Bachelor's Thesis

nova - A New Computer Music System with a Dataflow Syntax

carried out at the

Design and Assessment of Technology Institute
HCI Group
Vienna University of Technology

under the guidance of
Univ.Ass. Dipl.-Ing. Dr.techn. Martin Pichlmair

by

Tim Blechmann
tim@klingt.org
Matr.Nr. 0526789

Vienna, April 11th, 2008 _____

Abstract

nova is a new computer music system based on a dataflow syntax, which shares a common subset with Max-like languages like Pure Data, Max/MSP or jMax. Nevertheless, nova is not just a new implementation of the Max language, but has been redesigned from scratch in order to overcome some of its limitations. nova is written in the C++ programming language. The implementation is based on a highly modular object-oriented design, with the ambition to create a low-latency soft real-time system in order to avoid interruptions of the audio stream, a strong focus on parallelization on multi-processor computers, and easy portability to different platforms. The audio engine is highly optimized for performance on modern computers, mainly the x86 and x86_64 architecture.

This bachelor's thesis is structured as follows: The first Section gives an overview of different computer music systems, with a focus on the history of Max-like languages. The second Section describes the main aspects in the design of the nova language, followed by the third Section, which gives an overview of the software design of the implementation. This Section is followed by a conclusion and outlook to future developments.

Contents

Table 1: Max-like Computer Music Systems

	Started	License	Platform	Features
Patcher	1986	Proprietary	Macintosh	Messaging, MIDI
Max/FTS	1989	Proprietary	NeXT/ISPW	Messaging, MIDI, Audio DSP
Max/Opcode	1990	Proprietary		Messaging, MIDI
Pure Data	1996	BSD-Style	Crossplatform	Messaging, MIDI, Audio DSP
Max/MSP	1997	Proprietary	Macintosh, Win32	Messaging, MIDI, Audio DSP
jMax	1996	LGPL	Crossplatform	Messaging, MIDI, Audio DSP

1 Introduction

1.1 Motivation

When I started to work with computers as musical instruments, I became in involved with computer music systems. However, working with several systems and knowing that I'm going to work with the computer as my main instrument in the future, I was looking for a reliable tool to use for my own musical purposes. As no system that I found was somehow acceptable, I started to work on nova in summer 2005 with the ambition to take the strong concepts of existing systems, while being able to rethink the weak parts.
Since late 2006, I have been using it during live performances.
Nova is a real-time computer music system with a dataflow syntax. It has many similarities to max-like languages like Pd, Max/MSP or jMax, but introduces some new concepts to the max-language.

1.2 Max-like Language

Max-like programming languages [Puc02] have been used as computer music system since the late 1980s, transferring the concept connecting single units from hardware synthesizer to software. The term 'Max-like' is used to describe the common syntax of the systems, listed in Table 1.

1.2.1 Patcher

Patcher was the first incarnation of max-like languages. It was developed initially developed in 1986 for the realization of Philippe Manoury's composition 'Pluton' for solo piano & live electronics [Puc02]. It was able to process MIDI (Musical Instrument Digital Interface) events and

control the audio synthesis on a Sogitec 4X machine, a multi-processor workstation for digital signal processing developed at IRCAM (Institut de Recherche et Coordination Acoustique/Musique) [vS].

1.2.2 Max/FTS & ISPW

The IRCAM Signal Processing Workstation (ISPW) was a designed as the replacement for the 4X. The development was a collaboration between the IRCAM and Ariel Corporation, starting in 1989. The ISPW consisted of a NeXT computer with a extension board of 2 to 24 Intel i860 coprocessors for signal processing and thus was one of the first computers using general-purpose processors for real-time signal processing [Puc91b].

The software of the ISPW was consisting of two parts, a NeXTSTEP based user interface and control engine called Max, and a real-time signal processing engine called FTS (Faster Than Sound), running on the i860 extension boards. The FTS server could be controlled via Max's 'tilde' objects [Puc91a] and Eric Lindemann's ANIMAL system [Lin90].

1.2.3 Pure Data

In 1996, Miller Puckette started to write Pure Data (Pd) in order to be able to do a redesign of the Max program. Pd was rewritten from scratch with a focus on the handling of dynamic data structures, influenced by some concepts from the ANIMAL system [Puc96]. Pure Data was designed to use two processes, a server process written in the C programming language for the message interpreter and signal processing, and a gui process written in Tcl using the 'Tk' toolkit, which are communicating via network sockets. In contrary to Max/FTS it is designed to work on uniprocessing computers [Puc96]. Pure Data is free software, released under the 3-clause BSD License. Pure Data still has a wide user base.

1.2.4 Max/Opcode

Patcher was licensed to Opcode Systems, where it was adapted and extended by David Zicarelli, who was working at the IRCAM with Miller Puckette. In 1990, Opcode Systems released Max/Opcode as a commercial software, but was discontinued after a few years [Dé].

1.2.5 Max/MSP

In 1997 Cycling '74, a company founded by David Zicarelli, release the commercial software Max/MSP. Max/MSP is based on the original code-

Table 2: Scripting-based Computer Music Systems

	Started	License	Language
Music N	1957		Assembler, Fortran
CSound	1986	LGPL	C
Common Lisp Music	late 1980s	MIT-Style	Common Lisp (see Section 1.3.3)
SuperCollider	1996	GPL	C++, SuperCollider
ChucK	2003	GPL	C++

base from Max/Opcode, with the additional audio processing package MSP (Max Signal Processing), which is based on the audio engine of Pure Data [Puc97]. In 1999 nato.0+55+3d was released as add-on package for real-time video processing, although development stopped in 2001. In 2003, Cycling '74 released the Jitter add-on for real-time video, matrix and 3d graphics [cyca]. Max/MSP became widely used among computer musicians.

1.2.6 jMax

The "Real time systems" group of François Déchelle at IRCAM started to develop a successor of Max/FTS. The architecture of Max/FTS was re-engineered in order to split the graphical user interface from the signal processing engine [DBdC+98b], [Dé]. jMax reused the Max/FTS engine, while the graphical user interface was written from scratch using the Java programming language to ease the cross-platform portability [DBdC+98a]. jMax is free software released under the GNU Lesser General Public License and it supports the platforms Windows, Mac OS X and Linux. However it never got used as widely as Pure Data or Max/MSP and it's development has been discontinued in 2004.

1.3 Computer Music Systems based on a Scripting Language

Another widely used approach for computer music systems is the use of a text-based language. Table 2 gives a short of the discussed systems.

1.3.1 Music N

The "Music N" computer music languages written by Max Mathews were the first computer programs to create music and synthesize sounds on a digital computer. The series started with "Music I" developed at Bell Labs in 1957. The final and most influential of Mathews Music N lan-

guages was “Music V”, which was implemented in the late 1960s using the Fortran programming language instead of assembler and therefor was more portable to other computers [Mat].

The main concept behind the Music N family of languages is the separation of ‘orchestra definition’ and ‘score definition’. The ‘orchestra definition’ defines a number of instruments by their the signal and control flow. It consists of different kind of Unit Generators (ugens), oscillators, envelope generators, filters and function generators [Pop04]. The ‘score definition’ consists of some definitions for audio input and output settings, function tables and a note list, where each note is defined by the instrument identifier and a number of control parameters.

To execute a Music N program, the ‘orchestra definition’ is compiled and the ‘score definition’ is used to drive the program’s scheduler. The result is stored in a soundfile that then can be played back in real-time. It can therefor be seen as a soundfile compiler [Pop04].

Although Music V is not used any more, it influenced many computer music languages that have been written in the 1980s like “Music 360”, “Csound”, “CMix” or “SAOL”.

1.3.2 Csound

These days, the only Music V based language, that is still widely used is Csound. Csound was written at MIT by Barry L. Vercoe based on his earlier system “Music 360”, it was released in 1986. As the name suggests, Csound is implemented in the C programming language and therefor runs on every system, providing a C compiler [Ver07]. Csound follows the Music V paradigm of splitting ‘score’ and ‘orchestra’ to different files in order to compile a sound file. Since 1990 Csound had the capabilities to compute audio in real-time [Bou00]. In addition to writing score files manually, it became a common practice to generate score files algorithmically using external tools.

Csound has been widely used in the academic computer music community. It is still actively developed, mainly by John ffitch at the University of Bath. The latest version Csound 5 can be used stand-alone, with front-ends like “blue” or “cecilia”, as LADSPA or VST plug-in [Ver07, LW07] and embedded in Max/MSP or Pure Data. It is free software released under the GNU Lesser General Public License.

1.3.3 Common Lisp Music

Common Lisp Music (CLM) is a Music-V based synthesis language written by Bill Schottstaedt at the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University. CLM has originally been implemented in Common Lisp, but was ported to C, Scheme, Ruby and Forth [Scha]. CLM can be used from a Common Lisp environment, as well as from the SND sound editor [Schb] or from Common Music, a Common Lisp based music composition environment [Ama05].

1.3.4 SuperCollider

SuperCollider is a domain-specific scripting language with a real-time garbage collector. It was written by James McCartney and released in 1996. In 2002, it was released as free software under the GNU General Public License.

In the contrary to the older Music N based systems, SuperCollider is based on an object-oriented scripting language with a syntax based on Smalltalk and C++. The language is used to define instruments (called 'synthdefs') and control the audio synthesis [Pop04]. The interpreter for the SuperCollider language and the SuperCollider server, that does the audio processing, are separated processes communicating via Open Sound Control (OSC) [WF97]. The server can be controlled both via classes written in the SuperCollider language and directly by sending OSC control messages.

By decoupling synthesis engine and language, it was possible to generate control events in the background in advance. Several instances of the SuperCollider server can run on the same machine, to make use of multiple processors. [McC02]

SuperCollider can be used for both real-time and non-real-time audio synthesis. The language gives the user a fine-grained control over the signal graph. The server is designed to support dynamically changes to the synthesis graph, allocation or deallocation of synthesis modules or audio buffers [Ama05].

SuperCollider is one of the most widely used computer music systems. It runs on Mac OS X and Linux, with a port to Windows in an early state of development.

Table 3: Comparison of Computer Music Frameworks

	Started	License	Type	Application Domain
CLAM	2000	GPL	C++ Framework	Music Information Retrieval, Spectral Modeling Synthesis
SndObj	1998	GPL	C++ Library	General Purpose, Spectral Processing
STK	1995	MIT-Style	C++ Library	Synthesis
Faust	2002	GPL	C++ Code Generator	Digital Signal Processing
CSL	2002	MIT-Style	C++ Library	General Purpose

1.3.5 ChucK

ChucK [WC04] is a relatively new computer music language released in 2003 by Ge Wang and Perry Cook at Princeton University. ChucK is free software, released under the GNU General Public License. It supports the Windows, Mac OS X and Linux platforms. ChucK is designed as an on-the-fly programming language, with support for concurrency via a non-preemptive sample-accurate scheduler. The syntax uses a massively overloaded operator, the ChucK-operator `=>`.

1.4 Computer Music Frameworks

This section covers a small number of computer music frameworks, which are not intended to be used as stand-alone computer music systems, but rather as building blocks. Table 3 lists the discussed frameworks. A comparison of their implementation concepts can be found in Table 4.

1.4.1 CLAM

CLAM (C++ Library for Audio and Music) [Ama05] is a high-level C++ framework offering both C++ implementations of algorithms as well as conceptual models. It was developed at the Music Technology Group of the Pompeu Fabra University in Barcelona by Xavier Amatriain starting

Table 4: Comparison of Computer Music Frameworks (Features)

	Patching Semantics	Sample Representation	Audio-Rate Controls
CLAM	Graphical Patcher, not in API	Single-Precision	No
SndObj	Static	Single-Precision	Object-Specific
STK	No	Typedef (Double-Precision)	Yes
Faust	Compiled	Single-Precision	Yes
CSL	Dynamic	Typedef (Single-Precision)	No

in 2000 and released under the GNU General Public License. The framework has a focus on the use in music information retrieval and spectral modeling synthesis.

1.4.2 SndOBJ

SndObj [Laz01] is an object-oriented C++ class library for audio synthesis and processing written by Victor Lazzarini and licensed under the GNU General Public License. SndObj classes provide a strong encapsulation and modularity, while being portable to different platforms [Ama05]. Among the more than 100 classes, some algorithms for spectral audio processing are included. SndObj can be used to easily implement LADSPA or VST plug-ins, beside that, the SndObj classes can be used from Python.

1.4.3 STK

The Synthesis ToolKit (STK) [CS99] is a C++ class library with a focus on audio synthesis algorithms. STK was started by Perry Cook in the mid 1990s at CCRMA, Gary Scavone started working with STK in 1997. Most of the implementations are textbook implementations of the synthesis algorithms. The classes are processing the signal sample-wise, which is a computational overhead compared to block-wise processing. STK is free software, released under a permissive license, although some of the used algorithms are patented [CS].

1.4.4 Faust

Faust (Functional Audio Stream) is a functional programming language designed to define block diagrams at Grame [YO02]. Faust programs are not directly compiled to object code, but to optimized C++ code, which is automatically vectorized in order to use Altivec or SSE/SSE2 instructions [NS03]. Faust provides wrapper code to directly use the Faust programs as stand-alone application, using ALSA or Jack as audio backend with a Gtk user interface or stand-alone command line application. To use Faust applications from other programs, plug-in wrapper are provided for LADSPA, VST, Max/MSP, SuperCollider, Q and Pure Data [AG06, Gra07].

1.4.5 CSL

The Create Signal Library (CSL, pronounced “Sizzle”) was designed at the Center for Research in Electronic Art Technology (CREATE) of the

University of California Santa Barbara mainly by Stephen Travis Pope. It is an open source, general-purpose audio processing library, written in object-oriented C++. Its design is somehow similar to STK or SndOBJ as it is providing reusable C++ classes [STP03]. CSL provides implementations of algorithms for sound spacialization, including algorithms for ambisonics, binaural and vector based amplitude panning.

2 Language Concepts

The language of Nova is based on the max-like languages, mainly Pd and Max/MSP, but has been redesigned from scratch in order to overcome their limitations. As a real-time computer music system, it needs to handle to fundamentally different kinds of dataflow networks, controls and streams [Ama05]. Controls are event-driven, usually but not necessarily aperiodic and computed at a low rate. The audio signal however is a continuous stream of samples, computed at a much higher data rate. In the dataflow model of max-like language like Nova, both audio and control messages are expressed by the same dataflow graph.

2.1 Nova Scope

The Nova language is using an interpreter to run Nova programs, the Nova interpreter. Currently, there are two different versions of the interpreter, a simple command-line program and a gui version with a graphical patch editor. A Nova program is started, by loading a patch into the interpreter. Nova patches are state definitions, modelling a certain interpreter state. The state of the whole interpreter, i.e. the combination of all patches that are loaded into the same interpreter at the same time are called the **Nova Scope**.

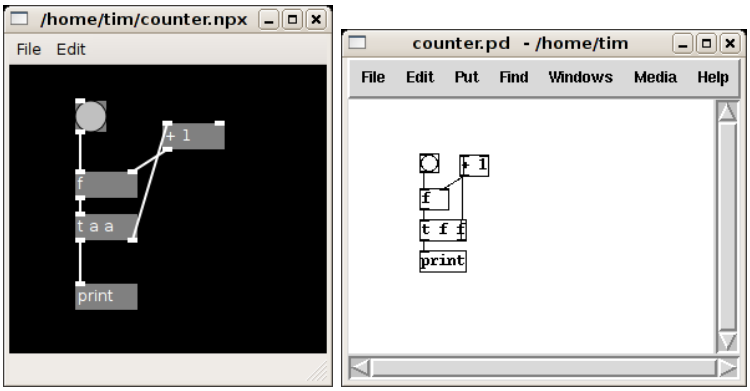
2.1.1 Language Elements

In max-like languages **canvases** are used as a container for objects. Objects can be primitives (written in C++) or written in the Nova language itself. Objects that are written in Nova are as well based on canvases. They are called **subpatches**, when implemented in the same file, or **abstractions**, when stored in a separate file. Subpatches and instantiated abstractions can be seen as nested canvases on their parent canvas. This hierarchy of canvases is called the **patch hierarchy**.

Primitive objects are either built into the Nova language or third-party plugin libraries, that can be loaded into the interpreter at runtime.

Objects have a number of **inlets** and **outlets**, which can be used to connect different objects. Outlets (on the bottom of an object) can be connected to inlets (on the top of an object). Figure 1 shows a simple counter in Nova and Pd. When canvases are used as objects, the xlets[1] are defined as objects, as shown in Figure 2, where the counter is imple-

[1] The term 'xlet' is used to denote the combination of inlets and outlets



(a) Nova (b) Pd

Figure 1: A simple counter

mented as subpatch. These objects are `inlet` and `outlet` for messages, and `inlet~` and `outlet~` for signals.

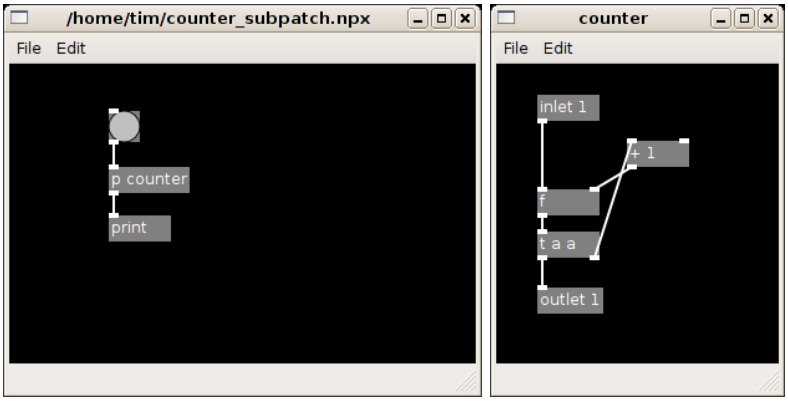
2.1.2 Class Resolution & Namespaces

Objects are instantiated by creating an object box, with the class name followed by a list of creation arguments. The resolution of the class name needs to handle the different kinds of class types (internal & external objects, abstraction) in a predictable way. Nova provides a hierarchical class lockup using namespaces, which are denoted by the delimiter '.' (e.g. `list.iter` denotes the object 'iter' in the namespace 'list').

The class name resolution is done in the scope of the patch, where the object is created. There are three kinds of class hierarchies that are used for the class-name resolution:

- the global class hierarchy, which contains all registered objects
- the global search paths (property-settings of the nova interpreter)
- the local search paths of the parent patch, which defaults to the folder containing the parent patch.

While classes in the global class tree hierarchy are are statically registered into their namespaces, the path-based class lookup takes the filesystem hierarchy into account. In the path-based resolution, an object with the name `myfancysynth.voice~` may resolve to (1) an abstraction 'voice~' in the folder 'myfancysynth', to (2) an external object 'voice~' in the folder 'myfancysynth', or (3) to an object 'voice~', that is part of the external



(a) root patch (b) subpatch

Figure 2: A simple counter as subpatch

library 'myfancysynth', which is located in the search path. When a class name is looked up, it can happen, that it resolves to more than one class. In the case of such a name clash, the instantiation of a class fails to avoid an undefined behavior of a patch.

2.1.3 Audio & Control Connections

There is a distinction between control and signal xlets, as controls and signals define two independent dataflow networks. Outlets can provide either signal or messages, but not both, while certain inlets are able to handle both types. This way, certain audio objects like oscillators can be controlled by messages or signals. The type of a connection is therefore dependent on the type of the source outlet.

2.1.4 Bindable Objects

Nova patches can contain resources, that are shared between different objects. These shared resources correspond to named variables in other programming languages. In Nova these objects are called **Bindable Objects**, as objects can be bound to these resources. Examples would be message or signal busses, audio buffers, that are used by sampling objects, or shared values.

Unlike other max-like languages, where bindable objects are resolved in the global scope[2], Nova provides a way to have fine-grained control of

[2]Local resources need to be simulated by using globally visible unique symbol

resource visibility. The resource visibility is dependent on the position of the resource in the patch hierarchy (see Chapter 2.1.1). When a `declare` object is located on a patch, a resource of a specific type is declared on this point in the patch hierarchy, which is visible on this canvas and all child canvases. However, this resource can be shadowed by a `declare` object of the same name and type somewhere lower in the patch hierarchy. Objects that try to bind to a specific resource, search for this resource upwards in the hierarchy. If no resource is found in the hierarchy, a globally visible resource is used. In order to make sure, that the global resource is not shadowed by another resource somewhere upwards in the hierarchy, it is possible to redirect the visibility to the global resource using `declare global`.

Certain resources can be allocated implicitly (e.g. message or signal busses) while others are required to be declared explicitly (e.g. audio buffers, which need to allocate a certain amount of memory). If no matching resource of an implicit bindable can be found during a binding request, a globally visible one is allocated and bound. If the same would happen when binding an explicit bindable, the binding request would fail.

2.1.5 Patch Lifetime & Encapsulation

In max-like dataflow programming, objects are rough equivalents to classes in object-oriented programming. Although some terms of object-oriented programming like inheritance or member functions can't really be applied to dataflow programming, the concept of constructors and destructors, that is transparent to the nova language, is very important in order to be able to maintain large nova patches. The equivalent to constructors and destructors of patches are `loadbang` and `endbang` objects. When a patch is loaded into the nova interpreter, all loadbang operations will be executed, the loadbangs on child canvases will be executed before the ones on the patch. Endbangs are executed before a canvas is unloaded from the interpreter.

Before the loadbang of a canvas has been executed, no information is allowed to be passed to the canvas or its child canvases. Instead of that, messages, that are sent to the canvas will have to be queued until the loadbangs of this canvas have been executed. The behavior of endbangs is similar. Messages that are passed to a canvas after the time of the endbang will be silently ignored.

2.2 Message Handling

Some computer music systems like SuperCollider or Csound have the notion of a signal-rate, that is used to compute the audio signal, and a lower control-rate, to control the unit generators at a lower rate in order to save processing power. Max-like languages use a different concept, which is related to the 'messages'. Messages are events containing data, that are triggered asynchronously but are dispatched synchronously from the audio scheduler. A control-rate can be simulated by scheduling messages at a certain rate, though.

2.2.1 Messages as Events

There are different event sources that can trigger events. They can be divided into user-generated events, like gui-events or events triggered by an OSC message received from the network, and system-events like timer events. When an event occurs, it will be passed in a depth-first order to the objects. In order to force a specific order of execution, one can use the `trigger` (abbrev: `t`) object. Figure 3 shows an example patch with two ways to print the numbers 1-2-3 to the console.

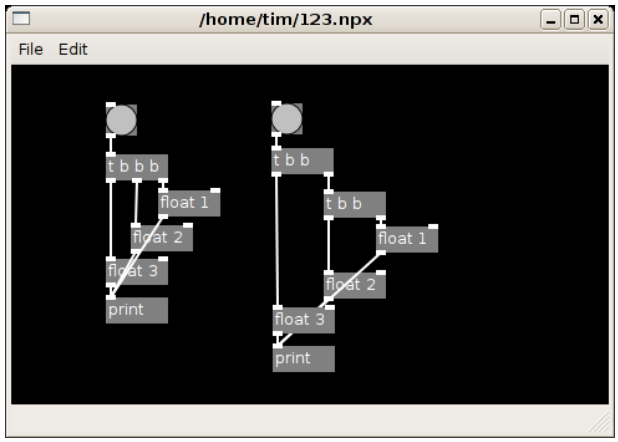


Figure 3: Message-passing example

All messages, that are triggered by the same event, will be executed at the same logical time and are synchronous to the audio computation. This logical time is directly connected to the audio computation and thus usually driven by the audio hardware. If the handling of the messaging takes too long, audio dropouts may occur. For more details see Chapter 3.1.

Messages that are passed through many objects are handled like a function call stack in other programming languages. It is easily possible to create a stack overflow. Figure 4 shows an example patch with two implementations to print the numbers from 0 to 999 to the console, one implementation using iteration, the other one using recursion.

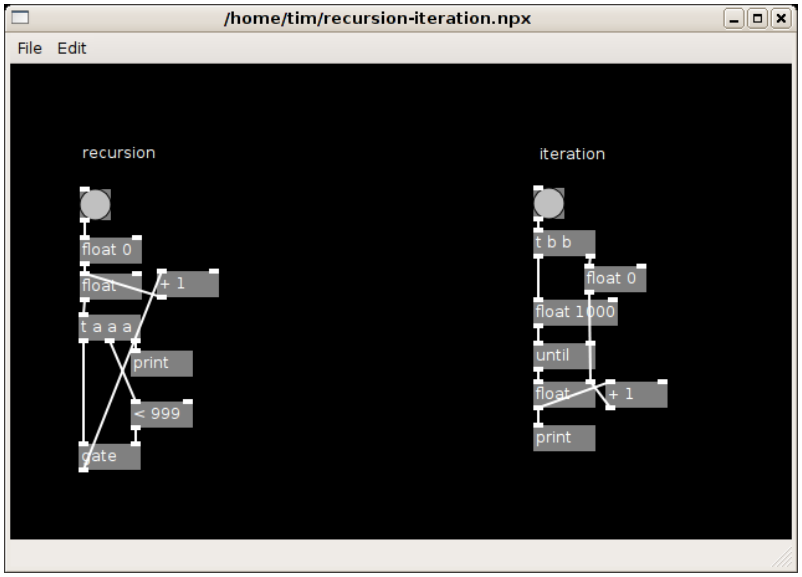


Figure 4: Recursion vs. Iteration

2.2.2 Messages as Data

Messages are not only triggered events, but also pass data. The supported data types are simple built-in types, but it is also possible to extend the type system with complex classes, which can be defined from the C++ interface. One instance of any of these types can be saved in an **Atom**.

Bang Event without data passing semantics, storage type is 'none'[3]

Float Double-precision floating-point number

Symbol Hashed string, intended to be used as message selector

String String data type, usable for efficient string processing

List List of atoms

Pointer Reference to a class instance of the extendable type system

[3]similar to None in Python

2.3 Signal Handling

Signals networks are handled quite differently from message networks. Signals graphs are used to build a linear DSP chain which is used to compute the signal stream synchronously in blocks of usually 64 samples. In Nova, it is not allowed to use cycles in signal graphs in order to have a predictable behavior. If this were not the case, a delay of one block-size would have to be introduced implicitly and thus would result in an undefined behavior (or at least difficult to predict). Using audio busses, cyclic graphs can easily be made acyclic.

Like in other max-like languages, it is possible to run certain parts of the DSP graph with a different sample-rate (resampling) or signal vector-size (reblocking) than the rest of the graph. In Max/MSP the objects `poly~`, and `pfft~` run a patch which is given as object argument in sandbox which implements resampling/reblocking as well as overlapping for FFT applications [Cycb]. Pd provides the `block~` object, which sets block-size, up/down-sampling factor and overlapping for a canvas and its child canvases. `block~` (or it's alias `switch~`) [Puc] can also be used to suspend the DSP computation for a canvas and its children.

Nova has the notion of **DSP contexts**. A DSP context is a suspendable part of the DSP graph, which runs at a certain sample-rate factor[4], vector-size factor and overlapping relative to it's parent context. If a canvas contains one of the detaching objects `switch~` or `reblock~`, this canvas and it's children are running in a DSP context of its own.

Changes to the DSP graph are only effective after the DSP chain has been resorted, which is done in the background. This has the advantage of fewer audio dropouts during DSP graph changes, but unfortunately message and signal objects may not be in sync for the time after the loading of objects until the synchronization of the root DSP context.

[4] This is not yet implemented into the DSP graph

3 Implementation Concepts

The implementation of Nova has been done in C++ instead of C, that has been used for Max, Pd and jMax. The kernel makes heavy use of the C++ Standard Template Library (STL) and the Boost libraries [boo]. The software design is completely object-oriented and makes use of template and template metaprogramming techniques, whenever reasonable.

Nova is a designed to support lowest audio latencies, which has an influence on several aspects of the implementation. Lowest latencies in terms of audio processing means, that the vector-size of the DSP engine can be as low as 64 samples without the occurrence of audio dropouts. Audio dropouts happen, when the DSP engine can't deliver the audio data to the driver before the deadline. For professional audio hardware, the deadline would be the duration of 64 audio samples. For a sampling rate of 48000Hz, this would be after 1.33 ms. Thus low-latency audio software needs to be written as soft real-time system.

Since the schedulers of modern operating systems are not designed for dispatching threads at such a low latency and suspending the audio thread is likely to increase the response time so that the deadline would be missed, the preferred way to implement thread synchronization is using nonblocking programming techniques like atomic operations or lock-free data structures and to avoid system calls whenever possible. Chapter 3.11 gives an overview on the framework for lock-free data structures in nova.

Although the processing power of modern personal computers is continuously increasing, audio processing still adds considerable load to the CPU. Because of that the audio engine implements some concepts to save CPU cycles.

3.1 Nova Scope vs. System Scope

For the implementation of nova, the separation into two different scopes is important, the nova scope and the system scope. The nova scope deals with the handling of the nova language, as it is exposed to the user, while the system scope deals with the implementation of the interpreter, which is invisible to the user of the language.

The nova language is running completely synchronous in a real-time thread, while the interpreter is implemented asynchronously and is distributed over a number of threads. The synchronization of system scope and nova scope needs to make sure, that the asynchronous interpreter is able to interprete a synchronous language without too much annoyance

from the side of the language. To achieve this, several paradigms are used:

callback synchronization Instead of maintaining the complete interpreter state from the system threads, synchronization operations are done by injecting callbacks into the real-time thread in order to make sure that changes to the nova scope are not blocking.

lazy locking If synchronization from system scope to nova scope requires exclusive access to data structures, the operation will be rescheduled if acquiring a lock fails. In the case that an operation needs to be rescheduled, all successfully acquired locks have to be released in order to avoid deadlocks.

lock-free synchronization Whenever feasible, utilize lock-free data structures.

From the nova scope, there are two different notions of 'time'. The 'logical' and the 'physical' time. The physical time is the time, that is reported from the operating system and thus is close to the physical time. More important for the nova language is the 'logical' time. The logical time is directly bound to the audio hardware, so timer events are dispatched from the scheduler itself (for more details about the scheduler see Chapter 3.2.2).

3.2 Interpreter

Both parts of the nova language (message and signal processing) are interpreted. The interpreter (or virtual machine), is encapsulated into a single C++ class, the `environment`, which exports the whole public API of the nova interpreter as public member functions. The current implementation only supports one interpreter instance per process. The `environment` class is derived from different base classes which provide implementations for certain aspects of the interpreter like handling loadable nova classes, class instances, loading patches or managing the DSP graph, only to name the most important aspects.

3.2.1 Synchronization

The nova interpreter is designed to work highly asynchronous. The crucial point is the synchronization with the realtime thread, that shouldn't be suspended at any time. Traditional synchronization paradigms are using mutexes or semaphores, in order to guard data structures, that are used for synchronization purposes. However, the worst-case response times of

blocking algorithms may be too high which could result in a miss of the deadline to deliver the audio data to the hardware. Thus the implementation of nova tries to avoid the use of blocking algorithms for synchronization purposes. Most of the synchronization is done by inserting callbacks into the main loops of the nova threads using a lock-free queue.

Operations that are executed in the real-time thread can have two kinds of priorities, operations, that may be delayed, and operations, that need to be completed when they are called. Operations that may be delayed (e.g. system scope to nova scope synchronization) are able to use conditional thread locks to guard concurrent data structures. Operations like generation of symbols need to be completed without delay. In these cases the use of shared lock-free data structures is required in order to achieve a real-time safe behavior.

3.2.2 Scheduling

The scheduling of the nova interpreter is done by the audio backend via the audio driver, which itself is driven by the interrupt of the audio hardware. Depending on audio hardware and driver, the scheduler ticks occur at a monotonic rate. The logical time of events, that occur in the nova scope, is tied to the audio scheduler, which increments the logical time of the nova interpreter during each scheduler tick. The operations, that occur during the scheduler tick are described in Figure 5.

3.3 Nova Objects

As already mentioned in Chapter 2.1.2, the resolution of nova objects knows the notion of namespaces. The interpreter maintains a global class tree of registered classes and in addition to that provides a runtime class resolution mechanism.

3.3.1 UUID References to Class Instances

One problem of the implementation is the necessity to refer to nova objects by a unique identifier. Basically, this needs to be taken care of in order to solve two problems. The first problem is related to the asynchronous class instantiation (for more details see Chapter 3.8), which requires a way to identify a nova class instance somewhere in the scope of the interpreter. The second problem is the necessity to be able to refer to a nova object locally on an Objectholder (see Chapter 3.4) at the time of loading and

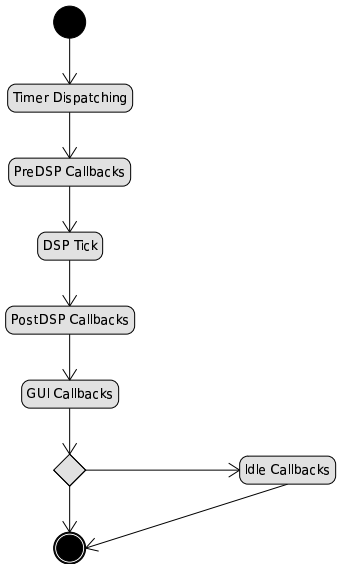


Figure 5: Scheduler Tick

saving of patches[5].

In order to solve these problems, nova class instances contain two unique identifiers, one for the global reference and one for the local. As unique identifiers UUIDs[6] are used. The global uuid can be used to lookup a nova class instance that is somewhere visible to the interpreter. If a global uuid of an object cannot be found in the interpreter, it is guarantied not to be alive. On the contrary to that, local uuids can only be used to lookup classes that are located on a certain Objectholder and that are visible from the nova scope. If an object can't be looked up by a local identifier, it is guarantied not to be visible from the nova scope, although it may be visible from the interpreter scope.

3.4 Objectholder and Object Lifetime

Canvases are a model of an **Objectholder**. An objectholder is a container for objects of the nova language, objects and subpatches, and connections. The `environment` class is a special instance of an objectholder, too, as it works as a container for the root patches, that are loaded into the interpreter. It is easy to see, that the objectholder classes of a complex system of patches models a tree structure of objectholders.

[5] and in future for proper handling of abstractions
[6] Universally unique identifier, 128-bit random numbers

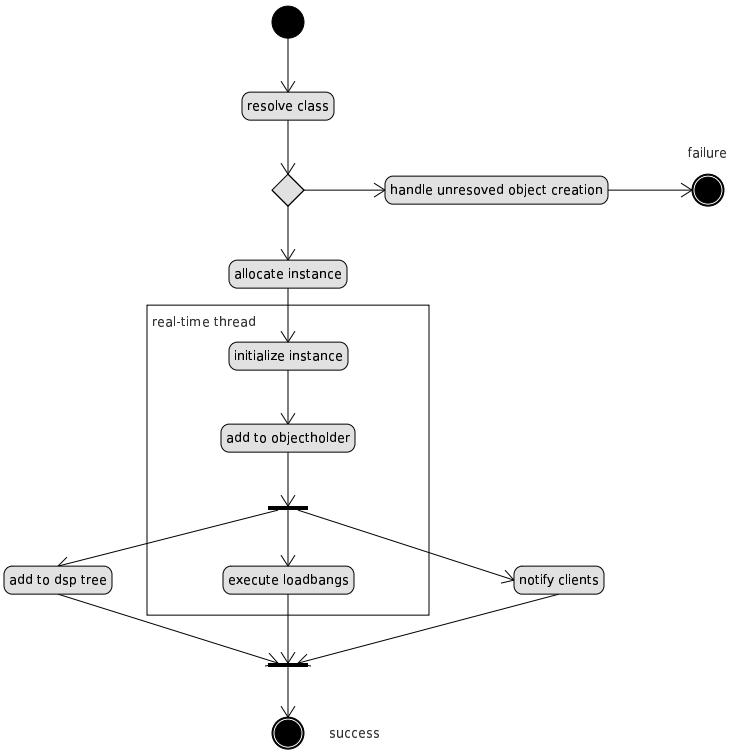


Figure 6: Object Lifetime: Object creation

In order to be able to maintain the objectholder hierarchy, the lazy locking concept is used. Changes to the objectholder are done from the real-time thread in order to avoid synchronization problems. Whenever changes to an objectholder are requested, the system will attempt to acquire a lock for it and complete the operation until it succeeds.

3.4.1 Object Memory Management

The memory management of nova objects is a considerable problem, since objects are shared between multiple threads and serialization of object lifetime is not always possible. Therefore the nova interpreter is using reference counting[7] to keep track of all visible objects and canvases. Owning references usually include the Objectholder hierarchy, the DSP graph and GUI clients. In order to avoid memory management overhead, the actual object disposal is done from the system thread.

[7]The implementation uses boost::shared_ptr or std::tr1::shared_ptr

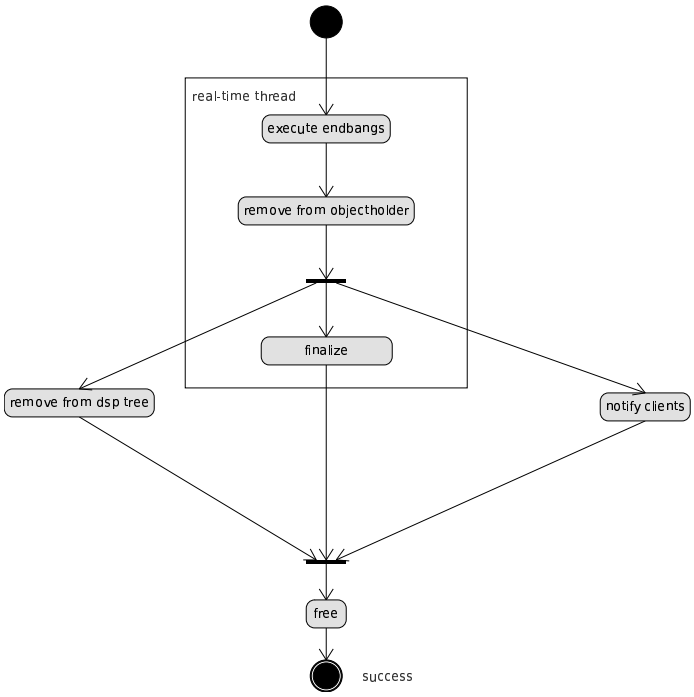


Figure 7: Object Lifetime: Object removal

3.5 Binding System

The basic concept of the use of Bindable Objects has already been described in Chapter 2.1.4. The implementation of this Binding System has to handle several problems:

Resource Lifetime Allocation and cleanup of bindables. Distinction between the handling of explicit and implicit bindables.

Resource Visibility Visibility of bindables in the patch hierarchy, handling of global and local bindables.

3.5.1 Bindable Resources & Bindable Allocators

Each bindable resource in nova is a class, that is derived from the `Bindable` base class, which provides a small interface for it's symbolic name and bind/unbind hooks, which are called at the time of binding and can be overridden from the implementation of a certain resource class. These hooks are required to implement resources, which need to know about their clients (e.g. busses).

Bindable resource types are usually referred to by their symbolic name (e.g. 'bus' for message busses). Thus it is necessary to provide the functionality to instantiate a bindable resource depending on their type name. In nova, this is solved using 'Bindable Allocators'. When a bindable resource type is initialized (this happens at the startup of the interpreter), an instance of a 'Bindable Allocator' is stored in a global 'Bindable Allocator Container', which provides a mapping from a symbolic name to a bindable allocator. At the time of resource instantiation, the specific bindable allocator can be looked up, which then allocates an instance of the requested resource.

3.5.2 Bindlists

Each Nova patch models the concept of a **Bindlist**[8]. A bindlist is one node in the tree of the bindable hierarchy. The root node of the bindable hierarchy is the environment itself. Each bindlist contains a dictionary of bindable types, that maps to a dictionary, which maps from a symbolic name to a bindable wrapper. This interface is only visible to bindable objects, though.

[8] The name 'list' has only been used for historic reasons

3.5.3 Bindable Wrapper

The same bindable resource needs to be located in different parts of the bindable hierarchy in order to allow the use of the `declare global` concept. In order to achieve that, the **bindable wrappers**, an additional indirection between the bindlists and the bindable resources, have been introduced. One needs to distinct between differnt kinds of bindable wrappers.

Local Bindable Owners Owner of a resource, that has been declared explicitly as local resource

Implicit Bindable Owners Owner of a resource, that has been declared implicitly and is thus a global resource

Global Bindable Owners Owner of a resource, that has been declared explicitly as global resource

Bindable Placeholder Reference to a Global Bindable Owner, that is located somewhere nested in the bindable hierarchy

Bindable wrappers are completely invisible to the bindable resources and to the user of the Nova language, but they maintain a list of their bindable clients.

3.5.4 Bindable Clients

Bindable Clients are objects, that can be bound to bindable resources. They expose all the functionality, that is needed to acquire or release bindings to resources as well as changing the binding to another resource.

When a bindable client tries to acquire a resource, it will search for a matching bindable wrapper upwards in the bindable hierarchy. If this search succeeds, the client is bound to the found resource. If no resource is found, the behavior is dependent, if an explicit or an implicit bindable resource is being searched for. If the resource is an explicit bindable, the binding request simply fails. In the case of an implicit bindable resource, a new instance will be allocated, which is then placed to the root node of the binding hierarchy and thus is visible in the whole scope unless it is shadowed by a local resource somewhere else in the hierarchy tree.

3.5.5 Bindable Declarators & the Mass Rebinding Problem

As described above, bindable resources can be declared explicitly somewhere in the bindable hierarchy using the `declare` object. This object can

be used to explicitly declare and allocate objects, that otherwise would be declared implicitly. However there are certain resource types, that should not be declared explicitly, but are declared using special objects (e.g. audio buffers, which need to allocated using the object `buffer~`).

Both the `declare` object and the special declaring objects are a model of 'Bindable Declarators'. Bindable declarators declare bindable resources which are allocated and located somewhere in the bindable hierarchy. This leads to a problem, that the visibility of bindable resources can change, when bindable declarators are added or removed. When a new bindable declarator is added, it injects a bindable wrapper to the hierarchy, which shadows all resources of this type and name, that are located upwards in the hierarchy. This leads to the fact, that all bindable clients, that are bound to this shadowed resource, need to be rebound in order to guarantee consistency. The problem is similar, when a bindable declarator is removed. In this case all bindable objects, that are bound to the corresponding bindable wrapper, need to be rebound.

3.6 Message Concepts

As described in Chapter 2.2, messages are used to pass control data between objects. All data types that can be used in messages, can be stored in the `atom` data type. Messages are emitted from the outlets and handled by the inlets. Inlets can only handle certain message types. Messages are always handled actively[9] by calling member functions of the object.

3.6.1 Extending the Type System

It is easily possible to extend the built-in type system from the C++ interface by deriving a class from the base class for the extendable type system (`CustomMessageType`). Classes of the custom message type are passed by reference. In order to be able to store custom messages in atoms, they need to be 'clonable' (this is achieved by implementing the virtual member function `CustomMessageType * CustomMessageType::clone(void)`).

In the Nova library, OSC messages are implemented using type system extensions.

[9] Other max-like languages use the notion of hot and cold inlets, where cold inlets are only storing the message data [Puc]

3.7 DSP Graph Concepts

3.7.1 DSP Graph

The DSP graph is a directed acyclic graph. The nodes of the graph are called Ugenholders, which means, it can provide a Ugen class (Unit Generator). The edges of the graph are the signal connections between xlets of nova objects.

In general, there would be two possible implementations for a DSP graph scheduling, a **static** scheduling algorithm or a **dynamic** one. The dynamic algorithm would imply a dataflow scheduling algorithm, where nodes can be executed, when all previous nodes have been executed and a node thus is runnable. The advantage of this approach is the ease of use in dynamically changing DSP graphs. The static scheduling algorithm requires a runtime compilation step in advance, computing the one-dimensional **DSP chain**, which specifies the scheduling order. The static scheduling algorithm has a better runtime performance, but the DSP chain generation may introduce an overhead.

The dynamic scheduling algorithm may be suited for small DSP graph (e.g. for jack graphs[10]), but as DSP graphs in max-like languages can easily use several hundreds or thousands of nodes, the scheduling overhead cannot be neglected. Thus the statical scheduling algorithm is widely used in computer music systems [Puc91b]. The DSP chain itself is triggered by the DSP backend (see Chapter 3.2.2). A detailed description of the DSP chain is given in Chapter 3.7.4.

3.7.2 Ugenholders & Ugens

The distinction between Ugenholders and Ugens is essential for the implementation of the DSP graph of nova. Each object of the nova language, that is usable in the signal processing part of nova, is a model of a Ugenholder and, as stated above, can provide a ugen class. Both Ugenholders and Ugens have the concept of 'ports', which is the equivalent to signal xlets for nova objects. While the ports of a Ugenholder are the direct equivalent to the signal xlets, as they are used to model signal connections at the scope of the DSP graph, the ports of the Ugens are only relevant for the generated DSP chain.

At the time of the DSP chain generation, each Ugenholder is requested to provide a Ugen, that should be used in the DSP chain. The Ugenholder is able to select an optimial ugen for the specific situation, mainly

[10] Jackdmp [SL05] uses a lock-free dynamic dataflow scheduling

depending on the used vector-size (e.g. for compile-time loop unrolling), or depending on the state of the DSP graph (e.g. to differentiate between message-driven and signal-driven oscillators).

3.7.3 DSP Contexts

As a refinement of the DSP graph concept, nova provides the concept of **DSP Contexts**. A DSP context is a nested DSP graph, which appears in the parent DSP graph as a Ugen. DSP contexts can be used to change some parameters of it's local DSP graph, such as suspending, reblocking or overlapping (see Chapter 2.3).

3.7.4 DSP Chain

As mentioned above, the DSP chain is the one-dimensional representation of the DSP graph, that describes the scheduling order of the ugens of the graph. The order of execution is simply the topologically sorted DSP graph.

The DSP chain itself is one of the most performance-sensitive parts of nova itself. The iteration across over the DSP chain needs to be very efficient and the data structures, that are stored in the DSP chain itself, should avoid indirections as much as possible to guarantee a good caching behavior. But unlike the implementation of the DSP chain of FTS and Pd [Puc91b], type safety and expressive power add additional constraints that make the implementation slightly less efficient. The data stored in the DSP chain for each ugen are references to the ugen, the used memory blocks, the vector size and a reference to a chain-specific data type.

When the DSP chain is generated, all memory blocks, that are used by the ugen need to be allocated. In order to minimize the memory transfers, the memory blocks are reused among several ugens. One constrain is, that the ugens always have the direct access to the memory regions of the output ports of their predecessors. In order to be able to reuse the signal vectors of predecessors, a reference count of connected input blocks per output block is maintained during the chain generation. Using this heuristic, memory chunks can be allocated in a reasonably efficient way. Especially, ugens will try to reuse the input signal vectors for their outputs to have a better memory locality. The signal vectors cannot be allocated from the memory arbitrarily, but need to be aligned to certain boundaries. The boundaries have to match two constraints. The first alingment constrain is necessary for the SIMD operations (see Chapter

3.10), which usually require a 16 byte alignment. The second constrain is not a necessary one, but a performance consideration. In order to minimize the utilized cache lines per signal vector, they should be aligned to cache line boundaries, which (on modern cpus) is usually 64 bytes. This alignment doesn't introduce any memory overhead for signal vectors of more than 16 samples.

When the DSP functions of the ugens are being executed, the memory regions that this ugen is working on need to be valid. Therefore some additional ugens are added to the chain explicitly.

Zero Ugens In the case of unconnected signal inlets, one needs to make sure, that the sample block is correctly wiped to zero. To ensure this, a shared signal buffer is used, that contains samples, that are set to zero. If the signal vector size of the specific DSP context is larger than the size of this shared vector, zero ugens, that clear an explicitly allocated signal vector are added to the DSP graph.

Add Ugen When multiple signal outlets are connected to one signal inlet, these signals need to be added implicitly. In this case ugens have to be added to the DSP chain, that prepare the signal input port by adding the audio signals of the audio outports. The adding operations are organized hierarchically, trying to add up to 4 signal sources at once in order to reduce memory overhead.

Certain signal vectors of unconnected xlets can be shared among different ugens. For unconnected inlets, a shared signal vector of samples, that are set to zero, is used. In a similar way, unconnected outlets can share a signal vector to store unneeded data. Only if the vector size of the DSP context is larger than the shared signal vectors, explicitly allocated memory has to be used.

The DSP chain generation algorithm itself is a simple graph traversal algorithm, which is split into two parts. The first part is a simple iteration over all ugenhoders of the DSP graph to initialize their ugens (see Chapter 3.7.2) and set up the data structures for the output block reference count. The second part is an actual depth-first graph traversal, building the actual DSP chain, adding implicit ugens and allocating memory for the signal vectors.

3.8 Client Concept (GUI Communication)

The nova interpreter itself is completely independent from a graphical user interface or a graphical patcher. However, it contains the concept of

observer clients. Observer clients are classes, that can be added to the interpreter at runtime, and that provide virtual member functions, which are called in the case of a state change in the nova interpreter. This way, they can provide the functionality that is needed to write a GUI, while offering a level of abstraction between the nova interpreter and the user interface. The communication from the clients to the server works just by calling the public API of the `environment` class.

Nova objects do not provide any hard-coded properties for their graphical representation (like size, position, color etc). Instead, they provide a general purpose dictionary allowing GUI clients to save and restore properties by their symbolic name.

3.8.1 GUI Object

All objects with a graphical representation are called GUI objects. The GUI objects are implemented in a way, that separates the graphical representation from the functional core. The implementation of the graphical part is a part of the observer client, while the functional part is written as ordinary nova object. This implies, that GUI objects can be loaded into the standalone interpreter without any requirements for the GUI client.

3.8.2 Patcher Prototype

At the moment, there exists only a prototype of a graphical patcher, which has been implemented using the Python bindings of the QT toolkit. The implementation of the patcher is split into two parts, a toolkit-independent wrapper from C++ to Python and a QT-specific Python part. The main nova classes, that are exposed to the Python are the abstract `observer` base class, canvas, object and connection classes and parts of the `environment` class (see Chapter 3.2). Beside that, GUI objects (and their related event classes) need to be exposed in order to write a toolkit-specific implementation. The Python bindings have been implemented using the Boost.Python framework [boo].

The QT-specific part is completely written in Python, implementing the `observer` class and a hierarchy of canvases and object boxes, which are addressed from the nova engine using UUIDs (see Chapter 3.3.1). For GUI objects the toolkit-specific implementation must be provided, using the exposed event classes for communicating with the nova interpreter. The canvas and object hierarchy is implemented using the "QT Graphics View Framework", which has been introduced in QT 4.2 [Tro].

3.9 Some Implementation Details

This section deals with some minor, but notable implementation concepts.

3.9.1 Shared Constant Resources

Certain resources of the nova interpreter are shared between class instances. In some cases the information can be stored in per-class data structures, in other cases this is not applicable or feasible (one case would be hint strings for xlets). In these cases one can make use of nova's constant-resource infrastructure. The constant-resource infrastructure is a set of C++ template classes, that allow resource sharing between instances.

The resources themselves are stored into a global intrusive container, while each owner just contains a reference to the shared resource in this global container. The shared resources are reference-counted in order to ensure the cleanup of the shared resource, when it's not needed any more. The access to the global container is threadsafe, but blocking, and thus not realtime-safe. But the global container is protected by reader-writer locks, which limits the necessity of the exclusive access by one thread. Since shared resources in the global container are unique, comparing unique resources is a lightweight operation, as expensive as comparing two pointers.

A similar framework called boost.flyweight has been developed by Joaquín M. López Muñoz. While lacking a fine-grained locking infrastructure, it is highly customizable and implemented using C++ template metaprogramming techniques. It has been accepted as Boost library in January 2008 [boo].

3.9.2 Threading Overview

As nova is a highly multithreaded program, this paragraph is giving an overview over the threading infrastrucure. There are two types of threads, running in nova. **Nova interpreter threads** are mandatory threads for the interpreter, while **object threads** are required for providing extra functionality, that is required by certain nova classes.

System Thread The System Thread handles most operations for the interpreter scope. Mainly, these are class loader operations (e.g. class resolution or object instantiation) and DSP graph manipulations. In addition to that, the system thread is used for memory management purposes (i.e. object deallocations).

Observer Thread The Observer Thread is a small helper construct to ease the communication with GUI clients. This thread is used to notify the registered observer client classes. This way, GUI notifications can be triggered from the realtime thread, without constraining the observer client implementations.

Nova Thread The Nova Thread is the thread, that is running the scheduler for the nova language (see Chapter 3.2.2). It is either a thread, that is started from the interpreter or that is started from the audio backend. The Nova Thread is considered as soft-realtime thread, meaning that functions, that are called from this thread, should not be blocking.

Network Thread Nova provides a framework for network communication. Network communication should be a low-latency operation although it is usually not real-time safe, which is the reason, why the networking framework is running in a separate thread.

Python Thread The nova library provides Python scripting objects. These objects make it possible to write nova classes in the Python programming language. While Python is a high-level programming language, it is not a realtime-safe language and thus should not be called synchronously from the nova thread. The Python objects therefore use this thread for evaluating Python bytecode. Depending on the number of instantiated Python scripting objects, multiple Python threads are required[11].

Soundfile Threads Accessing the hard drive for soundfile playback and recording is not possible in a realtime-safe manner either. Each object, that wants to read to or from soundfiles has to use a helper thread for the actual hard drive access and a lock-free ringbuffer for the communication.

3.9.3 Reusable DSP Framework

Nova provides a reusable DSP framework, that is written with C++ templates and can be reused from other applications. Some filters of the DSP framework are available as ladspa plugins [lad]. The DSP classes are designed with reusability in mind. The types that are used for the sample representation can be selected as template argument. The external sample type, that is used for the input and output signals, may be different to the internal sample representation. Other compile-time arguments are

[11] The current implementation is providing two global Python threads

specific to DSP classes. They include the maximum amplitude (phasor), the interpolation precision (wave-table oscillators), parameter interpolation (certain filters) or denormal bashing in feedback loops (IIR filters). The member function of these classes, that triggers the DSP tick, supports all signal vector implementations, that provide a sample extractor function.

3.10 SIMD in DSP Algorithms

Modern CPUs provide special instruction sets for operations that provide an operation to multiple data of the same type, as they are used when handling arrays or vectors. These instruction sets are called 'Single Instruction Multiple Data' (SIMD). Since there are SIMD instruction sets, that provide operations for floating point numbers, which are used to represent the audio samples in nova, the support of SIMD instructions in nova provides a big performance gain.

3.10.1 SIMD Hardware

Most modern CPU architectures provide a support for SIMD instructions. The most widely used architecture is the x86 architecture, which provides the SSE instruction set [Int] (beginning with the i686 architecture). The SSE instruction set is also supported on the 64-bit x86_64 architecture.

The SSE instruction set provides operations, working on a special set of 128-bit registers, called `xmm` registers, which can represent 4 32-bit floating point numbers. The SSE implementation on the i686 architecture provides 8 `xmm` registers, while the x86_64 architecture provides 16. SSE instructions are usually provided in two different variations, one for processing single scalars (e.g. `addss`) and one for processing 4 parallel scalars (e.g. `addps`). Traditional implementations require one CPU cycle for the single scalar operation and two cycles for a parallel scalar operation processing 4 scalars. Recent implementations, most notably the Intel Core architecture, implement the parallel scalar operation in one CPU cycle.

In addition to using additional registers, the SSE architecture requires the processed memory regions to be aligned to a multiple of 128 bit in order to efficiently transfer the memory between `xmm` registers and the memory.

3.10.2 Vectorizing Compilers

Generating vectorized code, that makes use of SIMD instructions, is a difficult problem for compilers. The two essential problems are the problems of **alignment** and **aliasing**. The first problem is, that the compiler doesn't have any information about the alignment of memory regions, except if the memory was allocated on the stack. Traditional pointers don't guarantee any alignment, that would be required for the automated generation of SIMD instructions. The second problem is the aliasing problem. Seeing two pointers, the compiler doesn't know, whether they point to overlapping memory regions or not and thus is not able to generate vectorized code.

If both problems can be addressed using non-portable, compiler-specific features, the compiler is theoretically able to generate vectorized code.

3.10.3 Classification of DSP Algorithms

There is a number of DSP algorithms, that can make use of SIMD algorithms. However they are different in the way, the source code needs to be transformed.

Vector Operations Vector operations are the simplest algorithms to vectorize. They can be both element-wise vector-vector or vector-scalar operations. These are the operations, that can be vectorized by auto-vectorizing compilers.

Vectorizable Branching Operations Vectorizable branching operations are operations, that need to be reformulated in order to be vectorizable, as they contain a branch in the original representation in the (C++) language. These operations include `min`, `max` or `clip` operations.

Vectorizable Bitmasking Operations Vectorizable bitmasking operations are operations, that require a bitmask to be used. Example operations are abs or sign operations or denormal bashing. They can be trivial bitmasking schemes (like `abs`), when the result is just bitwise ored with a bitmask or more complex (like denormal bashing or `sign`), where the result needs to be explicitly constructed from other bitmasks.

Vectorizable Special Operations Some operations are built into the instruction set as primitive instructions, which are not built into the (C++) language itself. Some operations are guarantied to be accu-

rate (e.g. the `sqrt` function of SSE), while some are only approximating the value (e.g. `rsqrt`) and thus would break the generation of these instructions would break the IEEE specifications for floating point operations.

Accumulating Operations Accumulating operations are not generating a vector as output, as the other operations mentioned above, but accumulate a value of one scalar value. Applications of these operations are peak-finding or power computation. These algorithms require two steps. The first step is an iteration across the whole memory section, accumulating the values to a vector of the size of one SIMD instruction (4 in the case of SSE). This vector now contains the accumulated values from interleaved vectors of the original memory section. In the second step, the values of this accumulated array will need to be accumulated to the resulting scalar.

3.10.4 SIMD in Nova

Nova provides a compile-time abstraction layer for SIMD algorithms. The only implementation is for the SSE instruction set and is based on compiler intrinsics, that are available for the C++ compilers from GNU, Intel and Microsoft. In addition to that, a fallback implementation in standard C++ in provided.

The SIMD framework provides two classes of implementations. Functions for dynamic vector sizes (which are required to be a multiple of 8) and functions for compile-time defined vector sizes. The second class of functions requires the vector size to be a multiple of 4 and generates completely unrolled machine code using the C++ template metaprogramming technique. Although the compile-time loop unrolled code is bigger than the looping code, the lack of jump instructions and the resulting linear program flow make the code very CPU-friendly. Depending on the operation and the vector size, the performance boost can be up to 50 %.

3.10.5 Micro-Benchmarks

On modern CPU architectures the performance gain of SIMD instructions is quite big. Table 5 shows the results of a micro-benchmark for the execution time of a plain C++ implemenation, a C++ implementation with a hand-coded loop unrolling of 8 samples, an SSE implementation unrolled by 8 samples and a completely unrolled SSE implementation. As vector-size 64 samples have been used, since that is nova's default vector-

size, the memory regions have been aligned to cache line boundaries. The benchmark was executed on an Intel Core2 Duo T7400 processor on a 32-bit debian linux system. The benchmark results were obtained using the oprofile profiler[12] and are normalized by the most efficient implementation. The binary has been compiled with GCC-4.2[13] at full optimization and architecture tuning.

Table 5: Benchmarks for vector-size of 64 samples

Operation	Plain C++	Unrolled C++	Unrolled SIMD	Completely unrolled SIMD
copy	6.54	4.76	1.24	1
vector/vector addition	6.08	4.52	1	1.31
vector/scalar addition	7.98	4.05	1.21	1
vector/vector minimum	9.45	9.11	1	1.09
vector/scalar minimum	9.58	14.06	1.63	1
absolute	8.70	3.13	1.37	1
sign	37.77	65.64	6.34	1
denormal bashing	7.58	-	1	1.58
sqrt	3.66	4.47	1.09	1
reciprocal	108	108	1	5.2
peak extraction	9.71	4.65	1	1.97
power summing	11.68	17.26	1	7.28
combined peak extraction & power summing	4.87	4.75	1	1.01

One can see, that for simple vector operations, a speedup of 4 can easily be achieved. Similar speedups can be achieved for operations like `abs` (bitwise and) or `sqrt` (built-in opcode), where the only difference to the compiler-generated code is the parallelization. For branching algorithms the speedup can be quite big. For simple branching operations (minimum and maximum), the speedup is bigger than 9 (although the generated code is free of branches). Bitmasking code (sign or denormal bashing) can perform significantly better, than the equivalent branching code. SIMD implementations of accumulating algorithms also perform between a factor 4 and 10 better. Due to the larger code size, the unrolled code may perform worse than the original code, though. Approximating operations (like the reciprocal approximation) can outperform their (exact) C++ equivalents by a factor of 100.

[12] http://oprofile.sourceforge.net/
[13] http://gcc.gnu.org/

For simple operations, complete loop unrolling can give another performance boost. If the code-size per sample grows too big, the code can perform worse due to caching behavior. Additional benchmarks show, that for larger vector sizes complete unrolling usually outperforms the iterating equivalents.

3.11 Framework for Lockfree Algorithms

Nova provides implementations of a few lock-free data structures, that are used mainly to ensure the real-time safety of the audio thread. Lock-free data structures are an area of active research in the last few year, as they are important in different kinds of applications like real-time or distributed systems. The important problems of lock-free algorithms are the requirement for special hardware instructions, the ABA problem, memory consistency and dynamic memory management.

Nova's lockfree framework provides generic C++ implementations of several lock-free data structures. They are implemented with template programming and template metaprogramming techniques.

3.11.1 Hardware Requirements

On recent CPUs, two different paradigms to write lock-free algorithms have been implemented [Moi97]. One way is to use "Compare-and-Swap" (CAS) instructions. CAS atomically compares a memory region with a value of a register and if this check succeeds, stores another value to this location. In addition to CAS instructions, which can only work on one machine word, DCAS instructions provide compare-and-swap operations for two machine words. Compare-and-swap instructions are implemented on the x86 and x86_64 architectures.

The second paradigm is to use pairs of "Load-Linked" (LL) / "Store-Conditional" (SC) instructions. LL instructions load the content of a memory region to a register. SC tries to store the content of a register to a memory region. This only succeeds if the content of this memory region has not been altered since the call to the LL instruction. LL/SC instructions are implemented on the PPC architecture.

3.11.2 The ABA Problem

The ABA problem [MS98] is one of the most important issues when writing lock-free algorithms using compare-and-swap. The ABA problem occurs in the situation, when one thread reads the value A, then another

thread changes this memory region to B and then back to A. When the first thread tries to update the memory region with a CAS operation, it finds the value A and thus the CAS succeeds. This may lead to an incorrect behavior if the algorithm assumes, that the memory region remained unchanged. The common solution for the ABA problem is to associate a counter with the specific memory region that is incremented on every modification [MS96]. This is the reason, why many lock-free algorithms require DCAS instructions. LL/SC-based algorithms are immune to the ABA problem, though.

3.11.3 Memory & Cache Consistency

Memory consistency is an implementation problem of lock-free algorithms. Both the compiler and the CPU are able to reorder the memory access [How], which can lead to problems, if an algorithm relies on the order of different memory accesses. On systems with CPU-specific cache lines, cache consistency is another problem. Changes to one value, may not necessarily be propagated to the shared memory immediately. To avoid these problems, memory barriers may have to be used. The lockfree framework of provides implementations of memory barriers for different platforms.

3.11.4 Dynamic Memory Management

Dynamic memory management of lock-free data structures is a non-trivial issue in languages without garbage collection like C++. A specific memory region cannot be returned to the operating system, unless it can be guaranteed that this memory region will not be accessed any more, i.e. no other holds a reference. Early solutions were to used a lockfree free-list as memory pool, never returning memory back to the operating system [MS96] or to use a reference count [Val95].

In order to avoid the overhead of reference counting, two algorithms have been developed. Maged Michael developed an algorithm with thread-specific 'hazard pointers' [Mic02b], which only requires atomic read and write instructions, and Maurice Herlihy, Victor Luchangco and Mark Moir came up with the 'Pass the Buck' algorithm [HLM02]. Two versions of the 'Pass the Buck' algorithm has been published, a wait-free version based on DCAS instructions, and a more efficient CAS-based (but not wait-free) version [HLMM05].

For the lockfree framework, both versions of the 'Pass the Buck' algorithm have been implemented and used for the provided data structures.

3.11.5 Queue

The most important lock-free data structure in nova is the lockfree queue, that is used extensively for synchronization purposes. The implementation of the queue in nova is based on a design by Maged Michael and Michael Scott [MS96], using the 'Pass the Buck' algorithm for memory reclamation [HLMM05]. Implementation is non-intrusive and generic. It can be configured to use a freelist to cache its internal data structures. To ease the use of smart pointers, a template specialization supporting dequeue operations to smart pointers is provided.

3.11.6 Stack

The lock-free stack is based on a singly-linked list [Tre86], that was extended in [HLMM05] to use the 'Pass the Buck' algorithm for memory management. The lockfree framework of nova provides two generic implementations, an intrusive and a non-intrusive one.

3.11.7 List-based Set

A sorted list can be used as a set data structure. Nova uses a list-based set to implement a lock-free hash table, as described in [Mic02a], which was adapted to be usable with the 'Pass the Buck' algorithm. The generic set implementation can be configured to use a special comparison function and a freelist to cache its data structures. The hash table is implementing a closed hashing, using a lock-free set for chaining the elements.

3.11.8 Low-Level Primitives

To ease the portability of the lock-free data structures, wrapper functions are provided for the commonly used low-level primitives like memory barriers or compare-and-swap instructions. These wrappers use built-in compiler primitives (e.g. for gcc), operating system specific function (e.g. for Win32 and OS X), and in rare cases assembler code for certain hardware or blocking emulations.

3.11.9 Utility Classes

Some utility classes are provided by the lockfree framework to ease the development of lock-free data structures.

atomic_int The atomic integer class provides an integer class with an API similar to the ordinary integer class, but assuring consistency

using atomic instructions and memory barriers.

atomic_ptr The atomic pointer class is a smart pointer class with an added tag to circumvent the ABA problem (see Chapter 3.11.2 for details).

pass_the_buck The pass_the_buck class implements the 'Pass the Buck' algorithm (see Chapter 3.11.4), which is used by several lock-free data structures, as object-oriented C++ class.

freelist The freelist class provides a caching memory pool, that is used in some data structures in order to cache reusable internal data structures.

4 Discussion and Conclusion

4.1 Conclusion

Although nova has not (yet?) gained a user base, its concepts proved to be highly powerful. I have been using nova for my own artistic works, both as a live instrument in more than 20 concerts and as a tool for composition. Due to the modular design, the components of nova are reusable in other systems, some code of nova has already been wrapped to be usable from within SuperCollider or from a LADSPA host.

The performance optimization, which has been an important point in the design of nova, has paid off. Synthetic benchmarks of the most common concepts, which are used in computer music applications, showed a speedup of about 1.84 compared to Pure Data and 1.76 compared to SuperCollider.

Finding a user base or even developers turned out to be more difficult than I originally expected. Now, after more than 2.5 years of development and almost one year of a graphical user interface, nobody tried to use nova, despite a talk on the Linux Audio Conference 2007 in Berlin. Some people offered their help in the development, but unfortunately nobody found the time to really contribute to the source code. I am not completely sure, what is the reason for that, but my guess is that most people, who are competent software developers are busy with other projects and thus do not find the time to contribute to other projects than their own.

4.2 Future Directions

In the future, some concepts of the nova language are likely to change. In the current approach, control structures and audio synthesis are tightly coupled and several decisions are made implicitly by the language and are not exposed to the user. Systems, which are based on a scripting language like SuperCollider, have some algorithmic advantages. A syntactic way to access the synthesis core of nova is one of the most challenging parts in the future development of nova.

4.3 Acknowledgement

I would like to thank Wolfgang Musil and Klaus Filip for their feedback based on their huge experience with Max/MSP and Dieter Kovacic for hosting the project website and source repository on his server. Finally I would like to thank Miller Puckette for writing Pure Data and releasing it

as open source software, which greatly influenced the design of nova, and for his conservative development process, which was the reason for me to start nova as project independent from Pure Data.

References

[AG06] Y. Orlarey A. Graef, S. Kersten. DSP Programming with Faust, Q and SuperCollider. In LAC, editor, *Linux Audio Conference 2006*, 2006.

[Ama05] X. Amatriain. *An Object-Oriented Metamodel for Digital Signal Processing with a focus on Audio and Music*. PhD thesis, 2005.

[boo] www.boost.org.

[Bou00] R. Boulanger, editor. *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing,and Programming*. The MIT Press, March 2000.

[CS] Perry R. Cook and Gary P. Scavone. *Synthesis ToolKit Manual*.

[CS99] Perry R. Cook and Gary P. Scavone. The Synthesis ToolKit (STK). In *Proceedings of the International Computer Music Conference. International Computer Music Association*, 1999.

[cyca] http://www.cycling74.com.

[Cycb] Cycling74. *MSP Reference Manual*.

[DBdC+98a] François Déchelle, Riccardo Borghesi, Maurizio de Cecco, Enzo Maggi, Joseph B. Rovan, and Norbert Schnell. jMax: a new JAVA-based editing and control system for real-time musical applications. In *ICMC: International Computer Music Conference*, Ann Arbor, USA, October 1998.

[DBdC+98b] François Déchelle, Riccardo Borghesi, Maurizio de Cecco, Enzo Maggi, Joseph B. Rovan, and Norbert Schnell. Latest evolutions of the FTS real-time engine: typing, scoping, threading, compiling. 1998.

[Dé] François Déchelle. A brief history of MAX.

[Gra07] Albert Graef. Interfacing Pure Data with Faust. 2007.

[HLM02] M. Herlihy, V. Luchangco, and M. Moir. The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures. *Distributed Computing: 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002: Proceedings*, 2002.

[HLMM05] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems (TOCS)*, 23(2):146–196, 2005.

[How] David Howells. *Linux Kernel Memory Barriers*.

[Int] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

[lad] http://www.ladspa.org.

[Laz01] V. Lazzarini. Sound processing with the sndobj library: an overview, 2001.

[Lin90] Eric Lindemann. ANIMAL - A Rapid Prototyping Environment for Computer Music Systems. In *Proceedings of the International Computer Music Conference*, 1990.

[LW07] V. Lazzarini and R. Walsh. Developing LADSPA plugins with Csound. In *Proceedings of the 5th International Linux Audio Conference*, pages 30–36, Technical University Berlin, 2007.

[Mat] Max Mathews. History of Computer Music According to Mathews.

[McC02] James McCartney. Rethinking the Computer Music Language: SuperCollider. *Comput. Music J.*, 26(4):61–68, 2002.

[Mic02a] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, New York, NY, USA, 2002. ACM Press.

[Mic02b] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *PODC*, pages 21–30, 2002.

[Moi97] M. Moir. Practical implementations of non-blocking synchronization primitives. *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 219–228, 1997.

[MS96] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

[MS98] M. M. Michael and M. L. Scott. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[NS03] S. Letz D. Fober N. Scaringella, Y. Orlarey. Automatic vectorization in Faust. In JIM, editor, *Actes des Journées d'Informatique Musicale JIM2003, Montbeliard*, 2003.

[Pop04] S. T. Pope. *Sound and Music Processing in SuperCollider*. 2004.

[Puc] M. Puckette. *Pd Reference Manual*.

[Puc91a] M. Puckette. Combining Event and Signal Processing in the MAX Graphical Programming Environment, 1991.

[Puc91b] M. Puckette. FTS: A Real-time Monitor for Multiprocessor Music Synthesis, 1991.

[Puc96] M. Puckette. Pure Data: another integrated computer music environment. In *Proc. the Second Intercollege Computer Music Concerts*, pages 37–41, 1996.

[Puc97] M Puckette. Pure Data: recent progress. In *Proceedings, Third Intercollege Computer Music Festival*, 1997.

[Puc02] Miller Puckette. Max at Seventeen. *Computer Music Journal*, 26(4):31–43, 2002.

[Scha] Bill Schottstaedt. *Common Lisp Music Manual*.

[Schb] Bill Schottstaedt. *SND Manual*.

[SL05] D. Fober S. Letz, Y. Orlarey. Jack audio server for multiprocessor machines. In ICMA, editor, *Proceedings of the International Computer Music Conference*, pages 1–4, 2005.

[STP03] Chandrasekhar Ramakrishnan Stephen Travis Pope. The CREATE Signal Library ("Sizzle"): Design, Issues, and Applications. 2003.

[Tre86] R. K. Treiber. *Systems Programming: Coping with Parallelism*. International Business Machines Inc., Thomas J. Watson Research Center, 1986.

[Tro] Trolltech. *Qt Reference Documentation*.

[Val95] J. D. Valois. Lock-free linked lists using compare-and-swap. *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222, 1995.

[Ver07] Barry Vercoe. *The Canonical Csound Reference Manual Version 5.07*, 2007.

[vS] Arie van Schutterhoef. Sogitec 4X.

[WC04] Ge Wang and Perry Cook. ChucK: a programming language for on-the-fly, real-time audio synthesis and multimedia. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, pages 812–815, New York, NY, USA, 2004. ACM.

[WF97] M. Wright and A. Freed. Open Sound Control: A New Protocol for Communicating with Sound Synthesizers. In *In Proceedings of the 1997 International Computer Music Conference*, pages 101–104, 1997.

[YO02] S. Letz Y. Orlarey, D. Fober. An Algebra for Block Diagram Languages. In ICMA, editor, *Proceedings of International Computer Music Conference*, pages 542–547, 2002.