2011. 19, 58, 65, 301
- [54] Pierre Boulez and Andrew Gerzso. Computers in music. *Scientific American*, 258(4):44–51, 1988. 275, 284
- [55] Lee Boynton. *Midi-lisp: A lisp-based music programming environment for the macintosh*. Ann Arbor, MI: MPublishing, University of Michigan Library, 1986. 281
- [56] E. Brandt. *Temporal type constructors for computer music programming*. PhD thesis, Carnegie Mellon University, 2008. ix, xii, 59, 61, 62, 63, 65, 216, 218, 317, 318
- [57] Eli Brandt. Implementing temporal type constructors for music programming. In *Proceedings of the 2001 International Computer Music Conference*, pages 99–102. Citeseer, 2001. 61
- [58] Peter Brinkmann, Peter Kirn, Richard Lawler, Chris McCormick, Martin Roth, and Hans-Christoph Steiner. Embedding pure data with libpd. In *Proceedings of the Pure Data Convention*, 2011. 291
- [59] Leo B Brodie. Forth inc., starting forth, 1986. 315
- [60] Benjamin M Brosgol, Ricardo J Hassan II, and Scott Robbins. Asynchronous transfer of control in the real-time specification for java. In *ACM SIGAda Ada Letters*, volume 22, pages 95–112. ACM, 2002. 115, 173, 205

REFERENCES

- [61] Andrew R Brown and Andrew Sorensen. Dynamic media arts programming in impromptu. In *Proceedings of the 6th ACM SIGCHI conference on Creativity & cognition*, pages 245–246. ACM, 2007. 16, 177
- [62] Andrew R Brown and Andrew Sorensen. Interacting with generative music through live coding. *Contemporary Music Review*, 28(1):17–29, 2009. 3, 289
- [63] Andrew R Brown and Andrew C Sorensen. aa-cell in practice: An approach to musical live coding. In *Proceedings of the International Computer Music Conference*, pages 292–299. International Computer Music Association, 2007. 287
- [64] W.J. Brown, C.R. Malveau, H.W. McCormick, and T.J. Mowbray. *Anti patterns: Refactoring software, architectures, and projects in crisis*. Wiley, 1998. 306
- [65] Phil Burk. Jsyn—a real-time synthesis api for java. In *Proceedings of the 1998 International Computer Music Conference*, pages 252–255. International Computer Music Association San Francisco, 1998. 24, 285
- [66] Alan Burns and Andrew J Wellings. *Real-time systems and programing languages: Ada 95, Real-time Java and Real-time Posix*. Addison Wesley, 2001. 33, 34, 107, 109, 110, 115
- [67] Alan Burns and Andy Wellings. *HRT-HOOD: A structured design method for hard real-time Ada systems*, volume 3. Elsevier, 1995. 33
- [68] William AS Buxton. A composer’s introduction to computer music. *Journal of New Music Research*, 6(2):57–71, 1977. 274
- [69] Nicholas Carrier and David Gelernter. Linda and friends. *Distributed shared memory: Concepts and systems*, 21:177, 1998. 109
- [70] Kim Cascone. The aesthetics of failure: ‘post-digital’ tendencies in contemporary computer music. *Computer Music Journal*, 24(4):12–18, 2000. 284
- [71] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A Plaice. Lustre: A declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188. ACM, 1987. 33
- [72] Sergio Cavaliere, Giuseppe Di Giugno, and Eugenio Guarino. Mars: The x20 device and sm100 board. In *Proceedings of the International Computer Music Conference*, pages 348–348. International Computer Music Association, 1992. 12, 279
- [73] Joel Chadabe and Roger Meyers. An introduction to the play program. *Computer Music Journal*, 2(1):12–18, 1978. 272
- [74] J Michael Clarke, PD Manning, R Berry, and A Purvis. Voxel: New implementations of the fof synthesis method. In *Proc. In. Comp. Music Conf., ICMC88, Cologne*, pages 357–371, 1988. 45
- [75] John Clough. Tempo: A composer’s programming language. *Perspectives of New Music*, 9(1):113–125, 1970. 272
- [76] DJ Collinge. *MOXIE: A language for computer music performance*. 1984. 22, 30, 281
- [77] Nick Collins, Alex McLean, Julian Rohrerhuber, and Adrian Ward. Live coding in laptop performance. *Organised Sound*, 8(03):321–330, 2003. 3, 289, 290
- [78] Iain Connell, Thomas Green, and Ann Blandford. Ontological sketch models: Highlighting user-system misfits. *People and Computers XVII*, pages 163–178, 2003. 312
- [79] Iain Connell, Ann Blandford, and Thomas Green. Cassm and cognitive walkthrough: Usability issues with ticket vending machines. *Behaviour & Information Technology*, 23(5):307–320, 2004. 312
- [80] Melvin E Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963. 31, 290
- [81] Curtis Cook, Margaret Burnett, and Derrick Boom. A bug’s eye view of immediate visual feedback in direct-manipulation programming systems. In *Papers presented at the seventh workshop on Empirical studies of programmers*, pages 20–41. ACM, 1997. 18
- [82] Perry R Cook and Gary Scavone. The synthesis toolkit (stk). In *Proceedings of the International Computer Music Conference*, pages 164–166, 1999. 14, 25, 285
- [83] Duncan Coutts, Isaac Potoczny-Jones, and Don Stewart. Haskell: batteries included. In *ACM Sigplan Notices*, volume 44, pages 125–126. ACM, 2008. 316
- [84] Brent Cowan and Bill Kapralos. Spatial sound for video games and virtual environments utilizing real-time gpu-based convolution. In *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, pages 166–172. ACM, 2008. 281
- [85] Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. Simula 67 common base language. 1967. 86
- [86] Roger B Dannenberg. Arctic: A functional language for real-time control. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 96–103. ACM, 1984. 286
- [87] Roger B Dannenberg. The canon score language. *Computer Music Journal*, 13(1):47–56, 1989. 286
- [88] Roger B Dannenberg. The cmu midi toolkit, version 3, 1993. 22, 30, 282
- [89] Roger B Dannenberg. The implementation of nyquist, a sound synthesis language. *Computer Music Journal*, 21(3):71–82, 1997. 23, 177, 207, 286
- [90] Roger B Dannenberg. Machine tongues xix: Nyquist, a language for composition and sound synthesis. *Computer Music Journal*, 21(3):50–60, 1997. 177, 207
- [91] Roger B Dannenberg. The nyquist composition environment: Supporting textual programming with a task-oriented user interface. *Computer Science Department*, page 512, 2008. 208
- [92] Roger B Dannenberg. Nyquist reference manual version 3.08, 2013. 208, 215

REFERENCES

- [93] Maurizio De Cecco, E Lindeman, and Miller Puckette. The ircam signal processing workstation prototyping environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 639–640. ACM, 1992. 14
- [94] Roger T Dean. *The Oxford handbook of computer music*. OUP USA, 2009. 11, 264, 266
- [95] François Déchelle and Maurizio DeCecco. The ircam real-time platform and applications. In *Proceedings of the 1995 International Computer Music Conference, International Computer Music Association, San Francisco*. Citeseer, 1995. 14
- [96] François Déchelle, Riccardo Borghesi, Maurizio De Cecco, Enzo Maggi, Butch Rován, and Norbert Schnell. jmax: An environment for real-time musical applications. *Computer Music Journal*, 23(3):50–58, 1999. 14
- [97] F. Détienne. *Software design - cognitive aspects*. Springer, 2001. 312
- [98] Gerzso Di Guigno and A Gerzso. La station de travail musical 4x. Technical report, IRCAM Technical Report, Paris, 1986. 12, 274
- [99] Stephan Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer, 2007. 314
- [100] Mark Dolson. The phase vocoder: A tutorial. *Computer Music Journal*, 10(4):14–27, 1986. 142
- [101] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based languages: From a new taxonomy to constructive proposals and their validation. In *ACM Sigplan Notices*, volume 27, pages 201–217. ACM, 1992. 86, 90
- [102] Christophe Dony, Jacques Malenfant, and Daniel Bardou. Classifying prototype-based programming languages. *Prototype-based Programming: Concepts, Languages and Applications*, 1998. 86
- [103] Paul Doornbusch. Computer sound synthesis in 1951: The music of csirac. *Computer Music Journal*, 28(1): 10–25, 2004. 273
- [104] Marc Downie. Fielda new environment for making digital art. *Computers in Entertainment (CIE)*, 6(4):54, 2008. 313
- [105] John W Eaton, David Bateman, and Søren Hauberg. *Gnu octave*. Free Software Foundation, 1997. 219
- [106] Kemal Ebcioglu and et al. X10: Programming for hierarchical parallelism and non-uniform data access (extended abstract), 2004. 302
- [107] Stephen A. Edwards, Vimal Kapadia, and Michael Halas. Compiling esterel into static discrete-event code. *Electron. Notes Theor. Comput. Sci.*, 153(4):117–131, June 2006. ISSN 1571-0661. doi: 10.1016/j.entcs.2006.02.027. URL <http://dx.doi.org/10.1016/j.entcs.2006.02.027>. 34
- [108] Martin Erwig and Bernd Meyer. Heterogeneous visual languages-integrating visual and textual programming. In *Visual Languages, Proceedings., 11th IEEE International Symposium on*, pages 318–325. IEEE, 1995. 313
- [109] Emmanuel Favreau, Michel Fingerhut, Olivier Koechlin, Patrick Potacsek, Miller Puckette, and Robert Rowe. Software developments for the 4x real-time system. In *Proceedings of the International Computer Music Conference*, pages 369–373, 1986. 275, 277
- [110] Guy Fedorkow, William Buxton, and KC Smith. A computer-controlled sound distribution system for the performance of electroacoustic music. *Computer Music Journal*, 2(3):33–42, 1978. 273
- [111] Jose L Fernandez. A taxonomy of coordination mechanisms used in real-time software based on domain analysis. Technical report, DTIC Document, 1993. 305
- [112] William Finzer and Laura Gould. Programming by rehearsal. *Byte*, 9(6):187, 1984. 313
- [113] James L Flanagan, DIS Meinhart, Roger M Golden, and Man Mohan Sondhi. Phase vocoder. *The Journal of the Acoustical Society of America*, 38:939, 1965. 142
- [114] M. Fowler. *Domain-specific languages*. Addison-Wesley, 2010. 58, 59, 285, 286
- [115] Martin Fowler. Language workbenches: The killer-app for domain specific languages. <http://martinfowler.com/articles/languageWorkbench.html>, 2005. [Online; accessed 22-Mar-2014]. 59, 286
- [116] Adrian Freed and Andy Schmeder. Features and future of open sound control version 1.1 for nime. In *NIME'09: Proceedings of the 9th Conference on New Interfaces for Musical Expression*, 2009. 239, 287
- [117] David Friend. A time-shared hybrid sound synthesizer. *Journal of the Audio Engineering Society*, 19(11):928–935, 1971. 12, 273
- [118] Hal Edwin Fulton. *The ruby way the second edition: Solutions and techniques in ruby programming*. Addison Wesley Professional, 2006. 91
- [119] D. Gabor. Lectures on communication theory. *Technical Report 238, Research Laboratory of Electronics, Massachusetts Institution of Technology*, 1952. 309
- [120] James Gabura and Gustav Ciamaga. Computer control of sound apparatus for electronic music. In *Audio Engineering Society Convention 33*, 1967. 273
- [121] Stephan J Garland. Dartmouth basic, a specification. *Hanover NH: Kiewit Computation Center. TM028*, 1973. 308
- [122] Brad Garton and Dave Topper. Rtcmix—using cmix in real time. In *Proceedings of the International Computer Music Conference. International Computer Music Association*, 1997. 12, 283, 286
- [123] Lalya Gaye, Lars Erik Holmquist, Frauke Behrendt, and Atau Tanaka. Mobile music technology: Report on an emerging community. In *Proceedings of the 2006 conference on New interfaces for musical expression*, pages 22–25. IRCAM-Centre Pompidou, 2006. 291
- [124] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985. ISSN 0164-0925. doi: 10.1145/2363.2433. URL <http://doi.acm.org/10.1145/2363.2433>. 109

REFERENCES

-
- [125] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96, 1992. 109
 - [126] Adele Goldberg and David Robson. *Smalltalk-80: The language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983. 178, 287
 - [127] Danny Goodman, Michael Morrison, and Brendan Eich. *Javascript® bible*. John Wiley & Sons, Inc., 2007. 75, 86, 91, 313
 - [128] James Gosling. *Java language specification*. Addison-Wesley Professional, 2000. 74, 86
 - [129] Thomas Green and Alan Blackwell. Cognitive dimensions of information artefacts: A tutorial. In *BCS HCI Conference*, 1998. 182, 259
 - [130] Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996. 259
 - [131] Thomas RG Green and Marian Petre. When visual programs are harder to read than textual programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics)*. GC van der Veer, MJ Tauber, S. Bagnarola and M. Antavolits. Rome, CUD. Citeseer, 1992. 191, 313
 - [132] TRG Green, M Petre, and RKE Bellamy. Comprehensibility of visual and textual programs: A test of superlativism against the 'match-mismatch' conjecture. *ESP*, 91(743):121–146, 1991. 191, 313
 - [133] Ira Greenberg. *Processing: creative coding and computational art*. Apress, 2007. 314
 - [134] Peter Grogono. Musys: Software for an electronic music studio. *Software: Practice and Experience*, 3(4):369–383, 1973. 12, 274
 - [135] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Springer-Verlag, 2010. 33, 317
 - [136] Stuart Halloway. *Programming Clojure*. Pragmatic Bookshelf, 2009. 290
 - [137] Kevin Hammond. Parallel functional programming: An introduction. *FTPable from ftp.dcs.glasgow.ac.uk*, 1994. 315
 - [138] Duane Hanselman and Bruce C Littlefield. *Mastering MATLAB 5: A comprehensive tutorial and reference*. Prentice Hall PTR, 1997. 219
 - [139] Drew Hemment. Locative arts. *Leonardo*, 39(4):348–355, 2006. 291
 - [140] Rich Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, page 1. ACM, 2008. 290
 - [141] Lejaren Hiller and Leonard Maxwell Isaacson. *Illiad suite, for string quartet*, volume 30. New Music Edition, 1957. 272
 - [142] Lejaren Hiller, Antonio Leal, and Robert A Baker. Revised musicomp manual. *U. of Illinois Experimental Music Studio Tech. Rep.*, (13), 1966. 272
 - [143] Jean-Michel Hoc. *Psychology of programming*. Academic Pr, 1990. 312, 313
 - [144] Robert R Hoffman, Beth Crandall, and Nigel Shadbolt. Use of the critical decision method to elicit expert knowledge: A case study in the methodology of cognitive task analysis. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 40(2):254–276, 1998. 312
 - [145] Manuel Hohenauer, Christoph Schumacher, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Hans van Someren. Retargetable code optimization with simd instructions. In *Proceedings of the 4th international conference on Hardware/Software Codesign and System Synthesis*, pages 148–153. ACM, 2006. 301
 - [146] Anthony T Holdener. *Ajax: the definitive guide*. " O'Reilly Media, Inc.", 2008. 314
 - [147] SR Holtzman. Using generative grammars for music composition. *Computer Music Journal*, 5(1):51–64, 1981. 272
 - [148] Hubert S Howe. Music 4bf, a fortran version of music 4b. *Princeton: Princeton University Music Dept*, 1967. 267
 - [149] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation—an algebra of music—. *Journal of Functional Programming*, 6(03):465–484, 1996. 290
 - [150] Roberto Ierusalimsky. *Programming in lua*. Roberto Ierusalimsky, 2006. xi, 31, 32, 74, 75, 82, 87, 91, 313
 - [151] Roberto Ierusalimsky. *Programming in lua, the third Edition*. Roberto Ierusalimsky, 2013. 45, 75, 78, 82
 - [152] Roberto Ierusalimsky, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua-an extensible extension language. *Software Practice and Experience*, 26(6): 635–652, 1996. 74, 75, 86, 290
 - [153] Roberto Ierusalimsky, Luiz Henrique De Figueiredo, and Waldemar Celes. The evolution of an extension language: A history of lua. In *Proceedings of V Brazilian Symposium on Programming Languages*, pages B–14–B–28. Citeseer, 2001. 290, 313
 - [154] Roberto Ierusalimsky, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1. ACM, 2007. 290
 - [155] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 119–128. ACM, 1999. 317
 - [156] Richard Jones and Rafael D Lins. Garbage collection: Algorithms for automatic dynamic memory management. 1996. 242

REFERENCES

- [157] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: The art of automatic memory management*. Chapman & Hall/CRC, 2011. 242
- [158] Simon Peyton Jones. Wearing the hair shirt: a retrospective on haskell (invited talk). In *ACM SIGPLAN Conference on Principles of Programming Languages (POPL03)*, 2003. 316
- [159] Sergi Jorda, Martin Kaltenbrunner, Günter Geiger, and Ross Bencina. The reactable*. In *Proceedings of the international computer music conference (ICMC 2005), Barcelona, Spain*, pages 579–582, 2005. 16, 189
- [160] Sergi Jordà, Günter Geiger, Marcos Alonso, and Martin Kaltenbrunner. The reactable: Exploring the synergy between live music performance and tabletop tangible interfaces. In *Proceedings of the 1st international conference on Tangible and embedded interaction*, pages 139–146. ACM, 2007. 3, 16, 189
- [161] P Kail. Forth programming language. *SOFTWARE WORLD.*, 16(3):2–5, 1985. 282
- [162] Martin Kaltenbrunner, Günter Geiger, and Sergi Jordà. Dynamic patches for live musical performance. In *Proceedings of the 2004 conference on New interfaces for musical expression*, pages 19–22, 2004. 3, 16
- [163] Martin Kaltenbrunner, Sergi Jorda, Gunter Geiger, and Marcos Alonso. The reactable*: A collaborative musical instrument. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2006. WETICE'06. 15th IEEE International Workshops on*, pages 406–411. IEEE, 2006. 16
- [164] Laszlo B Kish. End of moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3):144–149, 2002. 315
- [165] Andrew Koenig. Patterns and antipatterns. *Journal of Object-Oriented Programming*, 8(1):46–48, 1995. 306
- [166] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in action*. Manning, 2007. 91
- [167] Sasada Koichi. *Efficient implementation of Ruby virtual machine*. PhD thesis, The University of Tokyo, 2007. 240, 317
- [168] Topher La Fata. Max/msp: Writing max externals in java. 37
- [169] Paul Lansky. Cmix. *Program Documentation*. Princeton, New Jersey: Princeton University. <http://silvertone.princeton.edu/winham/man>, 1987. 271, 285, 286
- [170] Paul Lansky. The architecture and musical logic and cmix. 1990. 271, 285
- [171] Chris Lattner. Llvml and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008. 190
- [172] Chris Lattner and Vikram Adve. Llvml: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004. 189
- [173] Mikael Laurson. *PATCHWORK: A visual programming language and some musical applications*. PhD thesis, Sibelius Academy Helsinki, 1996. 290
- [174] Mikael Laurson, Mika Kuuskankare, and Vesa Norilo. An overview of pwgl, a visual programming environment for music. *Computer Music Journal*, 33(1):19–31, 2009. 290
- [175] Victor Lazzarini. The development of computer music programming systems. *Journal of New Music Research*, (ahead-of-print):1–14, 2013. 264, 283, 285
- [176] Victor Lazzarini, Alexis Kirke, Eduardo Miranda, Mika Kuuskankare, Mikael Laurson, Florian Thalmann, and Guerino Mazzola. A toolkit for music and audio activities on the xo computer. 2008. 24, 285
- [177] Paul Le Guernic, Albert Benveniste, Patricia Bournai, and Thierry Gautier. Signal-a data flow-oriented language for signal processing. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 34(2):362–374, 1986. 33
- [178] Edward A Lee. Computing needs time. *Communications of the ACM*, 52(5):70–79, 2009. 24
- [179] Insup Lee, Susan B Davidson, and Victor Fay-Wolfe. Motivating time as a first class entity. 1987. x, 40, 196, 198, 199, 206
- [180] AMC Leeming. A comparison of some discrete event simulation languages. *ACM SIGSIM Simulation Digest*, 12(1-4):9–16, 1981. 20
- [181] Rasmus Lerdorf, Kevin Tatroe, and Peter MacIntyre. *Programming PHP*. O'Reilly Media, Inc., 2009. 74
- [182] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 4.00. 2012. 59, 317
- [183] Clayton Lewis. *Using the “thinking-aloud” method in cognitive interface design*. IBM TJ Watson Research Center, 1982. 312
- [184] Henry Lieberman. A preview of act 1. *AI memo No 625*, 1981. 90
- [185] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '86, pages 214–223, New York, NY, USA, 1986. ACM. ISBN 0-89791-204-7. doi: 10.1145/28697.28718. URL <http://doi.acm.org/10.1145/28697.28718>. 90, 91
- [186] Eric Lindemann, Miller Puckette, Eric Viara, and Michel Starkier. The ircam signal processing workstation-an environment for research in real-time musical signal processing and performance. *Microprocessing and Microprogramming*, 30(1):167–174, 1990. 14
- [187] Eric Lindemann, François Dechelle, Bennett Smith, and Michel Starkier. The architecture of the ircam musical workstation. *Computer Music Journal*, 15(3): 41–49, 1991. 12, 277
- [188] Cort Lippe. Real-time computer music at ircam. *Contemporary Music Review*, 6(1):219–224, 1991. 277

REFERENCES

-
- [189] Cort Lippe. Real-time granular sampling using the ir-cam signal processing workstation. *Contemporary Music Review*, 10(2):149–155, 1994. 132
 - [190] D Gareth Loy. Notes on the implementation of musbox: A compiler for the systems concepts digital synthesizer. *Computer Music Journal*, 5(1):34–50, 1981. 281
 - [191] D Gareth Loy. Life and times of the samson box. *Computer Music Journal*, 37(3):26–48, 2013. 281
 - [192] Gareth Loy. The carl system: Premises, history, and fate. *Computer Music Journal*, 26(4):52–60, 2002. 271
 - [193] Hongmin Lu, Yuming Zhou, Jiangtao Lu, and Baowen Xu. A compile-time optimization framework for ada rendezvous. *ACM SIGPLAN Notices*, 39(2):18–25, 2004. 45
 - [194] E. A. Lyon. A sample accurate triggering system for pd and max/msp. In *Proceedings of the 2006 International Computer Music Conference*, 2006. 3, 37, 195
 - [195] Thor Magnusson. ixi lang: A supercollider parasite for live coding. In *Proceedings of the International Computer Music Conference*. University of Huddersfield, 2011. 286, 290
 - [196] Joshua B Mailman. The fluxations stochastic interactive algorithmic music engine (siame) and iphone app. In *Proceedings of the 9th Sound and Music Computing Conference, Copenhagen, Denmark*, 2012. 291
 - [197] Peter Manning. *Electronic and computer music*. Oxford University Press, 2013. 275
 - [198] Florence Maraninchi and Yann Rémond. Argos: An automaton-based synchronous language. *Computer languages*, 27(1):61–92, 2001. 33
 - [199] Shane Markstrum. Staking claims: A history of programming language design claims and evidence: A positional work in progress. In *Evaluation and Usability of Programming Languages and Tools*, page 7. ACM, 2010. 259
 - [200] Kjetil Matheussen. *Conservative garbage collectors for realtime audio processing*. Ann Arbor, MI: MPublishing, University of Michigan Library, 2009. 242
 - [201] Max V Mathews. An acoustic compiler for music and psychological stimuli. *Bell System Technical Journal*, 40:677–694, 1961. 43, 266, 267, 268
 - [202] Max V Mathews and Joan E Miller. *Music IV programmer's manual*. Bell Telephone Labs, 1979. 267
 - [203] Max V Mathews and F Richard Moore. Groove - a program to compose, store, and edit functions of time. *Communications of the ACM*, 13(12):715–721, 1970. 12, 273
 - [204] Max V Mathews, Joan E Miller, F Richard Moore, John R Pierce, and Jean-Claude Risset. *The technology of computer music*. MIT press Cambridge, 1969. 11, 42, 263, 266, 267, 308, 309
 - [205] MATLAB. *Matlab R2013b documentation*. The Math-Works Inc., Natick, Massachusetts, 2013. xvi, 223, 224
 - [206] Yukio Matsumoto and K Ishituka. Ruby programming language, 2002. 91
 - [207] Satoshi Matsuoka and Satoru Kawai. Using tuple space communication in distributed object-oriented languages. *SIGPLAN Not.*, 23(11):276–284, January 1988. ISSN 0362-1340. doi: 10.1145/62084.62108. URL <http://doi.acm.org/10.1145/62084.62108>. 110
 - [208] Andrew May. Philippe manoury: Jupiter. *Computer Music Journal*, 23(3):118–120, 1999. 275, 284
 - [209] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL <http://doi.acm.org/10.1145/367177.367199>. 316
 - [210] James McCartney. Supercollider: A new real time synthesis language. 1996. 12, 177, 286, 287
 - [211] James McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002. 44, 178, 286
 - [212] Alex McLean. *Artist-programmers and programming languages for the arts*. PhD thesis, Department of Computing, Goldsmiths, University of London, October 2011. 3
 - [213] F Richard Moore. The computer audio research laboratory at ucsd. *Computer Music Journal*, 6(1):18–29, 1982. xvii, 271
 - [214] James A Moorer, Alain Chauveau, Curtis Abbott, Peter Eastty, and James Lawson. The 4c machine. *Computer Music Journal*, 3(3):16–24, 1979. 12, 274
 - [215] Peter Naur, John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, et al. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 6(1):1–17, 1963. 75
 - [216] Click Nilson. Live coding practice. In *Proceedings of the 7th international conference on New interfaces for musical expression*, pages 112–117. ACM, 2007. 3, 289
 - [217] Vesa Norilo. Recent developments in the kronos programming language. In *Proceedings of the ICMC2013 International Computer Music Conference*, 2013. 240
 - [218] Vesa Norilo and Mikael Laurson. Kronos-a vectorizing compiler for music dsp. In *Proc. of the 12th Int. Conference on Digital Audio Effects (DAFx-09)*, 2009. 317
 - [219] Yann Orlarey, Dominique Fober, and Stéphane Letz. An algebra for block diagram languages. In *Proceedings of International Computer Music Conference*, pages 542–547. Citeseer, 2002. 290
 - [220] Yann Orlarey, Albert Gräf, and Stefan Kersten. Dsp programming with faust, q and supercollider. In *Proceedings of the 4th International Linux Audio Conference (LAC06)*, pages 39–47, 2006. 290
 - [221] Yann Orlarey, Dominique Fober, and Stéphane Letz. Faust: An efficient functional approach to dsp programming. *New Computational Paradigms for Computer Music*, 2009. 290

REFERENCES

- [222] John K Ousterhout. Scripting: Higher level programming for the 21st century. *Computer*, 31(3):23–30, 1998. 13
- [223] Andrenacci Paolo, Armani Fabio, Bessegato Renato, Paladin Andrea, Pisani Patrizio, Prestigiacomo Angelo, Rosati Claudio, Sapir Sylviane, and Vetuschì Mauro. The new mars workstation. In *Proceedings of the... International Computer Music Conference*, page 215. Computer Music Association, 1997. 279
- [224] K Peppler and Y Kafai. Creative coding: Programming for personal expression. *Retrieved August*, 30: 2008, 2005. 314
- [225] Mark Pilgrim. *HTML5: up and running*. “O’Reilly Media, Inc.”, 2010. 314
- [226] Larry Polansky. Live interactive computer music in hmsl, 1984-1992. *Computer Music Journal*, 18(2):59–77, 1994. 22, 281
- [227] M. Puckette. *The theory and technique of electronic music*. World Scientific Publishing Company, 2007. 44
- [228] Miller Puckette. The patcher. In *Proceedings of the International Computer Music Conference*, 1988. 277, 278
- [229] Miller Puckette. Something digital. *Computer Music Journal*, 15(4):65–69, 1991. 12, 274, 275
- [230] Miller Puckette. Fts: A real-time monitor for multi-processor music synthesis. *Computer music journal*, 15(3):58–67, 1991. 12, 277, 279
- [231] Miller Puckette. Combining event and signal processing in the max graphical programming environment. *Computer music journal*, 15(3):68–77, 1991. 12, 177, 277, 279, 280, 286
- [232] Miller Puckette. Pure data: Another integrated computer music environment. *Proceedings of the Second Intercollege Computer Music Concerts*, pages 37–41, 1996. 12, 44, 283, 286
- [233] Miller Puckette. Pure data. *Proceedings of the International Computer Music Conference*, 1997. 177, 283
- [234] Miller Puckette. Max at seventeen. *Computer Music Journal*, 26(4):31–43, 2002. 177, 275, 283, 307
- [235] Miller Puckette and MIT Experimental Music Studio. Music 500: A new real-time digital synthesis system. In *Proceedings of the International Computer Music Conference*, 1983. 275
- [236] Warren Rachele. *Learn Object Pascal with Delphi*. Wordware Publishing Inc., 2000. 315
- [237] Benjamin RAMSAY. Social spatialisation: Exploring links within contemporary sonic art. In *eContact! 14.4-Toronto Electroacoustic Symposium 2011 (TES 2011)*, volume 14. Canadian Electroacoustic Community, 2013. 284
- [238] JK Randall. A report from princeton. *Perspectives of New Music*, 3(2):84–92, 1965. 267
- [239] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O’Reilly Media, Inc., 2007. 301
- [240] C. Roads. Introduction to granular synthesis. *Computer Music Journal*, 12(2), 1988. 45
- [241] C. Roads. *The computer music tutorial*. The MIT Press, 1996. 41, 42, 266, 267, 270, 274, 275
- [242] C. Roads. *Microsound*. The MIT Press, 2004. ix, x, xviii, 24, 44, 47, 48, 50, 55, 57, 130, 140, 241, 310, 311
- [243] Curtis Roads and Max Mathews. Interview with max mathews. *Computer Music Journal*, 4(4):15–22, 1980. 11, 266
- [244] Curtis Roads and Paul Wieneke. Grammars as representations for music. *Computer Music Journal*, 3(1): 48–55, 1979. 272
- [245] Stewart Robinson. *Simulation: The practice of model development and use*. Wiley. com, 2004. ix, 21
- [246] X. Rodet. Time-domain formant-wave-function synthesis. *Computer Music Journal*, 8(3), 1984. 45
- [247] Julian Rohrerhuber, Alberto de Campo, and Renate Wieser. Algorithms today notes on language design for just in time programming. *context*, 1:291, 2005. 17, 178
- [248] Neil B Rolnick. A composer’s notes on the development and implementation of software for a digital synthesizer. *Computer Music Journal*, 2(2):13–22, 1978. 275
- [249] Jean-Pierre Rosen. Is ada education important? *Ada User Journal*, 29(3):208, 2008. 305
- [250] Robert Rowe. Machine listening and composing with cypher. *Computer Music Journal*, 16(1):43–63, 1992. 281
- [251] Robert Rowe. *Interactive music systems: Machine listening and composing*. MIT press, 1992. 281
- [252] James Rumbaugh. Controlling propagation of operations using attributes on relations. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA ’88, pages 285–296, New York, NY, USA, 1988. ACM. ISBN 0-89791-284-5. doi: 10.1145/62083.62109. URL <http://doi.acm.org/10.1145/62083.62109>. 91
- [253] Caitlin Sadowski and Sri Kurniawan. Heuristic evaluation of programming language features: Two parallel programming case studies. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, pages 9–14. ACM, 2011. 259
- [254] Kazuki Sakamoto and Tomohiko Furumoto. Grand central dispatch. In *Pro Multithreading and Memory Management for iOS and OS X*, pages 139–145. Springer, 2012. 301
- [255] Peter R Samson. A general-purpose digital synthesizer. *Journal of the Audio Engineering Society*, 28(3):106–113, 1980. 281

REFERENCES

- [256] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The asynchronous partitioned global address space model. In *Proceedings of The First Workshop on Advances in Message Passing*, 2010. 302
- [257] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification, 2011. 302
- [258] C. A. Scaletti and R. E. Johnson. An interactive environment for object-oriented music composition and sound synthesis. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '88, pages 222–233, New York, NY, USA, 1988. ACM. ISBN 0-89791-284-5. doi: 10.1145/62083.62103. URL <http://doi.acm.org/10.1145/62083.62103>. 12, 279
- [259] Carla Scaletti. The kyma/platypus computer music workstation. *Computer Music Journal*, 13(2):23–38, 1989. 12, 279
- [260] G Scavone and Perry Cook. Rtmidi, rtaudio, and a synthesis toolkit (stk) update. In *Proceedings of the 2005 International Computer Music Conference*, 2005. 285
- [261] Bill Schottstaedt. Pla: A composer's idea of a language. *Computer Music Journal*, 7(1):11–20, 1983. 281
- [262] Bill Schottstaedt. Machine tongues xvii: Clm: Music v meets common lisp. *Computer Music Journal*, 18(2): 30–37, 1994. 285
- [263] Carla Schroder. *The Book of Audacity*. No Starch Press, 2011. 316
- [264] B. Schwartz and M. Lentzner. Glyphic codeworks(tm) scripting. *Unpublished manual*, 1994. 86
- [265] Zack Settel and Cort Lippe. Real-time timbral transformation: Fft-based resynthesis. *Contemporary Music Review*, 10(2):171–179, 1994. 142
- [266] Ben Shneiderman and Shneiderman Ben. *Designing the user interface: Strategies for effective human-computer interaction*, 4/e (New Edition). Pearson Education India, 2003. 18
- [267] Fridtjof Siebert. *Hard realtime garbage collection: In modern object oriented programming languages*. BoD-Books on Demand, 2002. 242
- [268] Julius Smith, David Jaffe, and Lee Boynton. Sound and music on the next computer. In *Audio Engineering Society Conference: 7th International Conference: Audio in Digital Times*. Audio Engineering Society, 1989. 284
- [269] Leland Smith. Score-a musician's approach to computer music. *Journal of the Audio Engineering Society*, 20(1):7–14, 1972. 272
- [270] Randall B Smith. Prototype-based languages (panel): Object lessons from class-free programming. In *ACM SIGPLAN Notices*, volume 29, pages 102–112. ACM, 1994. 86
- [271] Ronald Bruce Smith. An interview with tristan murail. *Computer Music Journal*, 24(1):11–19, 2000. 275
- [272] Walter R Smith. The newton application architecture. In *Compton Spring'94, Digest of Papers.*, pages 156–161. IEEE, 1994. 86
- [273] Wesley Smith and Graham Wakefield. *Augmenting computer music with just-in-time compilation*. Ann Arbor, MI: MPublishing, University of Michigan Library, 2009. 190, 240, 317
- [274] Andrew Sorensen. Impromptu: An interactive programming environment for composition and performance. In *Proceedings of the Australasian Computer Music Conference 2009*, 2005. 30, 115, 177, 287, 288
- [275] Andrew C Sorensen. A distributed memory for networked livecoding performance. In *Proceedings of the ICMC2010 International Computer Music Conference*, pages 530–533, 2010. 201
- [276] A. Soresen and H. Gardner. Programming with time: Cyber-physical programming with impromptu. In *Proceedings of the 2010 SPLASH/OOPSLA*, 2010. xv, 30, 40, 106, 109, 110, 115, 177, 187, 194, 197, 200, 201, 202, 287
- [277] A. Soresen, B. Swift, and A. Riddell. Livecoding's many meanings. In *Compuer Music Journal*, in press. 18, 189, 240, 288
- [278] Michael Sperber, R Kent Dybvig, Matthew Flatt, and Anton van Straaten. Revised6 report on the algorithmic language scheme (non-normative appendices), 2007. 187, 287
- [279] EBNF Syntax Specification Standard. Ebnf: Iso/iec 14977: 1996 (e). URL <http://www.cl.cam.ac.uk/mgk25/iso-14977.pdf>, 1996. 70
- [280] John Stasko. *Software visualization: Programming as a multimedia experience*. MIT press, 1998. 314
- [281] Guy L Steele Jr. Rabbit: A compiler for scheme. 1978. 287
- [282] Daniel E Stevenson and Paul J Wagner. Developing real-world programming assignments for cs1. In *ACM SIGCSE Bulletin*, volume 38, pages 158–162. ACM, 2006. 238
- [283] Bjarne Stroustrup. *C++ programming language*, 3/e. Pearson Education India, 1994. 86
- [284] Minyoung Sung, Soyoung Kim, Sangsoo Park, Nae-hyuck Chang, and Heonshik Shin. Comparative performance evaluation of java threads for embedded applications: Linux thread vs. green thread. *Information processing letters*, 84(4):221–225, 2002. 84
- [285] Tucker S Taft and Robert A Duff. *Ada 95 reference manual. language and standard libraries: International standard ISO/IEC 8652: 1995 (E)*, volume 8652. Springer, 1997. 45, 115
- [286] Antero Taivalsaari. Kevo, a prototype-based object-oriented language based on concatenation and module operations. *Report LACIR*, pages 92–02, 1992. 86
- [287] Antero Taivalsaari. *A critical view of inheritance and reusability in object-oriented programming*. University of Jyväskylä, 1993. 86

REFERENCES

- [288] Atau Tanaka. Musical performance practice on sensor-based instruments. *Trends in Gestural Control of Music*, 13:389–405, 2000. 284
- [289] Atau Tanaka et al. Mapping out instruments, affordances, and mobiles. NIME, 2010. 291
- [290] Bruce A Tate. Seven languages in seven weeks: A pragmatic guide to learning programming languages. 2010. 74, 91
- [291] Heinrich Taube. An introduction to common music. *Computer Music Journal*, 21(1):29–34, 1997. 281
- [292] J.C. Tenney. Sound-generation by means of a digital computer. *Journal of Music Theory*, 7(1):24–70, 1963. 267
- [293] Henning Thielemann. Live music programming in haskell. *arXiv preprint arXiv:1303.5768*, 2013. 290
- [294] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby*, volume 13. Pragmatic Bookshelf, 2004. 74
- [295] Simon Thompson. *Haskell: The craft of functional programming*, volume 2. Addison-Wesley, 1999. 290, 315
- [296] Tim Thompson. Graphic editor for music. *Computing Systems*, 3(2):332–358, 1990. 281
- [297] Dave Topper. Rtcmix for linux (part 1). *Linux J.*, 2000(78es), October 2000. ISSN 1075-3583. URL <http://dl.acm.org/citation.cfm?id=364412.364417>. 12, 283
- [298] George Tzanetakis. Marsyas-0.2: A case study in implementing music information retrieval systems. *Intelligent Music Information Systems. IGI Global*, 2007. 14
- [299] George Tzanetakis and Perry Cook. Marsyas: A framework for audio analysis. *Organised sound*, 4(3):169–175, 2000. 285
- [300] David Ungar and Randall B Smith. *Self: The power of simplicity*, volume 22. ACM, 1987. 74, 86
- [301] Guido VanRossum and Fred L Drake. *The Python language reference*. Python Software Foundation, 2010. 74, 91, 239, 316
- [302] Barry Vercoe. Reference manual for the music 11 sound synthesis language. *Program documentation. Cambridge, Massachusetts: MIT Experimental Music Studio*, 1981. 267
- [303] Barry Vercoe. *Csound: A manual for the audio processing system and supporting programs with tutorials*. Massachusetts Institute of Technology, 1993. 267
- [304] Barry Vercoe. Extended csound. In *Proceedings of the International Computer Music Conference*, pages 141–142. INTERNATIONAL COMPUTER MUSIC ASSOCIATION, 1996. 281
- [305] Barry Vercoe and Dan Ellis. Real-time csound: Software synthesis with sensing and control. In *Proceedings of the International Computer Music Conference*, pages 209–211, 1990. 11, 12, 283
- [306] G Wakefield and W Smith. Using lua for multimedia composition. In *Proceedings of the International Computer Music Conference. San Francisco: International Computer Music Association*, pages 1–4, 2007. 31
- [307] Graham Wakefield, Wesley Smith, and Charles Roberts. Luaav: Extensibility and heterogeneity for audiovisual computing. In *Proceedings of the Linux Audio Conference*, 2010. xi, 31, 32, 105, 290
- [308] Graham David Wakefield. Vessel: Interleaving sample-accurate synthesis and control, 2007. 31, 290
- [309] David C Walden, Tom Van Vleck, and FJ Corbató. *The compatible time sharing system (1961-1973): Fiftieth anniversary commemorative overview*. IEEE Computer Society, 2011. 266
- [310] Larry Wall et al. The perl programming language, 1994. 74
- [311] Ge. Wang. *The Cambridge companion to electronic music.*, chapter A history of programming and music, pages 103–134. Cambridge University Press, 2007. 264
- [312] Ge. Wang. *The chuck audio programming language. a strongly-timed and on-the-fly environ/mentality*. PhD thesis, Princeton University, 2008. xi, 3, 14, 33, 35, 36, 44, 50, 74, 82, 105, 177, 264, 271, 286, 288
- [313] Ge Wang. Designing smule's iphone ocarina. In *Proceedings of the International Conference on New Interfaces for Musical Expression. Pittsburgh*, 2009. 291
- [314] Ge Wang and Perry Cook. Chuck: A programming language for on-the-fly, real-time audio synthesis and multimedia. In *Proceedings of the 12th annual ACM international conference on Multimedia*, pages 812–815. ACM, 2004. 44, 74, 177, 288
- [315] Ge Wang and Perry R Cook. The audicle: A context-sensitive, on-the-fly audio programming environ/mentality. In *Proceedings of the International Computer Music Conference*, pages 256–263, 2004. 288
- [316] Ge Wang, Rebecca Fiebrink, and Perry R Cook. Combining analysis and synthesis in the chuck programming language. In *Proceedings of the International Computer Music Conference*, pages 35–42, 2007. 219, 223
- [317] Ge Wang, Georg Essl, Jeff Smith, Spencer Salazar, P Cook, Rob Hamilton, Rebecca Fiebrink, Jonathan Berger, David Zhu, Mattias Ljungstrom, et al. Smule=sonic media: An intersection of the mobile, musical, and social. In *Proceedings of the International Computer Music Conference (ICMC 2009)*, pages 16–21, 2009. 291
- [318] Adrian Ward, Julian Rohrerhuber, Fredrik Olofsson, Alex McLean, Dave Griffiths, Nick Collins, and Amy Alexander. Live algorithm programming and a temporary organisation for its promotion. In *Proceedings of the README Software Art Conference*, 2004. 289, 290
- [319] Philippe Waroquiers, Stef Van Vlierberghe, Dirk Craeynest, Andrew Hatley, and Erik Duvinage. Migrating large applications from ada83 to ada95. In *Reliable Software Technologies? Ada-Europe 2001*, pages 380–391. Springer, 2001. 65, 305
- [320] S. Wilson, D. Cottle, and N. Collins. *The SuperCollider book*. The MIT Press, 2011. xi, xv, 4, 17, 44, 47, 177, 178, 180, 183, 184, 185, 258, 286, 307

REFERENCES

- [321] Niklaus Wirth. The programming language pascal. *Acta informatica*, 1(1):35–63, 1971. 75
- [322] T. Wishart. *Audible design*. Orpheus Books, 1994. 24, 45, 47, 54
- [323] T. Wishart. *Audible design, appendix 2*. Orpheus Books, 1994. 47, 134
- [324] Matthew Wright. Open sound control-a new protocol for communicationg with sound synthesizers. In *Proceedings of the 1997 International Computer Music Conference*, pages 101–104, 1997. 201, 287
- [325] Iannis Xenakis. *Formalized music: Thought and mathematics in composition*. Pendragon Pr, 1992. 24, 272
- [326] T. Yuasa. Real-time garbage collection on general-purpose machines. *J. Syst. Softw.*, 11(3):181–198, March 1990. ISSN 0164-1212. doi: 10.1016/0164-1212(90)90084-Y. URL [http://dx.doi.org/10.1016/0164-1212\(90\)90084-Y](http://dx.doi.org/10.1016/0164-1212(90)90084-Y). 242, 297
- [327] D. Zicarelli. An extensible real-time signal processing environment for max. In *Proceedings of the 1998 International Computer Music Conference*, 1998. 12, 44, 177, 283
- [328] John Zimmerman, Jodi Forlizzi, and Shelley Evenson. Research through design as a method for interaction design research in hci. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 493–502. ACM, 2007. 259
- [329] Iohannes Zmölnig and Gerhard Eckel. Live coding: An overview. In *Proceedings of the International Computer Music Conference*, 2007. 289

REFERENCES

6

Appendix I: Related Publications

This section lists the papers related to this thesis published during the author’s Ph.D study with the brief descriptions. The author is the first author of all the publications listed. While these papers describe both the conceptual background and the early design of LC, some of the recent improvements in its language design are still not published in the forms of conference papers nor journal papers; the author is currently preparing several journal papers for future submission.

Generally speaking, the first five papers in the following list describe the approach taken during the identification of the problem of the difficulty in microsound synthesis programming in unit-generator languages. While the HCI aspect described in these four papers is not the central issue of this thesis, such an approach played a significant role during the design process of LC, even though the thesis mainly focuses on the description of the resulting design and its contributions rather than the design process.

The rest of the four papers describe the resulting language design and its benefits. Yet, the design of LCSynth sound synthesis language, which is an early prototype of LC, was already described in “How Can a DSL for Expert End-users be Designed for Better Usability?: A Case Study in Computer Music” (the 4th paper in the below list).

Cognitive Issues in Computer Music Programming

This paper was presented as a short paper (with the poster presentation) at *the International Conference on New Interfaces for Musical Expression* (NIME) 2011, Oslo, Norway. The paper contextualizes the issues related in computer music programming

6. APPENDIX I: RELATED PUBLICATIONS

in the psychology of programming to prepare further discussion in the problems in computer music programming.

Misfits in Abstractions: Towards User-centred Design in Domain-specific Languages for End-user Programming

This paper was presented as a poster paper (with oral presentation) at *the ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity* (ACM SPLASH) 2011, Portland, Oregon, USA and also awarded the 3rd place in *the ACM SPLASH Student Research Competition* 2011.

This paper describes an approach for the identification and assessment of usability problems in domain-specific programming language design. While the paper takes the problem in expressing single-sample feedback in SuperCollider computer music language (320) as an example, it is the first paper that discusses the approach to assess problems in computer music language design by identifying gaps between the conceptualization of musical time-scales and the abstractions applied to the sound synthesis framework.

On Conceptual Misfits in Computer Music Programming

This paper was presented as a regular paper (with the oral presentation) at *the Asian Computer Music Conference* (ACMC), Tokyo, Japan, 2011. This paper is an extended version of “Misfits in Abstractions: Towards User-centred Design in Domain-specific Languages for End-user Programming”. Yet, the extended discussion in this paper suggested the problem of the difficulty in microsound synthesis programming in unit-generator languages for the further research, for the first time in this Ph.D study.

How Can a DSL for Expert End-users be Designed for Better Usability?: A Case Study in Computer Music

This paper was presented as a work-in-progress paper (with the poster presentation) at *the ACM CHI Conference on Human Factors in Computing Systems* (ACM CHI), Austin, Texas, 2011. This paper further discusses the issues in the difficulty involved in computer music programming from the perspective based on the previous papers. The paper also discusses the utilization of a HCI framework for the assessment of the problems in computer music programming language design and the evaluation of the

design of a new computer music language, referring to the previous works, such as the *Cognitive Dimension of Notations* framework (41)(129)(130), Sadowski's heuristics evaluation framework for additional features to the programming language (253), Markstrum's survey for the evaluation of the usability claims in programming language research (199), and Blandford's CASSM (Concept-based Analysis of Surface and Structural Misfits) framework (44).

The paper describes both problems of single-sample feedback and microsound synthesis in unit-generator computer music languages, taking the examples in SuperCollider and also provides the design proposal for a new computer music language. The assumption is also discussed that the difficulty in computer music programming can be reduced by appropriately designing a sound synthesis framework, which can fill the gaps between the conceptualization of musical time-scales and the abstractions in the sound synthesis framework. The design proposal was later implemented as LCSynth, a sound synthesis language.

Developing a New Computer Music Programming Language in the 'Research through Design' Context

This paper was presented as a short paper (with oral presentation) in the section of *the ACM SPLASH Doctoral Symposium of the ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity* (ACM SPLASH/OOPLSA), Tucson, Arizona, USA, 2012. The paper describes the approach taken in the design process of LC, which begins with the identification and assessment of the problems in the existing computer music languages and then uses the analysis for the design of a new computer music language, referring to the concept of *the research-through design* (328). While this paper focuses on such an approach rather than the resulting language design and its benefits, it describes the concept of mostly-strongly-timed programming for the first time.

Mostly-strongly-timed programming

This paper was presented as a poster paper in a section of *the ACM Student Research Competition at the ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity* (ACM SPLASH/OOPLSA), Tucson, Arizona,

6. APPENDIX I: RELATED PUBLICATIONS

USA, 2012. The paper briefly describes the concept of mostly-strongly-timed programming, together with the motivation behind the concept and the design proposal for the language feature to switch between the synchronous/non-preemptive context and asynchronous/preemptive context.

LCSynth: A Strongly-timed Synthesis Language that Integrates Objects and Manipulations for Microsounds

This paper was presented as a long paper (with oral presentation) at *the Sound and Music Computing Conference* (SMC), Copenhagen, Denmark, 2012 and was also nominated for best paper. This paper was co-authored by Prof. Naotoshi Osaka.

It describes the design of LCSynth, a sound synthesis language, which directly integrates objects and manipulations for microsound synthesis in its language design. The language design became the basis of the sound synthesis framework in LC.

Unit-Generator Considered Harmful (For Microsound Synthesis): A Novel Programming Model for Microsound Synthesis in LCSynth

This paper was presented as a long paper with the oral presentation at *the International Computer Music Conference* (ICMC), Perth, Australia, 2013 and was co-authored by Prof. Ryohei Nakatsu and Prof. Naotoshi Osaka.

The paper describes a programming model for microsound synthesis in LCSynth and the detailed examples of several different microsound synthesis techniques, together with more discussion on why the traditional unit-generator concept may not be very appropriate for microsound synthesis and the benefits of LCSynth's sound synthesis framework design.

LC: A Strongly-timed Prototype-based Programming Language for Computer Music

This paper was presented as a long paper with the oral presentation at *the International Computer Music Conference* (ICMC), Perth, Australia, 2013 and was co-authored by Prof. Ryohei Nakatsu and Prof. Naotoshi Osaka.

The paper describes an earlier prototype of LC, which was still a hosting language for LCSynth. Yet, the idea to adopt prototype-based programming to a computer music

language is discussed, so to provide better support in the dynamic modification of a computer music program for rapid-prototyping and live-coding.

6. APPENDIX I: RELATED PUBLICATIONS

Appendix II: A Brief History of Computer Music Languages and Systems - the Synergy between Technology and Creativity

The idea that computers can open up new territories of musical creation attracted researchers and composers, even when only the experts of the time were granted access to huge mainframe computers installed in research institutions; as we know today, the idea is still valid and under exploration by many researchers and artists.

Shortly after the first digital sound synthesis program was developed by Mathews and his colleagues at the Bell laboratory (204) in the late 1950s, the researchers began designing special-purpose languages tailored for computer music with domain-specific abstractions. Since then, computer music languages have been continuously evolved by researchers and engineers throughout the history, as primary tools for computer music research and creation. A number of new computer music languages have been designed and developed, being supported and influenced by the development of computer technology and the achievement in programming language research of the time.

This chapter briefly describes previous research in computer music languages, dividing it into two eras of early computer music programming languages and modern computer music languages. This division between ‘early’ and ‘modern’ is drawn by the

7. APPENDIX II: A BRIEF HISTORY OF COMPUTER MUSIC LANGUAGES AND SYSTEMS - THE SYNERGY BETWEEN TECHNOLOGY AND CREATIVITY

capability of real-time sound synthesis and interaction, as seen in the previous surveys described from the same perspective (175)(311)(312). While the transition between two eras is gradual and hard to separate clearly, the capability of real-time digital signal processing and interaction made a significant change in the design of computer music languages and systems.

Since the development of computer music programming languages has been significantly influenced by both the advance of computer technology and the related academic research, we describe computer music programming languages together with the related hardware and software in this section. However, it should be also emphasized that the evolution of computer music languages and systems were made not just by the advance of technology, but also through creative practices of the time; therefore, the emphasis on the synergy between technology and creativity and how it drove the evolution of computer music languages is also of significant interest in this chapter.

7.1 Early computer music programming languages and systems

7.1.1 MUSIC-N languages

Mathews and his colleagues began their experiments in the domain of digital sound synthesis with an IBM 704 mainframe computer in the 1960s. A series of computer music languages they developed are known for the establishment of two core domain-specific abstractions for computer music, which are *the unit-generator concept* and *the score-orchestra model*.

While an IBM 704 was huge enough to occupy the whole room (Figure 7.1), its capability was significantly limited (only 0.006-0.04 MIPS¹ with just 4,096 words (of 36 bits) in its magnetic core memory (94, pp.47-50)), compared to the computers we have today. Even though, it was one of the fastest mainframe computers available around that time.

¹MIPS stands for million instructions per second. So an IBM 704 could process only 6000-40,000 instructions per second.

7.1 Early computer music programming languages and systems



Figure 7.1: An IBM 704 computer - installed at NASA in 1957. (This image is public domain. - NASA copyright policy states that NASA material is not protected by copyright unless noted.)

7. APPENDIX II: A BRIEF HISTORY OF COMPUTER MUSIC LANGUAGES AND SYSTEMS - THE SYNERGY BETWEEN TECHNOLOGY AND CREATIVITY

Mathews and colleagues developed the first digital sound synthesis program in history in 1957, which is now known as MUSIC-I (201)(243), on an IBM 704 at the Bell laboratory. Yet, as it can be inferred from the specification of the IBM 704, the capability of MUSIC-I program was significantly limited; it could only play the monophonic triangle wave sound. However, it should be noted that MUSIC-I could take an input of a *score* file (94, p.47), even though only “a patient user could specify notes only in terms of pitch, waveform, and duration” (241, p.87). Music-II, developed in 1958, was little more advanced and introduced the concept of *wave tables*, but still could generate only four independent voices with a choice of 16 different waveforms stored in memory (243).

It took lots of effort to synthesize digital sound in those days. Mathews had to go to IBM World Headquarters in New York for the synthesis calculation on an IBM 740 computer and then bring a digital magnetic tape back to Bell Telephone Laboratories, New Jersey, where the tape was converted to sound by their 12 bit vacuum tube converter (241, p.87)(243).

Yet, after Mathews and his colleagues obtained access to an IBM 7094, *one of the biggest, fastest computers available* around that time¹, they developed MUSIC-III in 1960 (94, p.26)(243). MUSIC-III significantly improved the flexibility in digital sound synthesis compared to its predecessors; it is known as the first computer music language that introduced the unit-generator concept (201), which performs “conceptually similar functions to standard electronic equipment used for electronic sound synthesis” (204, p.15). This concept of unit-generators is considered as “one of the most significant developments in the design of digital synthesis languages” (241, p.89) even today.

While we discussed the unit-generator concept in more detail in Chapter 2, the pictorial representation of the instrument definition of the unit-generator concept in the early era was already almost the same as we have today, while the unit-generators of our time are extended by the programming paradigm such as object-oriented programming. Figure 7.2 shows a pictorial representation of an instrument built with the unit-generators and Figure 7.3 is the definition of the instrument in MUSIC-III, as seen

¹An IBM7094 was “able to add floating numbers at a speed of about 0.35MIPS” with 32 kilowords of 36-bit-word memory for standard 7094 (309, p.5).

7.1 Early computer music programming languages and systems

in their publication (201).

Thus, two core abstractions for computer music languages, the unit-generator concept and the score-orchestra model, were established in the very early stage of computer music history. This is remarkable when considering that these computer music languages were developed almost right after the commercial release of FORTRAN, the first high-level programming language, in 1957 (26).

Even after the establishment of these core abstractions, MUSIC-N series languages were continuously updated by Mathews and other researchers. While MUSIC-IV developed by Mathews and Miller was a re-coding of MUSIC-III in a new macro assembly language (202)(292), several variants of MUSIC-IV were developed also by the other researchers; Winham and Howe developed MUSIC-IVB in 1965 for IBM 7094 with many additions to the original MUSIC-IV (238) and MUSIC-4BF, in Fortran II and BAL assembler (Basic Assembly Language) at Princeton University (148). Chowning and his colleagues developed MUS10 for PDP-10 in PDP-10 assembler and Vercoe developed MUSIC 360 for IBM 360 in BAL assembler (241, p.789).

The development of a high-level programming language led to machine-independent implementations, such as MUSIC-V fully written in Fortran in 1966 (204). Together with the popularization of fast minicomputers and Unix workstations, the researchers and engineers began porting the MUSIC-N languages and the variants to many different environments. These variants were ported to the minicomputers and the workstations of their time, which became fast enough to execute computer music languages; for instance, MUSIC-4C was developed for DEC VAX-11 (Figure 7.4) as a variant of MUSIC-IV by Beauchamp in 1985 (34)(241, p.789) in C programming language and then ported to other Unix computers (35).

Music-11 developed by Varcoe and his colleagues in 1973 (302) was especially remarkable among the MUSIC-N descendants. Although it was first implemented in Macro-11 assembly languages for a PDP-11 minicomputer, Music-11 was then re-coded in C programming language as *Csound* for VAX-11 minicomputers in 1986 (303), and *Csound* itself is one of the most widely-used computer music language even today (52).

The improvement in computer technology has already released the researchers and composers in this era from the tiring procedures of the last decades; with faster com-

7. APPENDIX II: A BRIEF HISTORY OF COMPUTER MUSIC LANGUAGES AND SYSTEMS - THE SYNERGY BETWEEN TECHNOLOGY AND CREATIVITY

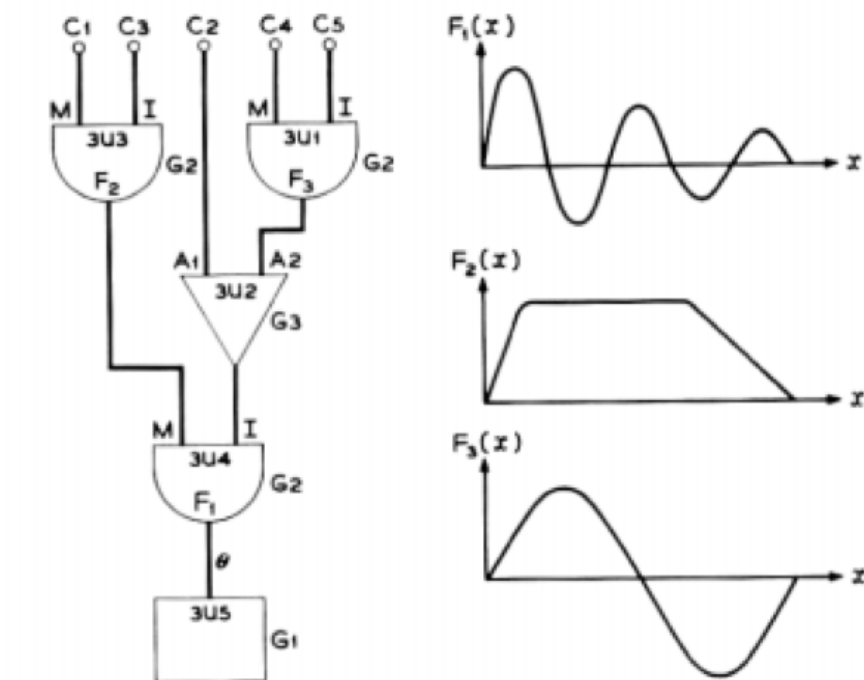


Figure 7.2: An instrument with attack, decay, and vibrato - as seen in *An Acoustic Compiler for Music and Psychological Stimuli* by Mathews (201) (reproduced from Figure 2.16).

Card Columns				
8	9	10	...	16
M	A	C		G2, 3U1, F3, (3U2,A2), x, x
M	A	C		G3, 3U2, (3U4,I), x, x
M	A	C		G2, 3U3, F2, (3U4,M), x, x
M	A	C		G2, 3U4, F1, (3U5, θ), x, x
				G1, 3U5
M	A	C		S1, I3, ((3U3,M,C1)(3U3,I,C3) \$
E	T	C		(3U2,A1,C2)(3U1,M,C4)(3U1,I,C5) \$
E	T	C		(3U3,S,PO))

Figure 7.3: An instrument with attack, decay, and vibrato - , which is pictorially described in Figure 7.2 (201). The definition is given to the mainframe computer as punched cards.

7.1 Early computer music programming languages and systems



Figure 7.4: VAX 11/750 - as exhibited in Vienna Technical Museum. (Photo by Dave Fischer. This file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license.)

7. APPENDIX II: A BRIEF HISTORY OF COMPUTER MUSIC LANGUAGES AND SYSTEMS - THE SYNERGY BETWEEN TECHNOLOGY AND CREATIVITY

puters and better human-computer interfaces, such as CRT Terminals (Figure 7.5), the process of digital sound synthesis and computer music composition was significantly facilitated, even though the computational speed was still far from real-time sound synthesis.



Figure 7.5: DEC VT-100 Terminal - (Photo by ClickRick. This file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license.)

The further development of technology made it possible to port computer music languages to personal computers around the end of 1980s. MUSIC-4C was ported to Apple Macintosh around 1988 by Gerrard and Csound was also ported to many different platforms such as IBM-PC, Apple Macintosh, and Amiga (241, p.790); thus, computer music programming languages rapidly spread outside the research institution and onto the personal desktops in this era.

7.1 Early computer music programming languages and systems

```
% cscore score.c | cmusic | filter lowpass | sndout
```

Figure 7.6: A command line example in CARL (213).

7.1.2 Other notable early computer music programming languages and systems

7.1.2.1 Other Music-N descendant and non Music-N descendant languages

Besides MUSIC-N languages, there also existed other noteworthy computer music languages and systems in the same era.

For instance, the CARL system by Moore and Loy (192) is another remarkable work in the 1980s. It was “a collection of small, command line programs that could send data to each other” (312) on a UNIX system. These command line programs can “intercommunicate via Unix pipes, which allow the output of one program to be fed into the input of another without intermediate file storage” and “several programs that process digital signals may be piped together, which is tantamount to connecting the programs together in a cascade fashion” (213). Figure 7.6 describes a simple example in CARL by Moore (213). In this example, the ‘cscore’ program generates an input score from the file ‘score.c’. Then the score is fed into cmusic, which is a sound synthesis program. The output sound data is then applied to the low pass filter by ‘filter’ program, cascaded to the sound output.

While it may not be a direct descendant of Music-N languages, Cmix developed by Lansky in 1984 (169) is also a notable computer music software with a significant influence from Music-N languages. Lansky describes Cmix as *essentially a toolkit for synthesis and analysis* and “differs substantially from most synthesis packages in that it has no scheduler and accumulates mainly by mixing to disk” (170). However, Cmix also provides MINC, a tiny programming language, which is sort of the subset of C programming language, to facilitate the compositional process. Figure 7.7 shows an example code of MINC.

7.1.2.2 Computer music programming languages and systems for algorithmic compositions

The use of computers for algorithmic compositions has been of significant interest among researchers and composers since the early days of the history of computer music;

7. APPENDIX II: A BRIEF HISTORY OF COMPUTER MUSIC LANGUAGES AND SYSTEMS - THE SYNERGY BETWEEN TECHNOLOGY AND CREATIVITY

```
01: /* START:
02: p0=start; p1=dur; p2=pitch(oct.pc); p3=fundamental decay time
03: p4=nyquist decay time; p5 = amp; p6=squish; p7=stereo spread [optional]
04: p8=flag for deleting pluck arrays (used by FRET, BEND, etc.)[optional]
05: */
06:
07: rtsetparams(44100,2)
08: load("STRUM")
09: makegen(2, 2, 7, 7.00, 7.02, 7.05, 7.07, 7.10, 8.00, 8.07)
10:
11: srand(0)
12: for (st = 0; st < 15; st = st + 0.1) {
13:   pind = random() * 7
14:   pitch = samplefunc(2, pind)
15:   START(st, 1.0, pitch, 1.0, 0.1, 10000.0, 1, random())
16: }
```

Figure 7.7: A MINC program example (taken from STRUM1.sco, which is a part of the RTcmix 4.0 package released under GPL license).

for instance, Hiller and Isaacson composed *Illiad Suite* in 1956 (141), which is known as the first algorithmic composition to involve a computer (ILLIAC I at the University of Illinois at Urbana-Champaign). Such an interest also led to the development of computer music programming languages for algorithmic composition.

Hiller and Baker developed MUSICOMP (142) for the ILLIAC computer between the late 1950s and early 1960s. While it was a library of subroutines for automated composition instead of a computer music language, it is considered to be the earliest known example of the kind (10). Xenakis, a Greek composer, is also known for the development of a series of computer programs for his stochastic compositions (325).

Researchers and composers developed a number of algorithmic composition languages and systems in the following years. The well-known languages include TEMPO by Clough (75), SCORE by Smith (269), PLAY by Cadabre and Meyers (73), Tree and Cotree by Roads (244), and GGD by Holtzman (147).

While these languages were mostly non real-time off-line compositional programming languages, the emergence of MIDI-based interfaces in the latter decade made it possible to perform algorithmic compositions in real time. The techniques developed for algorithmic compositions provided the basis for the establishment of interactive

music in the later decades.

7.2 Real-time computer music programming languages and systems

7.2.1 Early live computer music systems (before real-time digital sound synthesis)

Although computers were still far from performing real-time digital sound synthesis in the early era, researchers also sought for alternative means for live computer music. In 1970s, there emerged hybrid computer music systems composed of a computer and analogue sound synthesis hardware.

The earliest known experiment of a live computer music presentation was performed even before the first digital sound synthesis program (MUSIC-I) was developed by Mathews. The CSIR Mk1 computer, which was renamed to CSIRAC later, is known to be the first computer that played music around 1951. The CSIR Mk1 had a build-in loudspeaker, which was used as an output device for warning purposes, as it lacked any device for visual feedback. Geoff Hill programmed CSIR Mk1 to send pulses to this loud speaker so that it can produce an audible result to be heard as musical melodies. However, this experiment by Hill didn't influence the early computer music research much and is considered just a one-time experiment¹ (103).

The GROOVE system by Mathews and Moore is more frequently mentioned as an example of an early live computer music system in this era. They describe the GROOVE system as “a program to compose, store, and edit functions of time”, which controls analogue devices such as electronic synthesizers by a small computer (DDP-224). The system was equipped with knobs and joysticks as well as a computer screen and a keyboard, and high-level musical control could be performed (203). This concept to combine a microcomputer with analogue synthesizers was frequently applied to live computer music systems in the same era. The list of the known examples of this kind includes the Hybrid IV system (110), PIPER (120), the Yale synthesizer (117), and

¹Doornbusch reported that there was likely a similar experiment on another early computer, the Ferrati Mark I, and the tape recording is archived as No.H3942 in the British National Sound Archive. Yet, He states that the detailed documentation for this experiment could not be found (103).

7. APPENDIX II: A BRIEF HISTORY OF COMPUTER MUSIC LANGUAGES AND SYSTEMS - THE SYNERGY BETWEEN TECHNOLOGY AND CREATIVITY

MUSYS (134).

The use of peripheral sound devices greatly reduced the necessity for the computational power for live computer music, as Buxton's noted that "whereas digital synthesis requires a minimum of 32,000 samples per second, hybrid systems only need approximately 100 for each device being controlled" (68) in 1977. This approach was followed by the emergence of personal computers and MIDI (Musical Instrument Digital Interface) (23), replacing analogue synthesizers with digital synthesizers in the early 1980s; yet, while such hybrid systems composed of a small computer and separate synthesizers made it possible to control musical events in real time, it significantly limited the exploration in the new sound materials compared to unit-generator languages. The further advance in computational speed was still demanded for real-time digital sound synthesis.

7.2.2 The emergence of variable-function digital signal processors

While digital synthesizers began to commercialize in the late 1970s and the following emergence of MIDI made real-time digital sound synthesis possible, these commercial digital synthesizers were built on application-specific, *fixed-function* DSPs and are still similar to the analogue synthesizers of the 1960s in functionality (241, p.938). However, the emergence of general-purpose, programmable *variable-function* DSPs finally realized the application of the unit-generator concept for real-time DSPs.

Di Giugno and his colleagues at IRCAM developed a series of variable-function DSP hardware that work as highly-programmable digital sound synthesizers between the late 1970s and the early 1990s. The list of such DSP platforms¹ includes 4A in 1976 (229), 4B in 1977 (8), 4C in 1978 (214), and 4X in 1981 (98).

While the IRCAM 4A was still not a quite general-purpose DSP platform and just consisted of 256 digital oscillators and matching envelope generators, which can perform additive synthesis under the control of a PDP-11 mini computer, its successors,

¹In (229), Puckette describes at least one of the prototypes for 5A was also developed by Di Giugno and the prototype introduced floating point arithmetic and jump instruction, however no further information was provided in the publication.

7.2 Real-time computer music programming languages and systems

4B and 4C, extended the functionality and provided FM synthesis facilities(197, p.233).

As seen in the early hybrid systems for real-time computer music, in which a computer controls analogue synthesizer hardware, the control programs for these IRCAM digital synthesizers were also developed. Yet, unlike the early hybrid systems with analogue synthesizers, the software packages for these digital synthesizers often included the programming languages to describe digital sound synthesis algorithms.

For instance, Rolnick and Prevot developed SYN4B, a programming language for 4B in 1978. The SYN4B language could specify the oscillator connections to be processed inside 4B and was capable to let a 4B play the notes with a note list (score) or with the real-time input devices (248). 4CED is another notable example of this kind of languages. 4CED was developed by Abbot for a PDP-11/34 computer to work with an IRCAM 4C digital synthesizer. Its software package included a unit-generator language for 4C and a score language, together with a command language that can directly control the 4C synthesizer (4). Furthermore, 4CED extended the capability of interactive computer music performance, allowing each 4CED score to act as “an independent process that could accept input data and trigger events that caused other scores to start playing” (241, p.806).

After the development of 4C, IRCAM developed 4X (Figure 7.8). Unlike 4A, 4B and 4C, 4X was something more than ‘pure synthesizer’ and was capable of performing ‘signal processing’ of live instruments (229). 4X is well-known for the applications, both to musical creation and academic research; 4X was used to realize several important contemporary music compositions of the decade such as Répons by Pierre Boulez (54), Désintégration by Tristan Murail (271), and Jupiter by Philippe Manoury (208), and a series of programs written for the 4X precursor to the computer music research of the latter decade. While there were other noteworthy programming languages for 4X such as 4xy (a compiler of control programs by Rowe and Koechlin) and the 4X patch language (a visual programming language for real-time sound control by Potacsek) (109), MAX by Puckette should be especially noted for its popularization and influence that it gave to the design of visual computer music languages in the latter decade.

Puckette describes the first instance of what might be called ‘MAX’ was *the m orchestra language* for MUSIC 500 system (235) that he developed at MIT (234). The

7. APPENDIX II: A BRIEF HISTORY OF COMPUTER MUSIC LANGUAGES AND SYSTEMS - THE SYNERGY BETWEEN TECHNOLOGY AND CREATIVITY

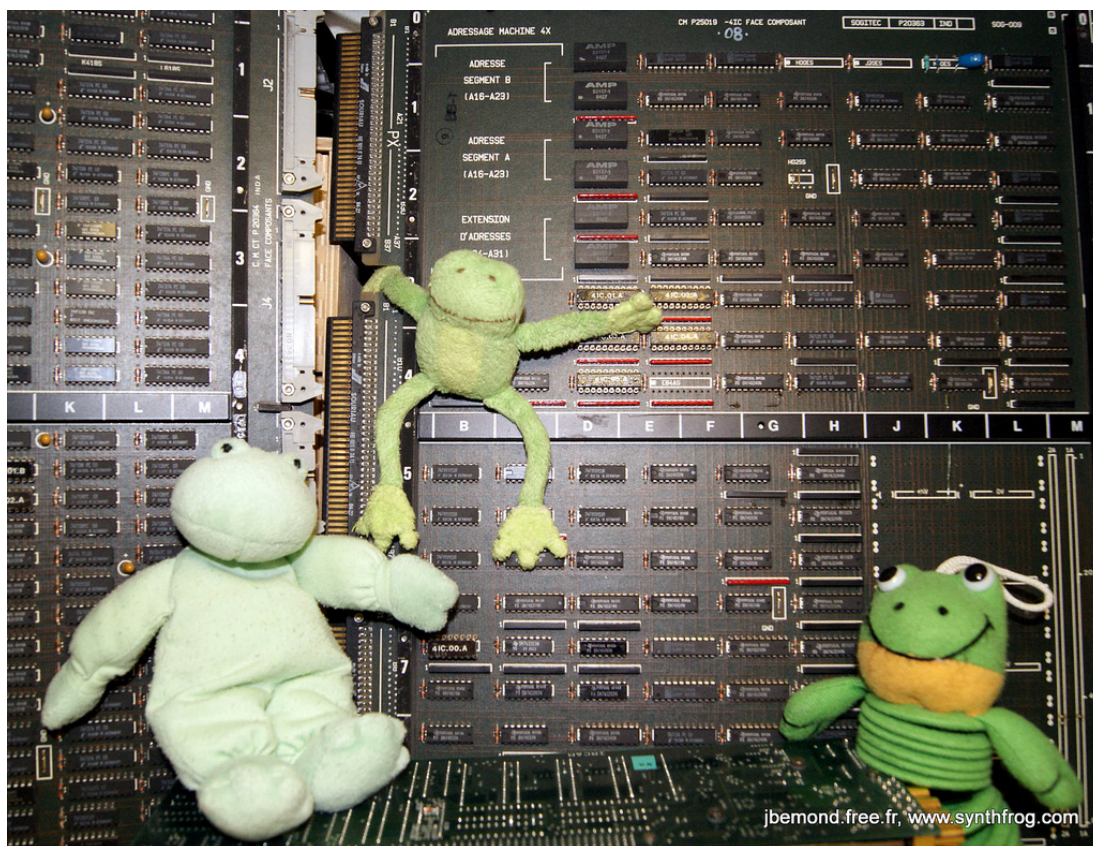


Figure 7.8: An IRCAM/Sogitec Real-Time Digital Signal Processor 4X's circuit boards - A photo by Jean-Bernard Emond. This image is licensed under the Creative Commons Attribution 2.0 Generic license.

7.2 Real-time computer music programming languages and systems

early version of MAX for 4X was written in 4xy language and is described as ‘a real-time control system’ that configures control processes (or objects) and manages message-based communications between the control processes (109). Puckette describes the ideas for MAX “aimed at making it possible to design elements of a system which can be combined quickly and without changing code” (109); in other words, the development of MAX reflects the necessity for rapid-prototyping of computer music systems in this era.

While this earliest version of MAX was configured by text files, soon after its emergence, Puckette developed *the Patcher* (228), a visual programming language for 4X. The visual programming environment as described in Puckette’s publication in 1988 was already quite similar to what we know as MAX today. Figure 7.9 shows the example of a Patcher program by Puckette shown in (228).

The Patcher program and the other early versions of MAX were visual programming languages that only interact and control the other devices such as MIDI instruments or 4X synthesizers via MIDI interfaces (188). As seen in the previous IRCAM systems, the digital signal processing system was separated from the language itself in these early versions.

Following the success of 4X, IRCAM developed IRCAM Music Workstation (IMW) (187) in the late 1980s¹. An IMW system consists of “one or more NeXT host computers together with between 2 and 24 i860 coprocessors (CPs) running at 40 MHz, nominally capable of 80 million floating-point operations per second (MFLOPS) apiece. The CoProcessor Operating System (CPOS), has been written specifically to fill the requirements this hardware poses for real time musical synthesis and control” (230). Figure 7.10 shows the pictures of an IRCAM Music Workstation and an I860 board.

The version of MAX developed for IMW integrates the DSP functionality and the users could write control programs that run on the computers seamlessly with the description of sound synthesis modules to be processed by I860 boards in one visual programming environment (231). While a computer program called FTS (“faster than sound”) processes DSP separately on CPOS (“CoProcessor Operating System”), an

¹IRCAM Music Workstation is also often referred as IRCAM Signal Processing Workstation (ISPW), yet in this thesis, we use ‘IRCAM Music Workstation (IMW)’ as the name appears in the related publications by IRCAM.

7. APPENDIX II: A BRIEF HISTORY OF COMPUTER MUSIC LANGUAGES AND SYSTEMS - THE SYNERGY BETWEEN TECHNOLOGY AND CREATIVITY

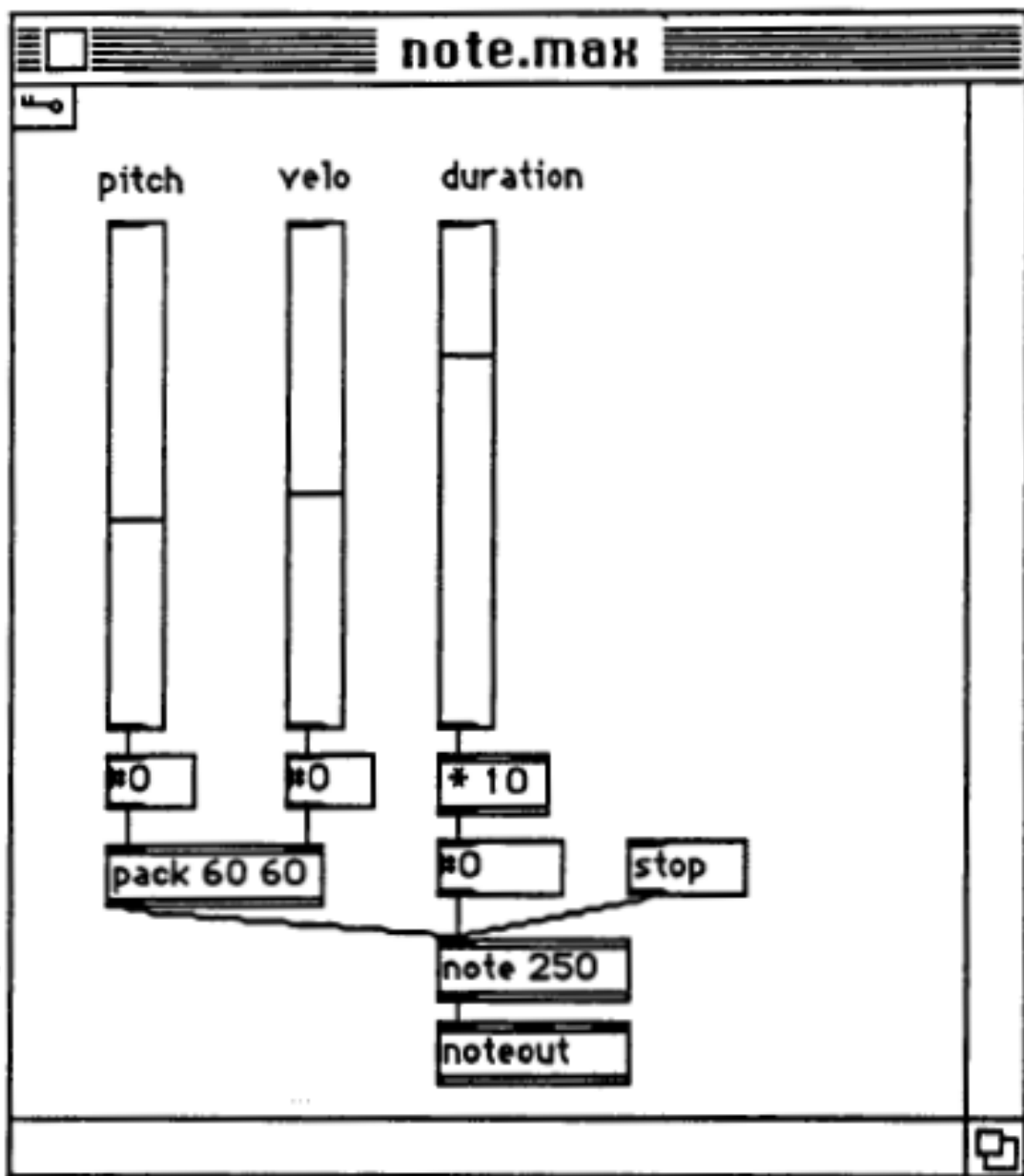


Figure 7.9: An example of a Patcher program - as seen in Puckette's publication ('The Patcher') (228).

7.2 Real-time computer music programming languages and systems



Figure 7.10: An IRCAM Signal Processing Workstation (left) and an i860 board (right) - A photo by Jean-Bernard Emond. This file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license.

operating system for the i860 board (230)¹, the programming environment of MAX hides this machine boundary entirely from the users. Figure 7.11 shows the example of a Max/FTS patch appears in (231).

Besides IRCAM digital synthesizers, there also exist a few similar examples of the hybrid systems composed of the computers and the external DSP platforms. One of the most widely-used systems of the kind is the KYMA/Platypus Computer Music Workstation by Scaletti and his colleagues. Kyma is a Smalltalk-based programming environment that runs on Apple Macintosh II, which controls Platypus, a DSP peripheral (258). Kyma/Platypus also provided a visual programming environment for digital sound synthesis and control (259).

The MARS workstation is another example that consists of an Atari computer and a SM1000 sound generation board, which Di Giugno developed after returning from IRCAM to Italy (20)(72). The MARS workstation was later refined for an IBM-PC and a NERGAL sound generation board (223). Both versions had its dedicated visual programming environment for sound processing and performance controls.

¹As this version of Max collaborates with the FTS program as described, it is often referred as Max/FTS today.

7. APPENDIX II: A BRIEF HISTORY OF COMPUTER MUSIC LANGUAGES AND SYSTEMS - THE SYNERGY BETWEEN TECHNOLOGY AND CREATIVITY

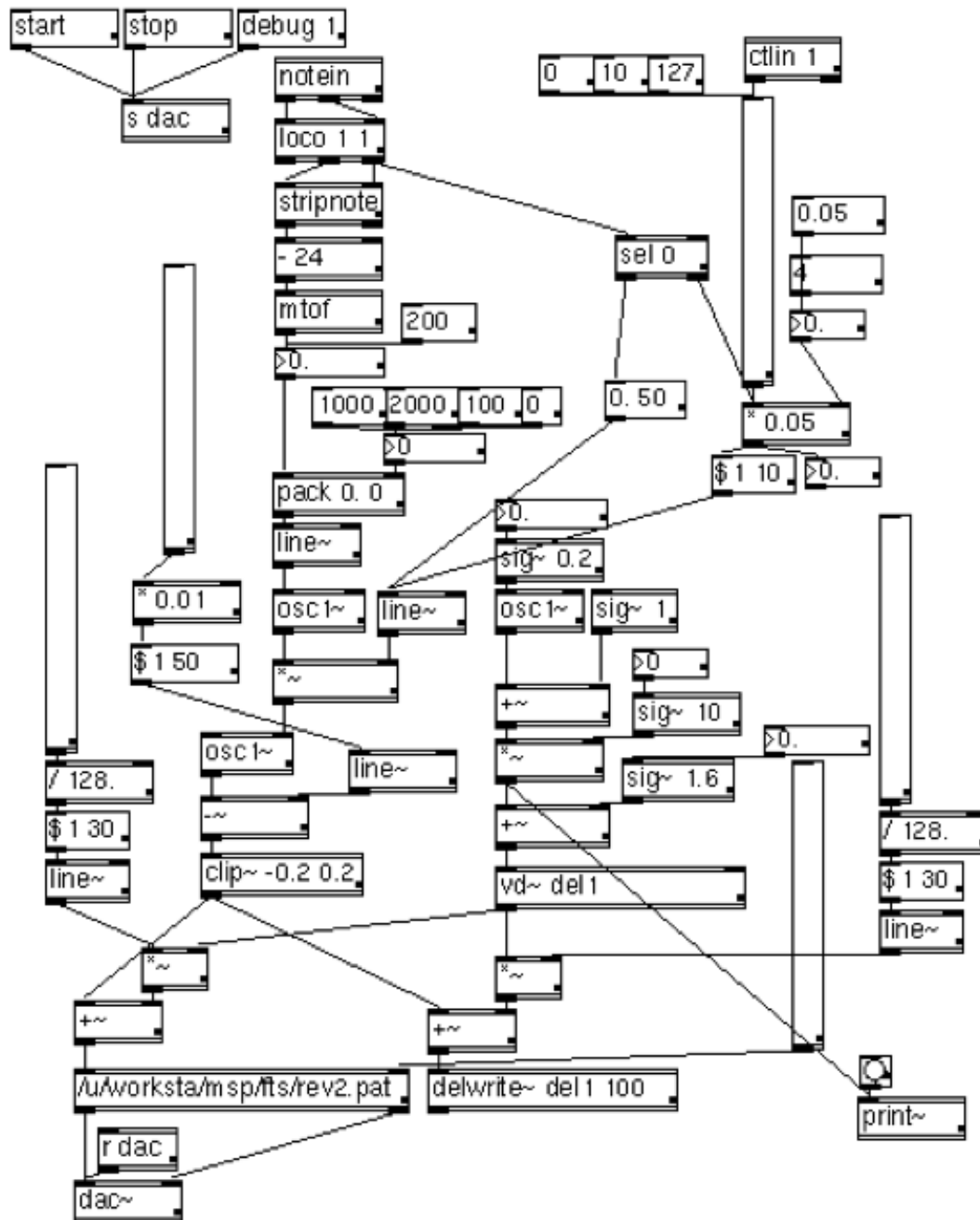


Figure 7.11: A Max/FTS patch example - as seen in Puckette's publication ('Combining Event and Signal Processing in the MAX Graphical Programming Environment') (231).

7.2 Real-time computer music programming languages and systems

Another noteworthy example is ‘a general-purpose digital synthesizer’ developed by Samson in the late 1970s (255). The digital synthesizer is referred to by the name ‘Samson Box’ (191), and CCRAMA is particularly known for developing several languages/software, such as the Pla language (261) and MUSICBOX (190), and for its uses in creative practices.

In addition to the examples of the development of new programming environments and systems for computer systems with external DSP hardware as above, the researchers of the time also extended the existing computer music languages with the external DSP hardware for better DSP performance. *Extended Csound* developed in 1996 by Vercoe and his colleagues falls into this category. It was a real-time variant of Csound with the custom hardware with a SHARK DSP chip (304). Yet, unlike the IRCAM workstation, Extended Csound did not involve the development of the custom DSP processors and made the best use of commercial DSP processors; such an idea may be the precursor to the recent research that utilizes GPGPU (General Purpose Graphic Processing Unit) for computationally expensive sound rendering. For instance, Cowan and Kapralos used GPGPU for real-time spatial sound rendering (84).

7.2.3 MIDI-based interactive computer music systems

As the real-time sound synthesis still required expensive external hardware in this era, the commercial digital synthesizers became rapidly affordable, and computers were already at least fast enough to react to non-audio musical events such as MIDI messages. Interactive algorithmic composition is made possible to perform on MIDI-based live computer music systems, and composers began building their own interactive computer music systems.

Some composers developed their own systems from scratch in general-purpose programming languages, such as *Cypher* (250)(251) by Rowe (developed around 1990 for Macintosh) and *AUTOBUSK* (32)(33) by Barlow (developed between 1986-2000 for PDP-11, ATARI ST etc.). Figure 7.12 shows a screenshot of Autobusk. Yet, computer music programming languages and libraries for algorithmic composition were also eagerly developed for MIDI-based systems. The list includes MIDI-LISP by Boynton and his colleagues (55), HMSL by Polansky and his colleagues (226), Keynote by Thompson (296), Common Music by Taube (291), Moxie by Collinge (76), and the CMU MIDI

7. APPENDIX II: A BRIEF HISTORY OF COMPUTER MUSIC LANGUAGES AND SYSTEMS - THE SYNERGY BETWEEN TECHNOLOGY AND CREATIVITY

Toolkit by Dannenberg (88).

Among these languages and systems, FORMULA (Forth Music Language) (15) by Anderson and his colleagues is of particular interests. While FORMULA was based on Forth programming language (161), the related works of FORMULA even included the development of an operating system (16) to support precise timing behaviour for the interaction with the external MIDI synthesizers¹.

Furthermore, IRCAM also licensed the MAX programming environment to Opcode Systems in 1989 for commercialization. After MAX became publicly available in 1990, many computer music composers began building their own interactive computer music systems in MAX. It was possible to explore the new domain of interactive music, even without expert programming skills and expensive hardware.

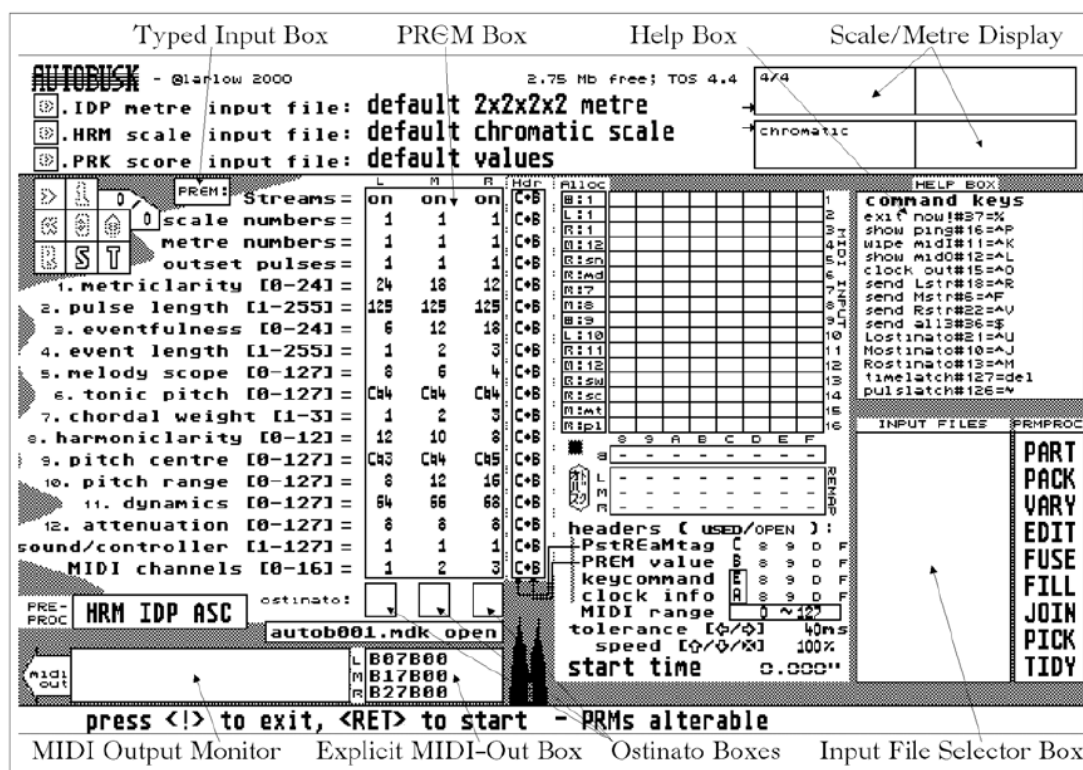


Figure 7.12: A screenshot of AUTOBUSK - as seen in Barlow's publication ('AUTO-BUSK: A REAL-TIME PITCH & RHYTHM GENERATOR') (33).

¹How FORMULA is designed regarding precise timing behaviour is described in Section 2.2.

7.2.4 The development of standalone real-time computer music programming languages

While real-time computer music languages and systems were first made possible by external DSP hardware, further advance in technology made computers fast enough to perform real-time DSP without such external hardware.

Around the beginning of the 1990s, researchers and the engineers began redesigning the existing computer music languages for real-time sound synthesis. Vercoe and his colleagues developed the real-time variants of Csound in 1990, one of the most popular computer music languages around that time (305). Garton and Topper developed RTcmix, the real-time versions of Cmix in 1995 (122)(297).

It should be noted that the development of real-time computer music languages often led to the reconsideration of both software design and language design, as discussed in (175) and (122). While non real-time computer music languages can simply process the input data given deterministically before execution, real-time computer music languages must interact with the incoming musical events. It is also required to process compositional algorithms and perform real-time sound synthesis at the same time. Such an issue was clearly beyond what the design of non real-time computer music languages assumed when they were developed.

Unlike these real-time computer music languages derived from their non real-time ancestors, MAX was originally developed as a real-time interactive control for IRCAM 4X and IRCAM Music Workstation by Puckette (234). Max was also evolved into a computer music language with stand-alone real-time for personal computers. Zicarelli made a significant extension called MSP to MAX for the capability of real-time DSP (327). Puckette also developed PureData (232), yet-another visual computer music language. It was released as open source software, first for SGI IRIX and Windows NT workstations (233) and soon ported to the other platforms such as Windows, Linux and Mac OS X.

The development of the stand-alone real-time variants of the existing languages also motivated new practices in computer music live performances. For instance, the development of Max/MSP popularized live processing of instrumental sounds among composers, which used to require the expensive DSP platform (e.g., IRCAM Music

7. APPENDIX II: A BRIEF HISTORY OF COMPUTER MUSIC LANGUAGES AND SYSTEMS - THE SYNERGY BETWEEN TECHNOLOGY AND CREATIVITY

Workstation¹ and KYMA/Platypus); by the time Max/MSP was commercially released, even laptop computers were fast enough to perform digital signal processing.

In the late 1980s, live digital signal processing of instrumental sounds as is seen in *Jupiter* by Manoury (208) or *Répons* by Boulez (54) was still only available for the composers that can be assisted by the academic institution with a large budget. However, the same signal processing and compositional techniques were made realizable on personal computers in this era. The end-user friendly visual programming environments in MAX and PureData may have also helped the rapid popularization of live computer music practices.

At the same time, as the technology became more affordable, many musicians in the techno and noise music community began using computer music software developed in academia on their laptop computers (43)(237). Such involvement of computer programs in the process of musical creations had a large influence on their musical styles as discussed by Cascone in *the Aesthetic of Failure* (70).

Furthermore, the popularization of real-time computer music programming environments in this era even invoked a question among the researchers about “how to articulate computer generated music in a concert setting” (288)² and the research on new interfaces began attracting considerable remarks, both from the researchers and from the artists in the following years. It is one of the topics of significant interests in the computer music community today³.

7.2.5 Software libraries for digital sound synthesis

The other approach taken in this era was to modularize the DSP functionality as software libraries for general-purpose programming languages. Such modularization made it possible to describe interactive control algorithms in hosting general-purpose programming languages and to collaborate the algorithms with the other software libraries (e.g., networking and graphics, provided in the programming environment).

NeXT Sound and Music Kit by Smith and his colleagues (developed around 1990) is one of the well-known early examples of this kind (268). Common Lisp Music developed

¹For instance, the original i860 board for IRCAM Music Workstation cost 12,000USD.

²Tanaka discusses this issue in together with his own practices in (288). Tanaka is also known for being a member of *Sensorband* and, together with Karkowski and van der Heide, Tanaka played sensor-based instruments in their live computer music performances (51).

³The International Conference on New Interfaces for Musical Expression (NIME) is held annually since 2001 (<http://www.nime.org>).

7.2 Real-time computer music programming languages and systems

by Schottstaedt in 1991 for NeXT workstations (262) is another well-known early work in this category. The list of the recent DSP frameworks and libraries for computer music includes Synthesis Toolkit (STK) by Cook and Scavone (82)(260) for C++¹, JSyn by Phil Burk (65) for Java (written in C++ and Java), CSoundXO for Python (176), CLAM for C++ (11) and Marsyas for C++ (299).

At the same time, it should be noted that such an idea of modularization also influenced the real-time variants of computer music programming languages. While the extension was also made to the original specification of non real-time languages, researchers and developers often reconsidered the whole software architecture and rewrote a significant amount of the implementation so that the DSP features could be used as software libraries in general-purpose programming languages, or even in other computer music programming environments. For instance, Lazzarini describes such an approach in the development of Csound5 in (175).

Cmix was particularly notable in that sense. Even though it was equipped with its own MINC scripting language, the design of Cmix also considered such library-use as design criteria even in the original version (169)(170), and this design concept was taken over into RTcmix; for instance, both CSound and RTcmix are integrated into Max/MSP programming environment as external modules.

7.2.6 New exploration in computer music programming language design

Meanwhile, new computer music programming languages and environments that are not based on the existing ones were also developed in this era. One of the remarkable trends around that time is that these new languages were designed as expressive as general-purpose programming languages, while they were still equipped with the domain-specific features for computer music applications.

Some of these languages were developed as internal domain-specific languages (DSLs) in the existing general-purpose languages, by providing the macros and software framework/library², while others were developed as external domain-specific languages with

¹STK also provides a simple scripting language called SKINI (82).

²“An internal DSL is a DSL represented within the syntax of a general-purpose language. It’s a stylized use of that language for a domain-specific purpose” (114, p.15). “Internal DSLs morph the host language into a DSL itself - the Lisp tradition is the best example of this” and are “often called

7. APPENDIX II: A BRIEF HISTORY OF COMPUTER MUSIC LANGUAGES AND SYSTEMS - THE SYNERGY BETWEEN TECHNOLOGY AND CREATIVITY

own syntax and semantics¹. For instance, languages like ChuckK and SuperCollider are examples of external DSLs, as they have their own syntax/semantics, together with their own compiler/interpreter and virtual machines. Furthermore, some of the recent computer music languages are even built on an existing computer music language, rather than a general-purpose programming language. For example, the *ixi* language (195) is an external DSL, as it has its own syntax and semantics, and *ixi* programs are translated to SuperCollider program by its own interpreter.

Beside such expressibility, these new languages tend to remove the traditional distinction between score and orchestra, intending seamless sound synthesis control by compositional algorithms.

While it was not a real-time interactive application, Nyquist is one of the notable examples of this kind, which is a precursor to the language design criteria in the following years, in that it explores the idea to integrate sound synthesis and compositional algorithms into one environment. Nyquist was developed by Dannenberg in 1997, as an internal DSL for sound synthesis and music composition built on LISP (89) after exploring the ideas related to behavioural abstractions in his previous works such as Arctic (86) and Canon (87). Nyquist is considered one of “the first computer music programming languages that remove the distinction between the ‘orchestra’ and the ‘score’”² (312).

McCartney developed SuperCollider (210)(211)(320), a real-time computer music language, in 1996. SuperCollider is another example of the efforts to integrate sound synthesis and compositional algorithms seamlessly in the language design. Figure 7.13 shows a screenshot of SuperCollider (of its first version). SuperCollider was designed as

‘embedded DSLs’” (115).

¹“An external DSL is a domain-specific language represented in a separate language to the main programming language it’s working with. This language may use a custom syntax, or it may follow the syntax of another representation such as XML” (114, p.15). “External DSLs are written in a different language than the main (host) language of the application and are transformed into it using some form of compiler or interpreter” (115).

²While such an argument may be fair in that Nyquist provides sophisticated abstractions for seamless integration between sound synthesis and musical control, it should be emphasized that some previous works also considered similar design to integrate compositional algorithms and sound synthesis in one language, as seen in MINC of Cmix (169) and RTcmix (122) or in Max (231) and PureData (232).

7.2 Real-time computer music programming languages and systems

a new object-oriented programming language with a significant influence from Smalltalk (126).

However, in the most recent version (SuperCollider 3), the whole programming environment of SuperCollider is divided into two different programs. One is *sclang*, which executes the control algorithms, and the other is *scsampler*, a real-time sound rendering server. While the communication between these two programs is done by Open Sound Control protocol (116)(324), and is usually hidden inside the class libraries, as discussed in Section 2.2, this can cause a significant obstacle when precise timing behaviour is required to control sound synthesis algorithms.

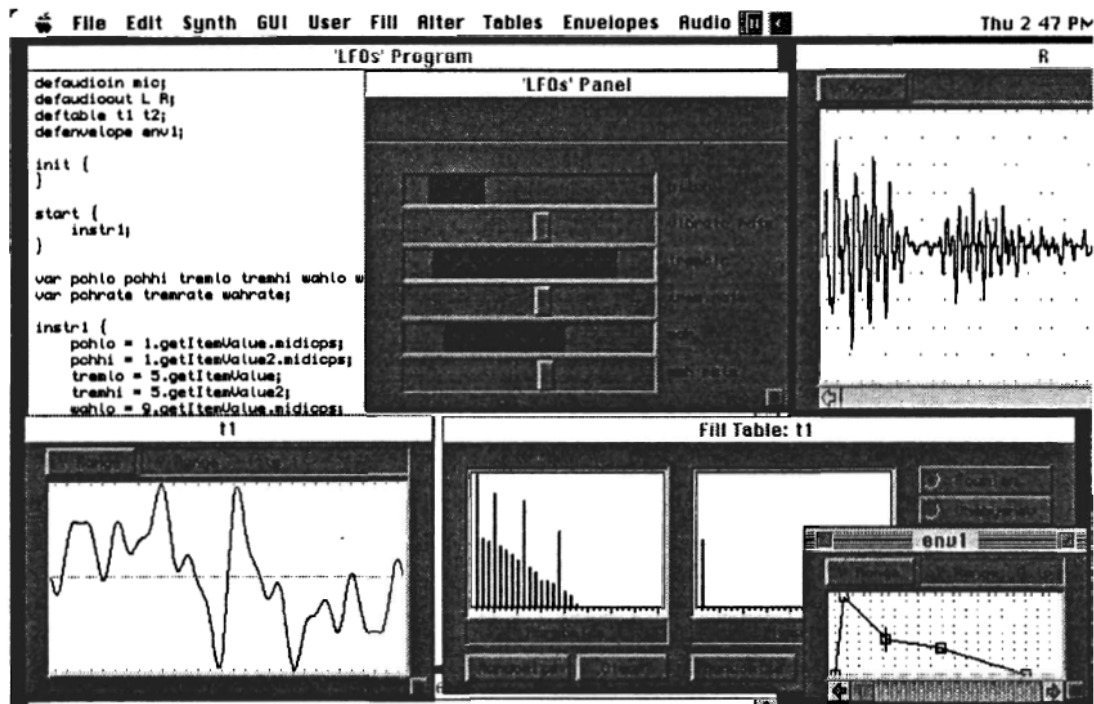


Figure 7.13: A screenshot of SuperCollider - as seen in McCartney’s paper ‘SuperCollider: a new real time synthesis language’ (210).

Sorensen and his colleagues developed Impromptu (63)(274)(276) in 2005, an internal DSL built on the Scheme programming language (281)(278). Figure 7.14 shows a screenshot of the early version of Impromptu.

Impromptu’s sound synthesis functionality depends on AIME (274), a C++ synthesis/scheduling engine Sorensen developed, for its early versions, and on *AudioUnits* (*AUs*), Apple’s sound synthesis framework (7) for the recent versions. While such soft-

7. APPENDIX II: A BRIEF HISTORY OF COMPUTER MUSIC LANGUAGES AND SYSTEMS - THE SYNERGY BETWEEN TECHNOLOGY AND CREATIVITY

ware design makes Impromptu able to utilize many commercially available AU modules, this also led to limitation in the exploration of sound synthesis algorithms. Impromptu does not offer the flexibility to describe various sound synthesis algorithms as much as unit-generator languages and a user mainly can use such ready-made AUs. Yet, the successor language, *Extempore* provides the features to write and compile a new sound synthesis module within its environment and allows more flexibility at the sound synthesis level (277).

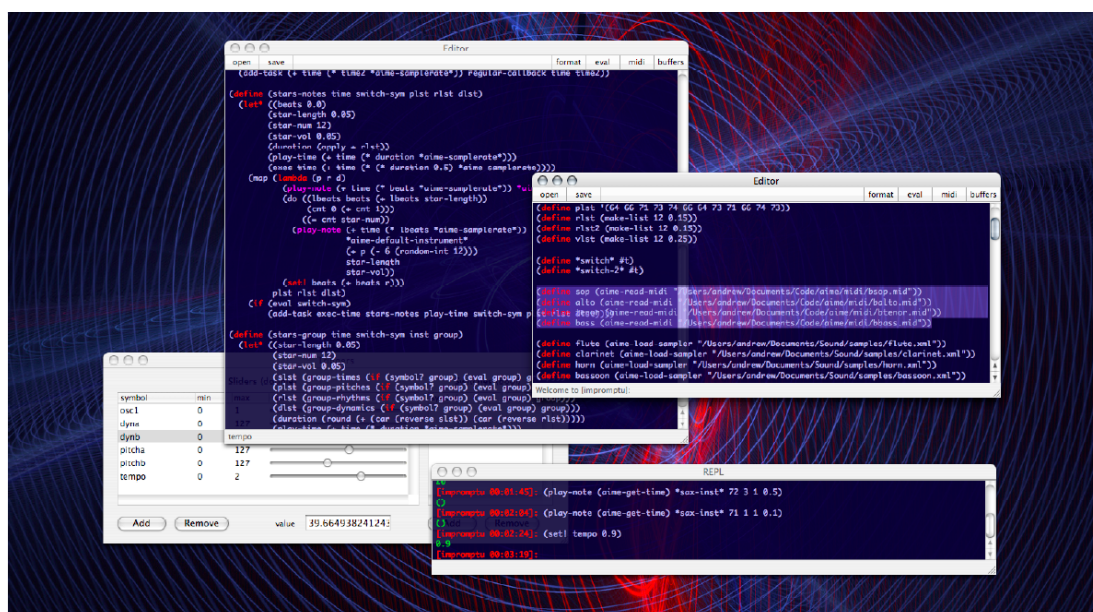


Figure 7.14: A screenshot of Impromptu - as seen in Sorensen's paper 'Impromptu: An interactive programming environment for composition and performance' (274).

ChucK, developed by Wang and his colleagues (312)(314)(315) in 2003, is also one of the most notable recent MUSIC-N descendant languages, especially for its *strongly-timed programming* concept. Figure 7.15 shows a picture of miniAudicle, an integrated development environment for the ChucK programming language.

Unlike many other computer music programming languages (and many general-purpose programming languages), the strongly-timed programming concept integrates the explicit control of the advance logical time into an imperative programming language, as a variation of the synchronous programming concept. By such an explicit control of logical time, a strongly-timed program can guarantee the precise timing behaviour in logical time with sample-rate accuracy, the lack of which is one of the most

7.2 Real-time computer music programming languages and systems

significant problems in computer music language design. The strongly-timed programming concept is discussed in detail in Section 2.2.4.

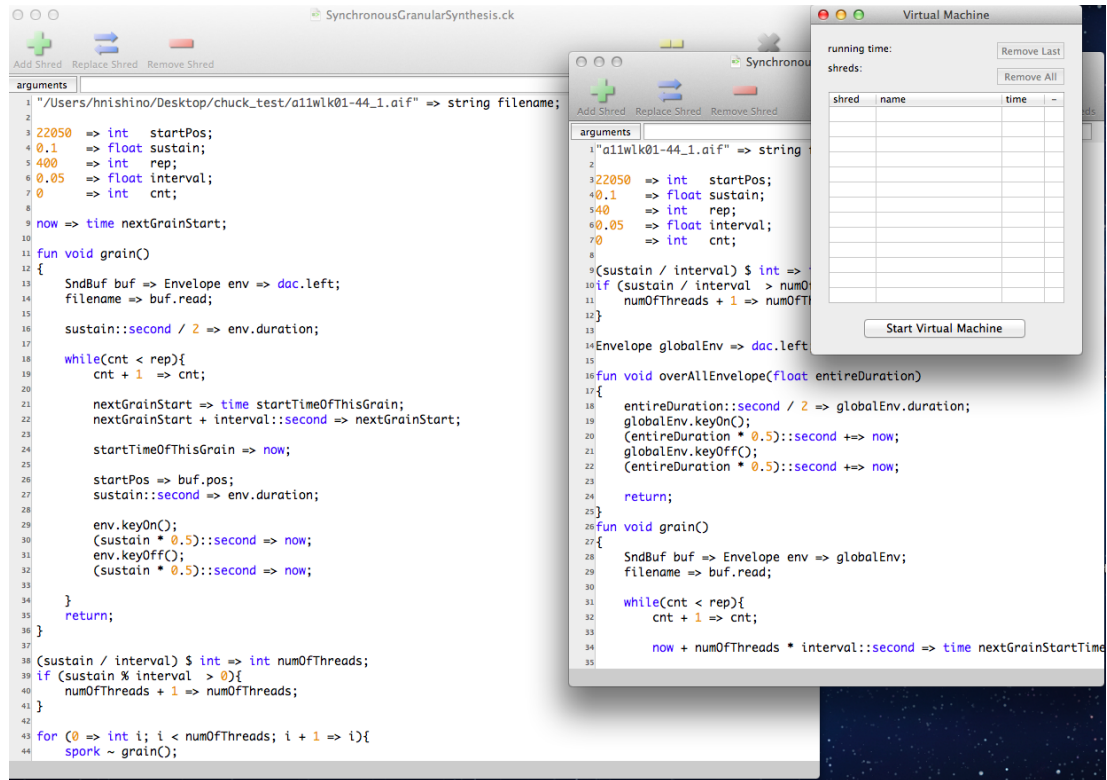


Figure 7.15: A screenshot of miniAudcle

, an integrated development environment for ChuckK

One of the most notable influences that these new computer music programming languages made to the creative practices, would be the emergence of *live-coding* performances. In live-coding, the performers write and modify computer music programs on-the-fly on stage (43)(62)(77)(216). While the origin of live-coding can be found in the earlier history of computer music¹, the emergence of the new computer music programming languages, which can interactively program compositional algorithms and perform sound synthesis in real time, largely contributed to the recent flourish of live-coding practices. The computer music software frameworks/libraries that can be accessed from general-purpose scripting languages are also often involved in live-coding

¹For instance, the papers such as (77)(318)(329) describe or refer to the earlier experiments in live-coding by the Hub and Ron Kuivila.

7. APPENDIX II: A BRIEF HISTORY OF COMPUTER MUSIC LANGUAGES AND SYSTEMS - THE SYNERGY BETWEEN TECHNOLOGY AND CREATIVITY

performances(77)(318).

The list of more recent examples of real-time computer music programming environments includes PWGL, LuaAV, Faust, Overtone, Haskore, and ixi.

PWGL (174) is yet-another visual computer music programming environment written in LISP, which Laurson and his colleagues developed around 2002, following their previous work called PatchWork (173). They argue that “PWGL offers the flexibility of both a traditional text-based programming language and a visual programming language” as it “provides a direct interface to its base languages, every Lisp function or CLOS¹ method can be automatically transformed into a box” (174).

Wakefield and his colleagues developed internal DSLs, Vessel (308), and LuaAV (307) in the Lua programming language (152)(153)(154). LuaAV also provides sample-rate accuracy in logical time as seen in ChuckK by the scheduling framework that involves coroutines², together with the extension for visual expression.

Orlarey and his colleagues has been developing Faust since the early 2000s (219)(221). Unlike the previous examples, Faust translates a program, which describes DSP algorithms, to its equivalent C++ source code. The C++ source code can be then compiled to work as an external library in many different environments such as PureData, Max/MSP, SuperCollider, etc (220).

Overtone (3) by Aaron is an internal DSL built on Clojure, a general-purpose functional programming language (136)(140). Overtone depends on *scs_server* (SuperCollider’s sound synthesis engine) for its real-time digital signal processing capability.

Haskore (149), originally developed by Hudak in 1996, is another internal DSL built on Haskell, a functional programming language (295). While the original version of Haskore lacked the real-time sound synthesis capability, the extensions were made to utilize the external real-time sound synthesis engines (e.g., CSound, SuperCollider (*scs_server*), etc (293)).

The ixi language is a DSL developed by Magnusson built on SuperCollider (195), a program of which is translated to a SuperCollider program. The ixi language focuses on live-coding improvisation in the style of techno-music like minimalism.

¹CLOS stands for Common Lisp Object System(49).

²For the concept of ‘coroutine’, see (80).

7.2.7 The emergence of mobile platforms

The recent development of mobile devices such as iPhone and Android have also had a large impact on computer music programming languages and systems. Many computer music languages are being ported to mobile devices today.

For instance, Allison and his colleagues reported that their project involved the Android versions of SuperCollider developed by Shaw (9). Brinkmann and his colleagues ported Pure Data to mobile devices (58). Wang describes the development of ChiP, the iPhone versions of ChuckK and its use for their commercial applications in (313)(317). Mailman also reported the use of iRTcmix, the iPhone port of RTcmix for computer music applications (196).

While the development of mobile computer music languages and systems is also contextualized in the context of locative arts (139) as described in (123), on the other hand, many recent works place mobile devices in the context of the research of new interfaces for musical expressions (NIME). As many mobile devices that we have today integrate the multi-touch interface and various kinds of sensors inside, together with the network interface, the mobile devices attract significant interests as NIME. As the recent devices are fast enough to perform real-time DSP, the mobile devices can host stand-alone computer music systems and often be involved in live computer music performances¹.

7.3 The synergy between technology and creativity

While computer music programming languages and systems have been continuously evolving in parallel with the advance of computer technology, it should also be strongly emphasized that there has existed a synergy between technological advance and creative musical practices, throughout the history of computer music.

Even in the earliest era when computer music languages were executed on the main-frame computers with punch cards, the researchers developed the domain-specific core abstractions (the unit-generator concept and the score-orchestra model) so that the composers of the time can explore the new domain of digital sounds, as end-users without programming skill. The desire for live computer music in the early era led to the research on hybrid computer music systems composed of minicomputers and external

¹Tanaka discusses such an aspect of the mobile devices as NIME (289).

7. APPENDIX II: A BRIEF HISTORY OF COMPUTER MUSIC LANGUAGES AND SYSTEMS - THE SYNERGY BETWEEN TECHNOLOGY AND CREATIVITY

analogue synthesizers, the concept of which is taken over by MIDI standards for digital synthesizers.

Both efforts were taken over into the next era when highly-programmable external DSP hardware was developed. As computer music systems became more complex and interactive, this raised the demands to make a computer music programming environment more end-user friendly and appropriate for rapid-prototyping in this era. This demand led to the development of interactive visual programming environments such as MAX and Kyma. Supported by real-time DSP capability and rapid-prototyping environments, the composers of the time explored the new territories of musical creations, as seen in Pierre Boulez's *Répons*, which involves live processing of the instrumental sounds.

While external DSP hardware was not yet affordable to individual composers, the emergence of MIDI and digital synthesizers also helped the exploration of the novel field of interactive computer music, even with the significant limitation in the flexibility of sound synthesis. The research on the hybrid computer music systems and algorithmic compositions in the previous era founded the basis for interactive music systems. The commercialization of the MAX programming environment also helped such practices without solid programming skills.

In the succeeding decades, the further advance in technology made it possible to perform real-time DSP even on personal computers. The popularization of real-time DSP largely influenced computer music language design; interactivity and real-time DSP became the essential criteria in new computer music languages. Two separate efforts, the research on the language for interactive/algorithmic compositions and the languages for real-time digital signal processing fused into one and the distinction between score and orchestra began to be reconsidered.

After interactive music with real-time sound synthesis was made possible, the composers began seeking means to control digital sound synthesis more expressively. The interests in new interfaces for live computer music systems rapidly grew and led to the research on new interfaces for musical expression (NIME). The NIME research also founded the basis for the recent applications of mobile devices for musical practices.

7.3 The synergy between technology and creativity

Furthermore, as computer music programming languages became even as expressive as general-purpose programming languages by importing the programming concepts such as object-oriented programming, expressiveness and interactivity in the recent computer music languages induced the rapid growth of live-coding practices, which began significantly influencing the design of new computer music programming languages.

Thus, computer music languages and systems have evolved through the synergy between technology and creativity. The advance of technology has often led to the emergence of novel creative practices, which often motivated further research by revealing the problems in the existing languages and systems.

**7. APPENDIX II: A BRIEF HISTORY OF COMPUTER MUSIC
LANGUAGES AND SYSTEMS - THE SYNERGY BETWEEN
TECHNOLOGY AND CREATIVITY**

8

Appendix III: the Implementation of the Proof-of-concept Prototype of LC

While the implementation issues such as performance efficiency are not a topic of significant interest in this thesis, the implementation of the proof-of-concept prototype is briefly described in this chapter.

8.1 System architecture

In the proof-of-concept prototype of LC, the programming environment of LC consists of two software components: the LC Editor and the LC Virtual Machine. The overall architecture of the proof-of-concept prototype is shown in Figure 8.1. These two are different software and independent from each other.

8.2 LC Editor

LC Editor is a front-end application with a text editor. It is entirely written in JAVA, as the compilation phase does not require significant performance efficiency in comparison with the LC virtual machine. Users write programs in the text editor and the integrated compiler compiles the selected part of a program into LC's bytecode and writes it to a temporary file. When the compilation is finished, the LC Editor sends a message to the LC virtual machine to load the bytecode file via UDP socket. Other messages such

8. APPENDIX III: THE IMPLEMENTATION OF THE PROOF-OF-CONCEPT PROTOTYPE OF LC

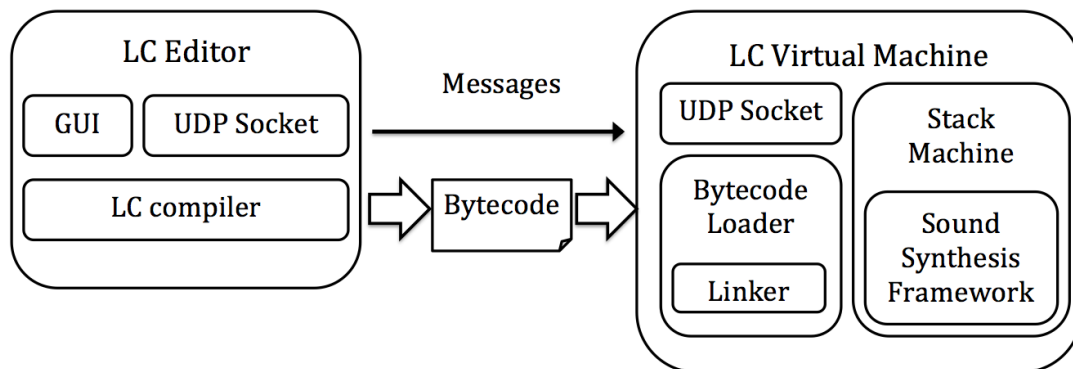


Figure 8.1: The overall system architecture of LC (the proof-of-concept prototype)

as the termination of all the active threads and patches can be also sent from this text editor.

Figure 8.2 is a screenshot of the LC Editor. As shown, the proof-of-concept prototype version of the editor is still quite simple. However, even in such an editor, a user can write a program text and immediately execute any part of the code. Any redefinition made to patches, functions, or variables can be performed on-the-fly; for instance, if a granular synthesis program is being executed, referring to a global variable ‘\$p’ for its pitch, the assignment of a new value to ‘\$p’ may immediately trigger the change in the pitch of the output sound.

8.3 LC Virtual Machine

LC Virtual Machine is a virtual machine that executes LC bytecode. When the virtual machine received a message via UDP socket, it immediately loads the bytecode from the file (from the path given by the message) and then its linker resolves the symbols such as function names or variable names. After this link phase is finished, the bytecode is passed to the stack machine. LC Virtual Machine is entirely implemented in C++.

The stack machine immediately executes the bytecode, launching a new software thread. As the sound synthesis framework is integrated within the stack machine and the execution of LC’s software threads are performed within the same native thread as real-time DSP, LC’s software threads can precisely synchronize its behaviour with real-time sound synthesis, as required for mostly-strongly-timed programming in LC.

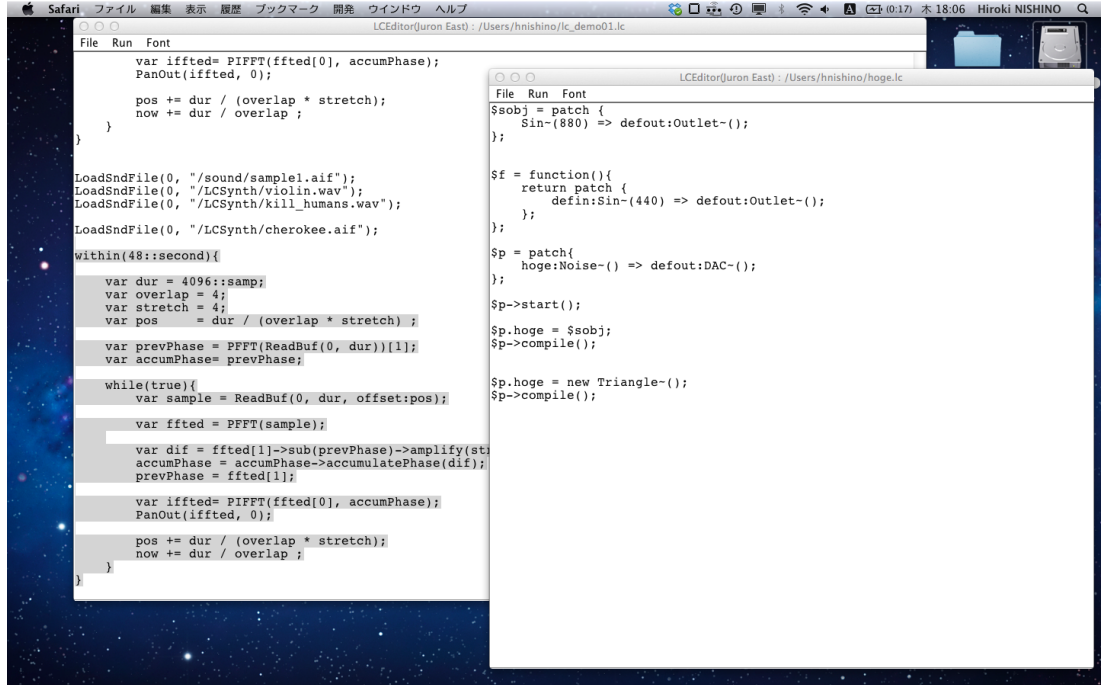


Figure 8.2: A screenshot of LC Editor (the proof-of-concept prototype)

As LC is a programming language with automatic garbage collection, LC Virtual Machine implements a simple incremental mark-and-sweep garbage collection, known as Yuasa’s snapshot-at-the-beginning algorithm (326).

8.4 Latency issues

The actual latency of a computer music system highly depends on the implementation. In this section, we briefly describe some additional discussion on this issue, together with the information on the current LC’s prototype.

First of all, since the latency depends on many different layers of a computer system and can not be fully controlled by a user program (e.g., a computer music language or application). The latencies that are not caused by a user program, such as the latencies caused by an audio device, hardware interface, driver software, operating systems, can hardly be improved by a user program. Thus, in the following discussion, the latencies caused by such factors are excluded and we focus only on the latencies that can be caused by a user program, for instance, the virtual machine and compiler of LC.

8. APPENDIX III: THE IMPLEMENTATION OF THE PROOF-OF-CONCEPT PROTOTYPE OF LC

When considering the response to a user program from the programming environment, there can be some delay caused by the compilation to the bytecode, sending the path information of the bytecode to the virtual machine. The phases to load the bytecode onto the virtual machine and to resolve the symbols (linking) also lead to some overhead before the user program. While the latencies caused by such factors may be improved by efficiently implementing the compiler/virtual machine and designing the bytecode file format to be more suitable for the runtime environment, these factors can cause some delay in interaction to users.

However, as LC is based on the ideal synchronous hypothesis¹, once the virtual machine begins the execution of the user program, the program is executed with sample-rate timing precision.

Another issue to consider is the latency between the audio input and audio output. While this kind of latency can be caused also by an audio device, hardware interface, driver software, and the underlying operating systems, a user program can cause some additional overhead, depending on how it is designed.

The most typical design of the audio I/O APIs is to provide the mechanism to call back a user-defined function when the next chunk of the samples are required for the output (or when the required amount of the input samples are ready). There are two widely-used strategies for how a user program provides the audio output samples (or handle the audio input samples).

The first strategy is simply to handle everything within the callback function. In this case, the latency caused by the user program can be minimized. However, as there is a deadline to finish the callback so that the next chunk of the samples can be passed for the sound output, the duration that can be used within the callback function can be limited. The problem of this strategy is that there can be less time for the background tasks. LC's mostly-strongly-timed programming allows a user to asynchronously execute the parts of a user code without synchronizing with the logical synchronous time when enclosing them within *async* blocks; LC provides the feature to execute time-consuming tasks in the background.

When handling everything within the call-back functions, because of the deadline to provide audio samples before the next DSP cycle, the time that can be spent for such

¹See Section 2.2.4 for the definition of the ideal synchronous hypothesis.

background tasks can be significantly limited; the available duration for the background tasks in each cycle is only the duration between the timing when the required amount of the audio samples are computed and the scheduled tasks are finished and the deadline when the entire callback must be finished; the intervals between the driver callback cycles are simply wasted. Figure 8.3 pictorially describes such an issue.

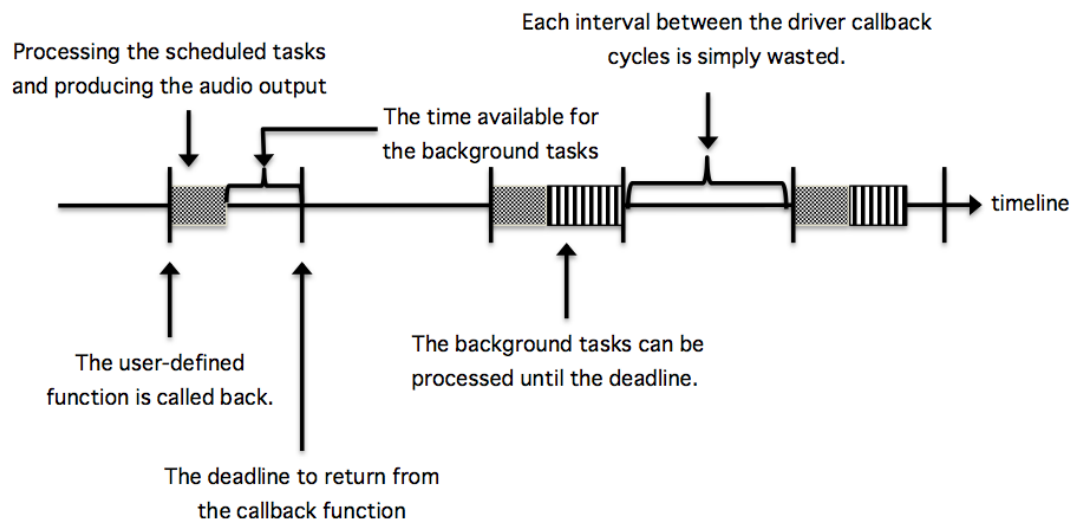


Figure 8.3: The implementation to handle everything within the audio callback function

The other strategy is to involve the ring buffers for the audio computation. The callback function pops the required amount of samples from the output ring buffer and pushes the input samples to the input ring buffer. The audio computation is performed in another real-time thread. The thread pops the input samples from the input ring buffer and then computes the audio output to push the output samples to the output ring buffer. As it must be guaranteed to ensure that the callback function does not starve for the audio output, the real-time thread must compute the audio samples ahead of time. Thus, there can be a certain delay in the audio I/O when compared to the other strategy described above. Figure 8.4 pictorially illustrates such a strategy.

Instead, as shown in Figure 8.5, the real-time thread have more chances to process the background tasks when there are already enough audio samples in the ring buffer for the audio output. By checking the number of the available samples in the ring buffer, the high-priority thread can decide when the background tasks can be performed or when the audio computation must be performed to supply the required amount of the

8. APPENDIX III: THE IMPLEMENTATION OF THE PROOF-OF-CONCEPT PROTOTYPE OF LC

samples.

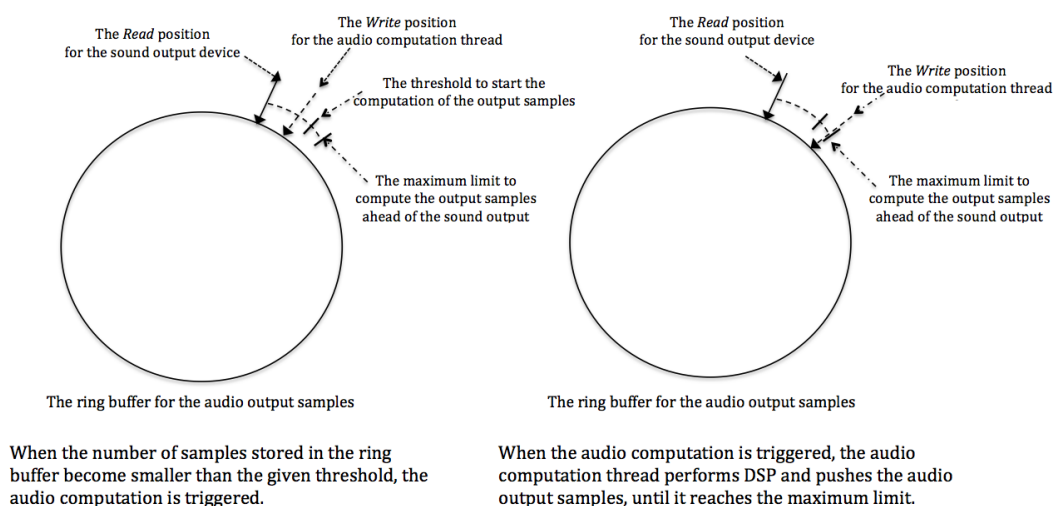


Figure 8.4: How the audio computation is triggered

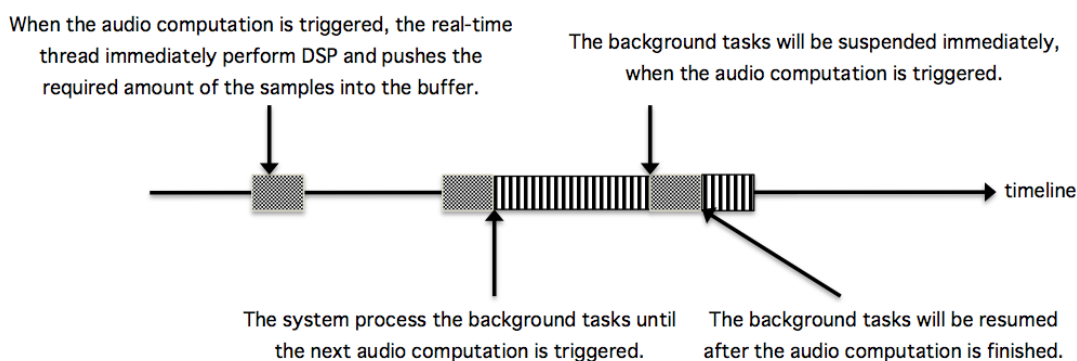


Figure 8.5: The implementation to perform DSP in an real-time thread

In the current version of LC, as we took the latter strategy at the cost of a certain latency so that the asynchronous (or background) tasks have more opportunity to be processed. However, it may be desirable to provide the option for users to explicitly select which strategy that the virtual machine should take, as the former strategy may be preferable in a certain musical context, for instance, live signal processing of an acoustic instrument. We leave such an issue for the future version, as the thesis focuses on the language design of LC and the implementation is only to prove the concepts are

realizable.

8.5 The issues related to the performance efficiency

8.5.1 Audio vectors

The use of audio vectors (53, p.467) in a unit-generator language can improve the performance efficiency. However, if the sample-rate accurate timing precision is required, the output must be computed sample-by-sample instead, since the state of a unit-generator (e.g., frequency and amplitude), can be changed at sample-rate. Thus, audio vectors can not be used in such a situation and performance efficiency can be much worse because the audio computation is performed sample-by-sample. The code optimization by SIMD instructions (145) can also hardly be applicable.

On the contrary, in LC's microsound synthesis framework, because *Samples* is an immutable object, no modification can be made to the samples within a *Samples* object after its instantiation. Thus, the audio computation for *Samples* objects can be performed block-by-block and the code optimization by SIMD instructions can be involved when applicable, even with the sample-rate accurate timing behaviour.

8.5.2 Parallelism

The current proof-of-concept prototype of LC is concurrent but not parallel. LC's multi-threading is performed by its software threads (green thread or lightweight concurrency) as described in Section 3.1.9 in the same native thread, to achieve the sample-rate accurate timing precision.

However, multicore processor parallelism is a different issue from multi-threading by lightweight concurrency. Making LC parallel requires the further consideration in its design, since the recent APIs for multicore processor parallelism, such as Grand Central Dispatch by Apple(254) or Intel threading building blocks (239), dispatch the sub tasks to the threads, each of which is assigned to one of the CPU cores.

While such a model can significantly improve performance efficiency in many cases, in the situation that sample-by-sample computation of the output samples is required, the overhead of task dispatching can be a problem when the cost of dispatching can be bigger than the performance efficiency obtained by multicore processor parallelism. It was observed that the use of Grand Central Dispatch to compute the output from

8. APPENDIX III: THE IMPLEMENTATION OF THE PROOF-OF-CONCEPT PROTOTYPE OF LC

the unit-generator graphs led to significant damage to overall performance efficiency rather than to the improvement in our experiment with the proof-of-concept prototype.

Max 6 runs every top-level patcher in its own thread for better utilization of multicore CPUs, however, such an strategy can not be directly applicable to a textual programming language. Also, multi-threading may damage the timing precision when the synchronization between these threads are performed; thus, one of the challenges in the parallelisation of a computer music language is to achieve the performance efficiency by multicore processor parallelism while maintaining precise timing behaviour.

The partitioned global address space (PGAS) model(256), as seen in X10 programming language (106) would be one of the suggestions in previous works for how a programming language should be designed for better use of multicore processor parallelism.

The programming languages based on PGAS model, “permit the programmer to think of a single computation running across multiple processors, sharing a common address space. All data resides at some processor, which is said to have affinity to the data. Each processor may operate directly on the data it contains but must use some indirect mechanism to access or update data at other processors. Some kind of global barriers are used to ensure that processors remain roughly synchronized.” (257).

Such a language design that considers multicore CPUs and affinity in its programming model may be also beneficial for computer music languages. For instance, by assigning one virtual machine to each CPU core and providing some means to communicate between them would make it possible to maintain the sample-rate accurate timing behaviour within each virtual machine, while making explicit that the inter communication between the different virtual machines may be less precise in timing to some degree.

Appendix IV: Additional Discussion

The sections in this chapter additionally describes the issues that are less relevant to the topics discussed in this thesis, as they may be beneficial for further investigation in the future.

9.1 The definition of ‘abstraction inversion’ referred in this thesis

It could be argued that the definition of ‘abstraction inversion’ is the problem that occurs when the representation of a certain lower-level abstraction, such as an object or an operation, is not exposed to higher-levels and that it differs from a simple lack of a certain object or operation. In this thesis, we use abstraction inversion in a broader sense as is seen in many previous works and also include the lack of a certain object or operation within the underlying software framework for the following reasons.

First of all, in *opening up Ada-tasking* (28), which is considered the publication in which the concept of ‘abstraction inversion’ was first introduced, the term ‘abstraction inversion’, is used in a much broader sense. The definition found in this publication is as follows: “abstraction inversion occurs when a programmer is forced to use a combination of higher-level abstractions to express a lower-level abstraction” (28). The example that immediately follows in this definition is the famous Ada’s rendezvous

9. APPENDIX IV: ADDITIONAL DISCUSSION

example, which is described in Section 2.3 in this thesis, and the problem of Adas rendezvous example seems obviously caused because Ada does not expose a simple mean directly to obtain mutual exclusion at the level of a user program; this is likely the example referred to the most frequently when discussing ‘abstraction inversion’ and it also seems to fit the narrower definition of abstraction inversion; the problem is caused because of the inexistence of a lower-level object, which is implemented under the abstraction barrier within the same software framework.

Yet, other examples and expressions by Baker suggest his original concept of abstraction inversion is considered in a much broader sense beyond the problem of the inexistence of the lower-level representations. For example, he describes another example of buffering in Ada as follows: “Buffering. In Ada, this involves an intermediary task, whereas for a single producer and a single consumer it can be implemented safely at the machine level without even using a semaphore” (28)¹. This is a problem of an inappropriate abstraction rather than a problem of the inexistence of a certain representation in the lower-level.

For example, the expression as flows can be also found in the paper: “Since an abstraction inversion involves expressing a simple operation in terms of a combination of higher-level operations, that inherently more complex, the end program is almost certainly going to be inefficient. Moreover, information is lost. The resulting code will not clearly reflect the original thinking of the programmer. Anyone reading it will be forced to infer the programmer’s intent. The difficulty is similar to trying to infer the underlying mathematics of a FORTRAN program by reading the code” (28).

As seen in the above expressions, Baker’s original concept of abstraction inversion is used in a broader sense, which also includes the lack of a certain object or operation, and both the performance inefficiency and the comprehensibility of the resulting code; clearly, Baker didn’t intend to limit the definition of ‘abstraction inversion’ just within the discussion on whether or not a certain object or operation in the lower-level in the software design is exposed to the higher-levels.

¹Think of the case in which a simple FIFO queue for a single write thread and single reader thread is implemented. Such a FIFO queue can be implemented without a lock mechanism. Yet, Baker discusses that an Ada program must involve a task unit, which is normally utilized for concurrent algorithms, just to implement such a FIFO queue.

9.1 The definition of ‘abstraction inversion’ referred in this thesis

Baker even discusses abstraction inversion in the context of compiler optimization as follows: “one partial solution to the problem of abstraction inversion is to adopt certain tasking “euphemisms” that is, patterns of coding that recognized by compilers and human readers as standard ways of expressing simple non-Ada operations in terms of more complex Ada ones. This is analogous to vectorizing FORTRAN complies, which infer the possibility of vector operations by analysing the loops. Examples of euphemisms which may be optimized include fast interrupt handler tasks, monitor tasks, and buffering tasks” (28).

Such a broader definition of abstraction inversion seen in the original Bakers paper seems still accepted in Ada’s community. For instance, In ‘Ada User Journal Volume 29, No.3’ published in 2008, Rosen describes as follows.

“In C (or Fortran), there is no array assignment. Therefore, compilers are very clever at recognizing patterns that can be optimized, like:

```
for (I = 0; I < N; I++)  
    A[I] = B[I]
```

Note that from a theoretical point of view, there is a real abstraction inversion here, since the compiler recognizes a high level statement (an array assignment) from the detailed description of its implementation” (249).

Fernandez also explains abstraction inversion as follows: “abstraction inversion exists when a simple coordination mechanism is simulated while using a complex one. An example would be simulating a semaphore with rendezvous. Sometimes (but not always) compiler optimizers are able to overcome the penalties of abstraction inversion” (111). Waroquiers and his colleagues also explain abstraction inversion as “using a high level structure to implement a low-level concept” in (319). It should be noted that this expression contrasts ‘a high level structure’ to ‘a low-level concept’, not ‘a low-level structure’; it is implied that they do not are not concerned if a certain object that implements the ‘low-level concept’ actually exists under the abstraction barrier, when describing abstraction inversion.

9. APPENDIX IV: ADDITIONAL DISCUSSION

A similar explanation of abstraction inversion can be also seen in *Critique of DIN Kernel Lisp Definition Version 1.2* written by H.G. Baker in 1992¹. He describes abstraction inversion as the problem, “in which a simpler notion is defined in terms of more complex notions.” (27). The use of the word ‘notion’ clearly suggests Baker considers abstraction inversion at the conceptual level rather than at the implementation level of a certain software framework or library.

Additionally, it seems rare to find publications that clearly limit the definition of abstraction inversion within the inexistence of a low-level object/function to higher levels in the implementation of a certain software framework/library. Many publications seem to discuss the conceptual level issues rather than the implementation issues. *On Some Myths About Network Intelligence* by Minerva is one of the rare examples that include a narrower definition of abstraction inversion. It states that “abstraction inversion [... arises ...] when users of a construct need functions implemented within it but not exposed by its interface. The results that the users re-implement the required functions in terms of the interface, which in turn uses the internal implementation of the same functions”, this definition seen in Minerva’s paper is actually a citation from Wikipedia, which is not very desirable for academic discussion; as described so far, the majority of the academic publications discuss abstraction inversion in a much broader sense.

The concept of abstraction inversion was presented in 1990 by Baker before the notion of software anti-patterns are coined in the developer community. The earliest known publication that mentions the idea of software anti-pattern is ‘*Patterns and Antipatterns*’ by Koenig, which was published in 1995 (165) and ‘*AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*’ (64), one of the most widely-known books on anti-pattern, does not seem to contain the term ‘abstraction inversion’ at all. However, while it may be interesting to investigate how and why the narrower definition of abstraction inversion began to be argued as seen in Minerva’s paper, such an investigation is beyond the scope of this thesis.

¹H.G. Baker should not be confused with Ted Baker, the author of the publication, to which we are referring as the original definition of abstraction inversion (28)

9.1 The definition of ‘abstraction inversion’ referred in this thesis

Thus, the original concept of abstraction inversion described in (28) by Baker is defined and used in a much broader sense, which includes more situations than the inexistence of lower-level objects/functions in the implementation of a certain software framework or library; what Baker and others discuss in their publications is about a more general issue of whether the appropriate abstractions are applied to the software and language design. In this thesis, we refer to abstraction inversion in such a broader sense and include the problem that the lack of certain objects/functions that belongs to the lower-level abstraction in its concept within the sound synthesis framework.

Moreover, in most cases, abstraction inversion in a narrower sense, in which abstraction inversion is defined only as a problem with respect to the inexistence of the lower-level objects/functions, still can be also applied to the sound synthesis framework design when discussing the difficulty in implementing microsound synthesis within unit-generator languages. Many unit-generator languages provide dedicated unit-generators for microsound synthesis techniques¹, and the buffer objects that can load the sound data². Within a microsound synthesis unit-generator, a certain object that represents a microsound is implemented under its interface. The buffer object can represent the sound samples that can represent microsounds, yet it must involve a unit-generator to read from the buffer and the samples within are not directly accessible. Such a situation fits the concept of abstraction inversion in a narrower sense.

As above, the use of *abstraction inversion* in the discussion of the sound synthesis framework design in this thesis can be considered fair both in a broader sense and a narrower sense, as suggested by the previous publications and the actual design of the existing sound synthesis frameworks.

¹For instance, SuperCollider (320) provides *TGrains*, *GrainSin*, *GrainFM*, *GrainBuf*, *GrainIn*, and similar unit-generators. Csound (52) provides *grain*, *granule*, *fof*, *fof2*, *grain3*, *partikkel*, *syncgrain*, *sndwarp*, *sndwarpst*, and similar unit-generators. Max (234) also provides a number of external objects for microsound synthesis, such as *grainbuffer~*, *grainstretch~*, *grain.bang~*, *grain.phase~*, *grain.pulse~*, *grain.stream~*, provided by its user community.

²For example, SuperCollider has the *Buffer* object and Max has the *buffer~* object.

9.2 The HCI related issues

The focus of this thesis is on technical aspects, rather than the HCI aspects of programming language design. However, it would be desirable to briefly discuss the related HCI issues. The following sections describes the topics, which are not within the scope of this Ph.D thesis but may be beneficial to investigate for future work.

9.2.1 The expected users

The thesis does not make any particular assumption on how much expertise in programming is expected for users, as it focuses on the technical aspects of computer music programming language design. While such HCI-related issues are not within the scope of this thesis, we briefly provide the additional discussion on the expected users in this section.

Generally speaking, the skill sets of a programmer significantly depends on his/her individual experience. This is the same in computer music programming; such an argument can be found even in ‘the technology of computer music’, one of the earliest computer music books published in 1969: “the widely varied technical and mathematical background of this audience makes it hard to select a technical level for this presentation” (204).

Moreover, unlike the languages specifically targeting novice programmers like first year computer science students such as BASIC¹, for which it may be justifiable to limit the available features of the language so that the students do not have to face the advanced programming concepts from the beginning, computer music programming languages are also expected to provide such advance features, as more sophisticated compositional and sound synthesis algorithms can be implemented within the language as desired²

However, even though a general assumption on what kind of users are going to use the language can not be made, computer music languages historically have expected users to posses some expert knowledge in the domain of computer music, or at least be

¹BASIC stands for Beginner’s All purpose Symbolic Instruction Code (121).

²For instance, LC provides features such as light-weight concurrency, lexical closure, first-class functions, etc.

highly motivated to acquire such knowledge. Computer music languages are designed as domain-specific languages tailored for computer music and do not particularly expect users to be experts in programming, while they may or may not possess such expertise in programming; such users can be categorized as ‘*expert end-user programmers*’. Blackwell described expert end user programmers as follows: “end-user programmers should not be regarded as “deficient” computer programmers, but recognised as experts in their own right and in their own domain of work. They might only write programs occasionally or casually, but it is possible that they have done so for many years, possibly distributing their work for use by many others” (43).

Thus, it would be justifiable to claim that the expected users of LC are expert end-user programmers, while this thesis does not make any assumption on the skill set of users, as it focuses mainly on the technical issues in computer music language design, rather than the HCI issues in the context of end-user programming.

9.2.2 The conceptual gap between the unit-generator concept and microsound synthesis techniques

In the previous sections, we describe the difficulty in microsound synthesis programming in the context of software anti-patterns. However, the issue is an interesting topic also in the context of the usability problems in programming language design. While this thesis focuses on the former context of software anti-pattern and sound synthesis framework design, several issues with respect to usability are briefly discussed in this section.

As discussed in Section, 2.3, a significant conceptual gap in how a sound object is modelled between the unit-generator concept and microsound synthesis techniques can be observed. In fact, Gabor himself, whose idea originates microsound synthesis, contrasted his theory to “the orthodox method of analysis”, which “starts with the assumption that the signal is a function $s(t)$ of time t ” in (119); this ‘orthodox method’ that Gabor discussed may have a strong association with the unit-generator concept, which is modelled after the sound synthesis by the analogue electronic equipment, as Mathews, the inventor of the unit-generator concept, clearly mentioned in his publication (204, p.36).

9. APPENDIX IV: ADDITIONAL DISCUSSION

Such a conceptual gap can be also seen in derived sound synthesis techniques. In the chapter of “A Survey of Classic Synthesis Techniques in Csound” of *the Csound Book* (52, Chapter 11), Fischman classifies classic synthesis techniques into two categories: frequency-domain techniques and time-domain techniques. He describes that frequency-domain techniques are “based on the assumption that any signal can be considered to be the sum of sines and cosines – each with its own amplitude, frequency and phase – according to theoretical principles developed by Fourier (1768-1830)” while time-domain techniques are “based on the construction of signals from the combination of short sounds” (52, p.223). Figure 9.1 describes the classification of sound synthesis techniques by Fischman.

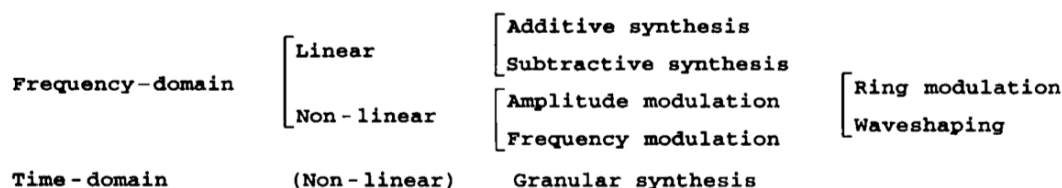


Figure 9.1: Classic synthesis techniques classified according to their principles of realization - by Fischman (52, p.224)

This gap in the concepts of sound synthesis techniques can be clearly reflected by the explanations of sound synthesis techniques in many computer music textbooks. Figure 9.2 enumerates some of the examples of the descriptions of frequency-domain synthesis techniques found in the Csound book.

On the other hand, *Microsound*, the book entirely dedicated to microsound synthesis techniques written by Roads (242), contains a lot of the expressions that describe microsound synthesis techniques as the manipulations of microsound entities. For instance, the expressions as in Figure 9.3 can be found.

While more examples can be found in these books and also in the other documents, even the above short descriptions taken from the major books in computer music clearly contrast the traditional sound synthesis techniques, such as additive synthesis and subtractive synthesis, to microsound synthesis techniques; while many traditional sound

- “Sinewaves of various frequencies and amplitudes are added together (mixed) in order to produce complex sounds” (52, p.224) (on additive synthesis).
- “Subtractive synthesis uses complex spectra as inputs that are shaped by enhancing or attenuating the component sinewaves” (52, p.231) (on subtractive synthesis).
- “This nonlinear techniques consists of the use of a signal, the modulator, to modify the amplitude of another signal, the carrier. Each samples of the modulator multiplies a corresponding sample of the carrier, distorting the latter and creating new spectral components” (52, p.237) (on ring modulation).
- “Another way of producing distortion consists of the creation of a dependency between the amplification applied to a sample and its actual value” (52, p.243)(on waveshaping).
- “The use of a modulator in order to modify the frequency of a carrier, may be controlled to produce varied dynamic spectra with relatively little computation overheads” (52, p.249) (on frequency modulation).

Figure 9.2: The examples of the descriptions on frequency-domain synthesis techniques in Csound book (52).

- “A single grain serves as a building block for sound objects. By combining thousands of grains over time, we can create animated sonic atmospheres.” (242, p.87)
- “In synchronous granular synthesis (SGS) sounds results from one or more streams of grains. Within each stream, one grain follows another, with a delay period between the grains. Synchronous means that the grains follow each other at regular intervals.” (242, p.93)
- “Formant wave-function synthesis (fonction d’onde formantique or FOF) generates a stream of grains, each separated by a quantum of time, corresponding to the period of the fundamental frequency. So a single note produced by this technique contains hundreds of FOF grains.” (242, p.164).
- “To double the duration of a sampled signal, the algorithm segments it into grains, cloning each so that two grains appear for everyone in the original. To halve the duration, it deletes every other grain.” (242, p.197).
- “Waveset transposition substitutes N copies of a waveset in the place of M wavesets, for example 2 in the space of 1, or 1 in the space of 4 for doubling and quartering of frequency respectively.” (242, p.207).

Figure 9.3: The examples of the descriptions on microsound synthesis techniques in Microsound (242).

9. APPENDIX IV: ADDITIONAL DISCUSSION

synthesis techniques are conceptualized as *function $s(t)$ of time t* (as Gabor discussed as the orthodox method), microsound synthesis techniques are described as the manipulations (or algorithms) that are applied to short sound particles, which are rather algorithmic.

This suggests the computer music language design that was developed only for the traditional sound synthesis techniques without taking microsound synthesis into account may not be very appropriate for describing microsound synthesis techniques, as the users' conceptualizations of these sound synthesis techniques seem to significantly differ between these two categories of sound synthesis techniques.

Generally speaking, a software design that is incompatible with the users' conceptualization can cause a significant problem in the usability of software. As Blackwell discusses that “even where developers are well motivated and sympathetic to user concerns, incompatible abstractions are a constant challenge to user centered design” (42).

According to the previous studies in the field of psychology of programming (143), such expressions in the documentations are considered as one of the data sources to investigate the conceptualization of the tasks by the users (44), together with the verbal data obtained by the other methods such as a think-aloud protocol (183), Contextual Inquiry interviews (40), Critical Decision Method interviews (144), etc.

A series of the works by Blandford and her colleagues on the CASSM (Concept-based Analysis of Surface and Structural Misfits) framework¹ (44)(45)(46)(78) is suggestive even for such a language design issue. The CASSM framework focuses on “the identification of misfits between the way the user thinks and the representation implemented within the systems” (44).

Since programming activity is considered to “use knowledge from at least two domains, the application (or problem) domain and the computing domain, between which they establish a mapping” (97, p.22), such *cognitive misfits* can lead to a significant obstacle in programming activity.

However, we do not further discuss this difficulty in the context of the usability problem in human-computer interaction, as this thesis focuses on the aspect of software anti-pattern in the context of software design/engineering; Yet, such a perspective is

¹CASSM was previously known as Ontological Sketch Modelling (79).

very suggestive in that the gap in the concepts between the users' conceptualization and the software design can lead to an obstacle in programming activity.

9.2.3 User interface design

The previous two sections briefly describes the issues that may be relevant to user-centred design. The clarification of expected users is normally a first phase in user centred design and it is also considered to be in the context of HCI, especially of the psychology of programming (143) to discuss how the conceptual gaps in the domain-specific expert knowledge possessed by users and the programming language design can be removed for better usability of a domain-specific programming language.

User interface design is another undiscussed topic, which is also not within the scope of this Ph.D thesis, yet likely interesting to investigate for future work. For instance, a unit-generator graph may be more comprehensible if represented graphically in a visual computer music language than written as program text in a textual language. The research on the comprehensibility in visual and textual programs already exists (132)(131) and researchers and engineers also developed the hybrid (textual + visual) programming languages; for instance, the recent version of Max integrates textual languages such as JavaScript (127) and Lua (153)(150) and previous research also exists to integrate visual programming elements into textual languages (108) and to translates solution made by a user in a visual programming environment into a textual program text (112). *Field* by OpenEndedGroup, which consists of three artists (Marc Downie, Shelley Eshkar, and Paul Kaiser) is one of the notable recent examples. In the programming environment in *Field*, one can implement graphical user interfaces (e.g., sliders, buttons, comboboxes, etc.) directly within a program text (104).

The issue of user interface design in the context of live-coding may be also desirable to investigate, as a programmer in such an unusual programming context may require further assistance to reduce cognitive loads during programming; live-coders must write and modify a program text while they must also pay a good attention to the current sonic presentation on stage. For example, the visualization of the runtime status of a program that is being executed may benefit so that a live-coder can easily grasp what is going on, as it can be sometimes hard to distinguish the sonic outputs from different threads if multi-threading is involved in live-coding. A programmer may want

9. APPENDIX IV: ADDITIONAL DISCUSSION

to terminate a certain thread just by clicking a corresponding icon in the programming environment.

The visualization of programs being executed may be also beneficial to the audience. While program texts are often projected to a screen in a live-coding performance so that the audience can see the process of live-coding, it would be less helpful for the people who cannot comprehend the meaning of the program texts. The visualization of programs being executed may be beneficial for the audience to understand what is going on during the performance, but also for the communications between live-coders on stage.

This issue can be also considered as a topic of software visualization (99)(280) and may benefit the field, especially in the visualization of a program structure and behaviour at runtime.

As above, while the topic of user interface design would be also of significant interest in the context of creative-coding (133)(224) as seen in computer music or media-art, it may also benefit user interface design of programming environments in general.

9.3 Other miscellaneous issues

9.3.1 Popularization

The popularization of LC is clearly out of the scope of this study. However, it may be beneficial to discuss such an issue, as it may suggest future work for further investigation. Generally speaking, why and how a programming language acquires a significant number of users is largely unforeseeable. The popularization of a programming language seems to depend on many different factors and each programming language seems to have its own reasons for why it became one of the widely-used programming languages. Moreover, there are even those programming languages, which used to be very popular, but now have less programmers.

For instance, when the recent rapid popularization of JavaScript seems largely due to significant demands for client-side dynamic web page development techniques/technologies, such as Ajax (146) or HTML5 (225), which were hardly foreseen at the time when the language was developed. Erlang was first developed and used only inside Ericsson, yet since its language design provides the significant benefits when developping

concurrent/fault-tolerant/real-time/distributed systems, Erlang is rapidly gaining considerable remarks after its public release (21). Haskell (first developed in 1990) is another example of a programming language that rapidly gained attention among the community (295).

While none of functional programming languages had ever gained significant popularity in the software industry before, the recent trends of multicore programming and distributed systems, which are largely due to the expectation that the processing speed of CPUs may be harder than before because the physical limit (the end of Moore's law) (164), leads to attracting more recognitions to the benefits of functional programming languages; their characteristics, such as immutable data structures and lazy evaluations, are considered suitable for parallelisation (137) and preferable to make better use of multi-core CPUs and distributed systems (21, Chapter 19: Multicore Prelude).

On the other hand, there exist some programming languages that have lost their popularity. For instance, FORTH programming language (59) was once popular in the 1980s especially for micro computers of the time because of its portability and small memory foot print, yet today it is not a major language. Object Pascal (236) was also once popular for rapid application development on the Microsoft Windows platform, but it seems the population of Object Pascal programmers rapidly shrank in the last several decades; thus, it is very hard to foresee the popularization of a programming language.

Furthermore, the popularization of a programming language may largely depend on available support from the developers and the community. This may be especially true when considering computer music languages. For instance, the popularity of Max seems largely due to the fact that it is a commercial product with support from both the developer and the large user community. SuperCollider, another widely-used computer music language, was first developed as a commercial product by James McCartney, until the developer decided to make the source code open under GPL license. Before the open source release, SuperCollider users have been well supported by the developer and the user community seemed to have grown large enough when it became an open-source product. PureData has been developed as a open source project from its beginning, yet its similarity to Max may contribute to the rapid growth of the user community; it seems the popularization of a computer music language may require such dedication

9. APPENDIX IV: ADDITIONAL DISCUSSION

from the professional developers both in the development and the support. Such an issue may be also interesting to investigate for further discussion.

Some computer music languages also survive as plug-in languages in different programming environments. For instance, many RTCmix modules are available in Max. Nyquist is now gaining more users than before as a plug-in language in Audacity sound editor (263). Such a direction may be also noted as a different kind of popularization of computer music languages.

Also, it should be noted that even if a language fails to be popular, the concept and design that the language proposes often survives, influencing the design of new languages. A good example is LISP (209), which is one of the oldest programming languages first developed in the late 1950s. Such concepts as ‘lambda’ and ‘eval’ in LISP survive in many recent programming languages today.

Additionally, the rapid growth of the community may not be always good for a programming language, as the change of the language specification can be difficult. For instance, while Python (301) is one of the most successful recent programming languages, a considerable number of Python (301) programmers are still using Python 2 even long after the official release of Python 3, because Python 3 is not completely backward compatible due to the change of the language specification. On the other hand, HASKELL designers has an unofficial motto: “avoid success at all costs” (158), which epitomises “a culture of agility, where new research results are integrated into the language and library suite while the user base nimbly adapts” (83); thus, having the rapid growth of the programmer community is sometimes considered not favourable when considering the further development of the language.

As this thesis and project are more about contributing to the research on computer music language design, the contribution to the academic knowledge together with the design exemplar is considered more important than the popularization of the language itself. Yet, it would be desirable to discuss the issues as above for further development of LC.

9.3.2 Musical practices that LC may be suitable for and may not be suitable for

As described in Chapter 3, LC is a highly dynamic programming language suitable for an interactive programming environment. LC’s features with respect to time such as mostly-strongly-timed programming, start-time constraint, light weight concurrency, and time-tagged inter-thread message communication can also reduce the effort required for scheduling musical events in a user program. Mostly-strongly-timed programming would be helpful to describe some algorithms as a task, which may be suitable for transformational systems, “whose inputs are available at the beginning of the execution and which deliver their outputs when termination” (135), within a task in an interactive system. By giving the task a execution-time constraint, the feature of time-fault tolerance (timeout) in LC makes it possible to describe how to handle the violation of the execution-time constraint in a user code. In addition, as LC’s microsound synthesis framework and prototype-based programming at the sound synthesis level may be beneficial for further investigation algorithmic compositions at the audio level, even in live-coding. Thus, LC is designed so that it can be suitable for interactive live computer music systems with such application domains.

However, as the current version of LC was implemented without much consideration on the performance efficiency, as it is a proof-of-concept prototype for the language design proposed in this thesis, it is not very good for some musical practices that may require high-performance. Yet, the performance efficiency of a dynamic programming language is a traditional topic and many techniques has been developed, as surveyed in (167), and Just-in-Time compilation (25)(155) is also investigated in the context of computer music programming language (218)(273); there exists a significant body of previous research that is beneficial to improving the performance efficiency of LC in the future.

On the other hand, as LC highly focuses on live interactive computer music applications, the language may not be beneficial for non real-time usages. For instance, Chronic (56), an internal domain-specific built on OCaml (182) for non real-time sound synthesis developed by Brandt and Chronic’s programming model has a problem in causality, which is a significant obstacle to apply the programming model for interac-

9. APPENDIX IV: ADDITIONAL DISCUSSION

tive, real-time computer music systems, as Brandt himself admits in(56, p.77); Chronic is acausal in that the future events or output can affect the past events or output.

However, this lack of causality in Chronic can be beneficial for some musical practices that do not require real-time and/or interactive presentation. While the majority of the recent computer music programming languages are designed for real-time interactive computer music applications, how a non real-time computer music language can be better designed and how it can support creative music practices are issues that are worthy for further investigation but still neglected.

While LC highly focus on real-time, interactive computer music, it may be an interesting topic to investigate how strongly-timed programming can be adopted to non real-time usages, for instance by allowing *now* to be shifted to the past. Cmix's MINC scripting language is suggestive in this sense, as it is a non real-time computer music language with the ability to schedule sound objects by user algorithms, written in the style of imperative programming; however, such a topic is also beyond the scope of this thesis.