On The Design of Extensible Music Authoring Tools

Vamshi Raghu

Music Technology Area
Schulich School of Music
McGill University
Montreal, Canada

January 2007

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Master of Arts in Music Technology.

2007/01/21

Abstract

The past half-decade has seen progress in methodology and reusable components available to designers of music authoring tools. This thesis examines currently prevalent architectures for music making software and applies currently available technical means to update design methodology and architectural patterns for the next-generation of tools. It aims to map the various categories of advances and the manner in which they relate to each other, to the problem of building these tools. The focus is on conceptualization. The thesis aims to understand, from historical perspectives, as well as from perspectives provided by other domains, the fundamental problems encountered in the process of designing authoring tools.

Issues examined include building rich visual and interactive interfaces for authoring, the use of multiple notations and formalisms to describe multiple aspects of musical structure, end-user extensibility and end-user scriptability. The results of design experiments implementing core ideas are documented, and the manner in which the ideas from these prototypes may be applied to the construction of real-world tools is discussed. As far as possible, the thesis investigates existing tools, frameworks, design ideas, and architectural possibilities that scale. In conclusion, the manner in which the investigation relates to the future of authoring tools, and to problems faced by contemporary artists and tool-makers is discussed.

Sommaire

Les concepteurs d'outils logiciels pour la création musicale ont á leur disposition les progrés en méthodologie et en composants réutilisables réalisés ces cinq derniéres années. Cette thése examine les architectures actuelles des outils de conception musicale, et applique les moyens techniques actuels pour améliorer la méthodologie et les modéles architecturaux qui permettront de concevoir la prochaine génération d'outils. Cette thése essaie d'établir le lien entre les diverses catégories d'avancés, leurs inter-relations et les problémes d'élaboration d'outils de conceptions musicale. L'emphase est mise sur la conceptualisation. La thése essaie de comprendre, selon une perspective á la fois historique et éclairée par les autres domaines, les problémes fondamentaux rencontrés dans le processus d'élaboration d'outils de conception musicale.

Les problémes examinés incluent la conception d'interfaces visuelles riches et interactives pour la conception musicale, l'utilisation de plusieurs notations et des formalismes de descriptions des divers aspects des structures musicales, l'extensibilité par les utilisateurs ainsi que la capacité á écrire des scripts. Les résultats de prototypage de systémes de conception implémentant les idées principales sont documentés, et les maniéres dont les idées déduites de ces prototypes peuvent être appliquées á l'élaboration des outils réels sont discutées. La thése investigue autant que faire se peut les outils, les environments, les idées de conception, et les possibilités architecturales qui s'accorde bien au probléme de construire les outils réels. Enfin, la façon dont la recherche se relie au futur des outils de conception musicale, et aux problémes auxquels les artistes contemporains et les concepteurs d'outils font face.

Acknowledgments

I would like to thank my thesis advisor, Gary Scavone, for supporting and encouraging my ideas and line of investigation, and for his constant support and instant feedback. He let me pursue my ideas in a line of investigation where the conceptual nature of the work made it difficult to pre-plan and articulate the research in earlier stages, and yet, kept me focussed on the scope of what was acheivable in the time available for a masters thesis. I would like especially, to thank him for his extra work, including weekends, to help me meet the submission deadline. Thanks Gary!

A number of people have contributed both directly and indirectly to the ideas and content of this thesis. I would like to thank Hans Vangheluwe from the School of Computer Science at McGill for his overwhelming enthusiasm for interdisciplinary work and for always keeping the door to his office open (a courtesy that I took great advantage of). He was instrumental in helping me make the connection with domain-specific modeling and model-driven architecture. AtoM^3, the tool that led me to meet Hans and GME, another tool I was referred to by Hans, provided me with first-hand experiences in building multi-formalism modeling tools. Thanks Hans!

I would like to thank Jim Hugunin from the Common Language Runtime team and Christian Schormann, Unni Ramakrishnan and Amir Khella from the Sparkle team at Microsoft for personally taking the time off to meet with me to discuss the manner in which their tools could be applied in the context of my work. Thanks to Ganesh Anandan for many discussions on applying software and electronics to his music, and for introducing me to artists and bands in Montreal. Thanks to Cory McKay and Philippe Depalle for helping me translate the abstract to French.

Financial assistance from my parents, the State Bank of India, the Schulich School of Music and employment from McGill ICS made my graduate studies possible. I would like especially to thank McGill ICS for tolerating my intermittent absence and accomodating innumerable changes to schedule. Special thanks to Elia Salera, Anna Mancini, Anastasia Vasilopanogos and Miriam Zia for accomodating the changes, to Kim Huynh for hiring me, and to Swaminathan Mahadevan for referring me to ICS. Thanks to everyone at ICS Help Desk, ICS Desktop and NCS for the light-hearted atmosphere at work.

None of my pursuits would have been possible if not for the support from my family. It is impossible to thank them sufficiently. So I will not attempt it.

Contents

List of Figures

Chapter 1

Introduction

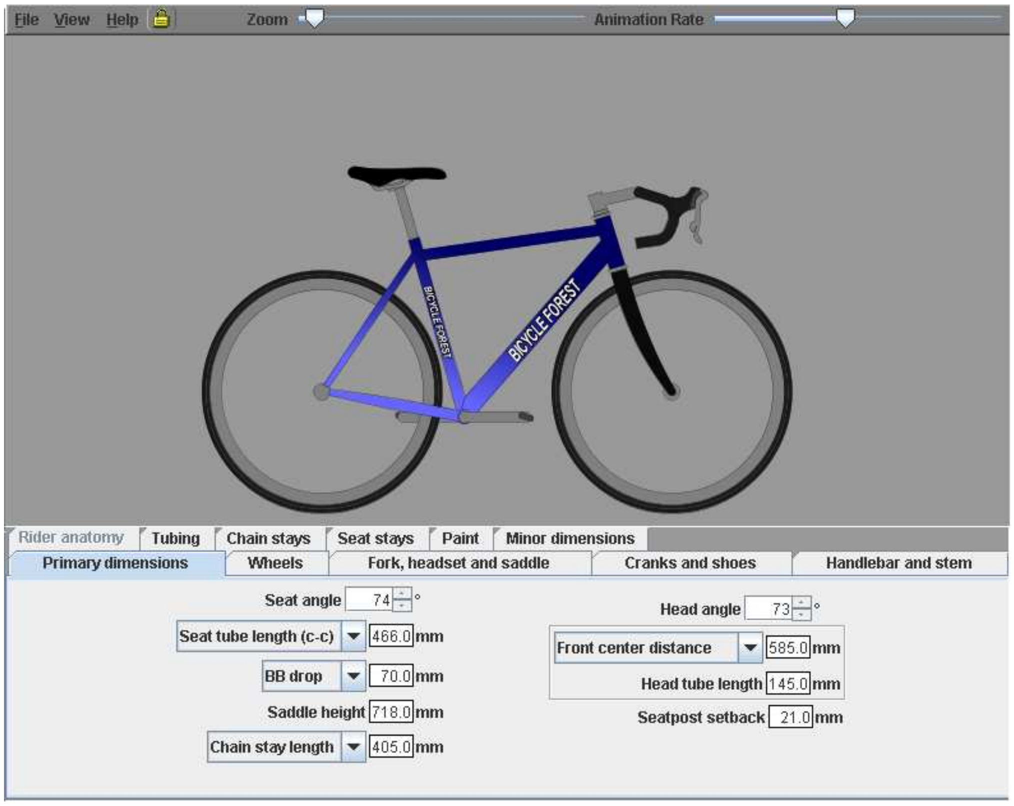
1.1 Context

The current work concerns the application of recent innovations in programming languages, modeling environments, software/component engineering and human-computer interaction to the problem of constructing tools for music designers. In particular, the research surveys the technology currently available to construct music authoring tools and documents the results of several design experiments with tools, frameworks and methodologies from other domains to illustrate how these could be used to construct better tools for music authoring and sound design. It aims to be an updated 'field guide' for music software architects, designers and engineers.

The work focuses on practical considerations involved in arriving at an architecture for music authoring tools that best leverages currently available technical means and anticipates the evolution or extension of the design to serve different users of such a tool. A recurring theme in the thesis is end-user programmability. An example of an end user is a musician, sound designer or sound engineer. Allowing end users to extend the tool using terms and patterns they are familiar with enables them to customize the tool in a manner that the original designer could have neither anticipated, nor had the musical expertise to realize. An environment in which users can extend the tool using terms and notation from their domain of expertise is called a 'Domain Specific Design Environment'. The ultimate goal is to enable musicians to customize their tools using familiar (if not simple) terms, and for these customizations to be made available to other musicians with similar needs.

1.2 Motivation

The need for such an extension environment has arisen in many other domains. Figure 1.1 shows a parametric Computer Aided Design (CAD) tool for bicycle design [Curry, 2006]. Such a system can be used by a cycling enthusiast with no knowledge of mechanical engineering or CAD to generate valid bicycle designs using a constrained design environment. The strength of such a tool lies in its ability to allow the user to generate designs that conform to a notion of correctness as defined by the original programmer of the design tool. The same strength is also a weakness when the end user wants to alter this notion of what a correct and useful design is, and expand it to include a range of designs that the original programmer did not anticipate.



Seat angle 74°
Seat tube length (c-c) 466.0 mm
BB drop 70.0 mm
Saddle height 718.0 mm
Chain stay length 405.0 mm
Head angle 73°
Front center distance 585.0 mm
Head tube length 145.0 mm
Seatpost setback 21.0 mm

Fig. 1.1 Parametric CAD tool for Bicycle Design. The Bicycle Forest, Inc.

The problem stems from the fact that the underlying tool itself is implemented directly in a general-purpose programming language like C, C++, Java or Python[1]. Such a framework provides a plethora of facilities for describing any tool conceivable in any domain.[2] While such a language is *expressive* enough to describe any tool, it is not an *efficient* vehicle of expression. Usability on the one hand, and flexibility on the other, thus seem to present conflicting requirements for a tool designer.

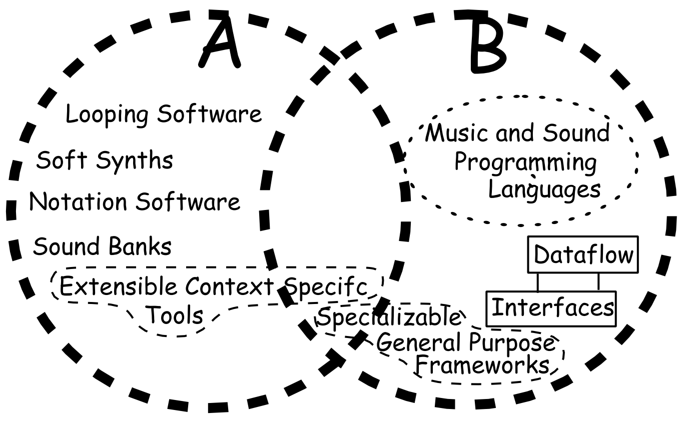

A
B
Looping Software
Soft Synths
Notation Software
Sound Banks
Extensible Context Specifc Tools
Music and Sound Programming Languages
Dataflow
Interfaces
Specializable General Purpose Frameworks

Fig. 1.2 Categories A and B

In the context of music authoring and sound design tools, this dichotomy manifests itself as two categories of audio software as shown in Figure 1.2. Category A consists of tools designed to do a *specific task or set of tasks* very well and Category B consists of systems that provide a *language* or a *framework* for expressing a very large number of musical tasks. In each category, it is anticipated that Music Authoring Tools will evolve to occupy the ideal space in the intersection (A∩B), resulting in context specific tools that are easily extensible, and general-purpose tools that are easily specializable. Categories A, B and A∩B are revisited at various points in later chapters. They are central to some of the questions that are addressed by the thesis in the context of tools and frameworks available today: How can we make music authoring tools that are both usable and flexible? How can we make music authoring tools that are extensible by end-users? How can we 'boot-strap' such an extensible environment from a small authoring-system core?

[1]One of the ideas explored in the thesis is to build multiple languages for describing functionality at various levels of implementation.

[2]Provided that the tool can be realized by a computable process.

1.3 Scope and Contributions

The work aims to explore existing technology with the aim of reusing it. By integrating various existing components, it aims to arrive at a conceptualization of what is anticipated to be the architecture of the next generation of music authoring tools. It conducts practical prototyping exercises to confirm design hypotheses. Chapters 2 and 3 introduce the state-of-the-art in computer-music software and frameworks and tools in other domains that were identified as being useful to the problem of constructing authoring tools. Chapter 2 identifies progress in computer music software and anticipates the features that will be available in future computer-based music authoring and prototyping systems. It motivates the design investigations that are conducted as part of the current work. Chapter 3 identifies tools and frameworks from other domains, like graphics engines, visual modeling environments and scripting and extensibility engines that may be plugged together to form an extensible core for an authoring application. It also tries to identify various classes of frameworks that are representative of the state-of-the-art in these domains. Chapter 4 details the results of design investigations with tools identified in Chapter 2 and 3 to build prototypes of various components of a music authoring tool. Architectural and design possibilities are illustrated by demonstration. Chapter 5 summarizes the results and identifies possible directions for future investigations and implementation.

Chapter 2

Background

The design of computer music tools in general and music authoring tools in particular has a long and rich history, almost as old as the history of digital computers. Advances in computer music authoring software have mirrored advances in computer hardware and (perhaps more so) software. In essence, the basic functions of a computer music authoring system have remained the same. Incremental improvements have been made in terms of usability, performance, extensibility and cost. New generations of music authoring tools are usually distinguished by a degree of improvement in the aforementioned areas that, in comparison with the previous generation, becomes a difference in kind rather than a difference in degree. It is a common observation that design tools within the same generation seem to all 'do the same thing' with subtle differences that get consolidated into significant differences *across* generations.

This chapter identifies the components of a music authoring tool and traces intra-generational and inter-generational progress in computer music systems. It identifies current advances in computing relevant to each of the components of a music authoring tool that may anticipate a new generation of music authoring tools. It also additionally motivates and sets the context, from various historical perspectives, for the investigations and design experiments conducted as part of the current work. It tries to classify various systems as Category A or Category B systems.

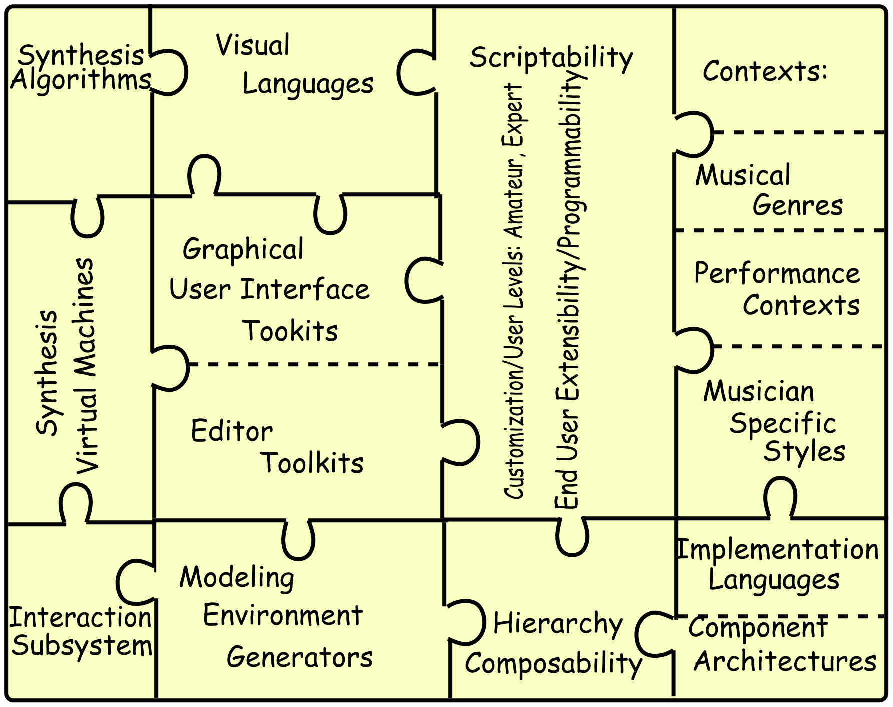


Fig. 2.1 Problems related to the design of music authoring tools

2.1 Pieces of the Puzzle

Figure 2.1 shows sub-problems encountered in the design of music authoring tools. The remaining portion of this section introduces these sub-problems. In some cases a description of existing art in the sub-problem is given, if relevant.

2.1.1 Infrastructure Frameworks

Various efforts in the history of computer music have focused on different sets of sub-problems. Of these, synthesis algorithms, synthesizer design (increasingly in the form of a virtual machine that accepts primitive synthesis-commands) and the problems related to formalizing musical contexts are peculiar to the music domain. The remaining problems are general problems in software engineering and architecture. Nevertheless, they are germane to the problem of constructing music authoring tools. Common sub-problems encountered in various domains are often referred to, in software engineering, as 'infrastructure' problems, and the frameworks that are developed to address these are known as 'infrastructure-frameworks'.

Since these problems are common to many domains and hence useful to many people, work on infrastructure frameworks is always ongoing and progress in software in many domains is determined by the state of the art in infrastructure frameworks. Very often, these frameworks are baked into the operating system itself, or inside development tools and frameworks. Examples of such frameworks are those provided with the operating system APIs of Unix, GNU/Linux, Mac OS and Windows families of OSes. The Java Virtual Machine (JVM) and the .NET Common Language Runtime (CLR) are language and OS independent abstractions over computer hardware and operating systems, provided in the form of a universal programming-language-runtime available on different platforms.

It is sometimes also the case that infrastructure frameworks expand to include components from progress made in specialized domains that may be useful to many others, including music authoring. Current OSs are beginning to include advanced graphics subsystems that were once considered neccesary only for graphics intensive applications like CAD and 3D games. Similarly, compilers and interpreters for domain-specific languages defined for a specific application had to be written from scratch. Current versions of language runtimes (such as the CLR) include compiler front-ends and template-based byte code generation as part of the API.

By aligning the design of music authoring tools to inter operate with the design of existing infrastructure frameworks, we can design them to take advantage of future advances in infrastructure frameworks without re-design of the authoring tool itself. The following sub-sections present background material related to domain-specific problems as well as infrastructure problems.

From a more radical - or perhaps creative - perspective, Atau Tanaka [Tanaka, 2003] discusses how artists are adapting their creative ends to match the new means available in the form of kinds and layers of infrastructure over which to build real and virtual tools and instruments. Thus, the layers of infrastructure define the medium available at the disposal of the artist, and the evolution of these layers of infrastructure frameworks influences the creative direction of electronic and computer-based arts in general and electronic and computer-based music in particular.

2.2 Evolution of Audio Synthesis Languages and Frameworks [Category B]

2.2.1 CSound and the Music-N Family

The grand daddy of all computer music software is a program called MUSIC written in 1957 by Max Matthews at Bell Labs. MUSIC evolved into an entire family of languages, MUSIC-II, III, IV, IV-B, IV-BF, V and MUSIC 360. Of these, MUSIC II through IV and V were developed at Bell Labs. MUSIC IV-B was developed at Princeton University. MUSIC 360 was developed at the MIT Media Lab by Barry Vercoe. This later evolved into a program called CSound that is being widely used in the computer-music community to this day [Wikipedia, 2006i].

The MUSIC-N family introduced idioms for the design of sound generation software that continue to be used in current designs. The software differentiates between two types of data streams generated by computational processes that are called 'signal-rate' (or sampling-rate) and 'control-rate', respectively. Signal-rate processes specify computations that occur between every sample of the digitized sound (in the case of CD quality mono audio this would be 44,100 Hz). These are normally used to model the signal (sound) structure and are implemented by functions called unit generators (or 'opcodes') written in C. The unit generators are parameterized by control variables that often change much less frequently than the signal-rate[1], often at rates not exceeding 10 Hz. The processes and data that describe the control variables are called 'scores' because they model the musical structure of the piece being rendered. Figure 2.2 shows a modern CSound data file with both orchestra and score combined into a single XML file [Wikipedia, 2006b].

2.2.2 Max/MSP, Pd, OSW and SynthBuilder/MusicKit

Visual environments involving patch-cord based interfaces for music authoring and sound design can be traced back to a program called 'Patcher' written by Miller S. Puckette in the mid-1980s as a tool for realizing a piece by Philippe Manoury called 'Pluton' [Wikipedia, 2006j; Puckette, 1991]. Patcher can be considered a descendant of the MUSIC-N family and the Patcher program itself led to a family of visual sound design and event sequencing environments, the most notable of them being a commercial variant sold by Cy-

[1]FM Synthesis is an exception.

```
<CsoundSynthesizer>;

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

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

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

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

</CsoundSynthesizer>
```

Fig. 2.2 A modern CSound data file

cling'74 called Max/MSP. Miller Puckette went on to develop a free version of Max/MSP called Pd [Puckette, 1996]. An environment with a similar interface was also developed by Amar Chaudhary at the University of California at Berkeley called Open Sound World (OSW) [Chaudhary et al., 1999].

Figure 2.3 shows a Max/MSP patch (left) and an OSW patch (right). As can be seen in the figures, Max/MSP, Pd (very similar to Max/MSP) and OSW provide a data-flow based language to describe both signal generation/processing and event generation/processing. Max/MSP and Pd share a common architecture, though not the same code-base. OSW has a significantly different architecture that centers around providing an extensible type system [Chaudhary et al., 1999]. SynthBuilder is a similar system that was developed to prototype various signal processing algorithms developed at CCRMA, Stanford University [Porcaro et al., 1998]. It was one of the earlier systems that emphasized extensibility and

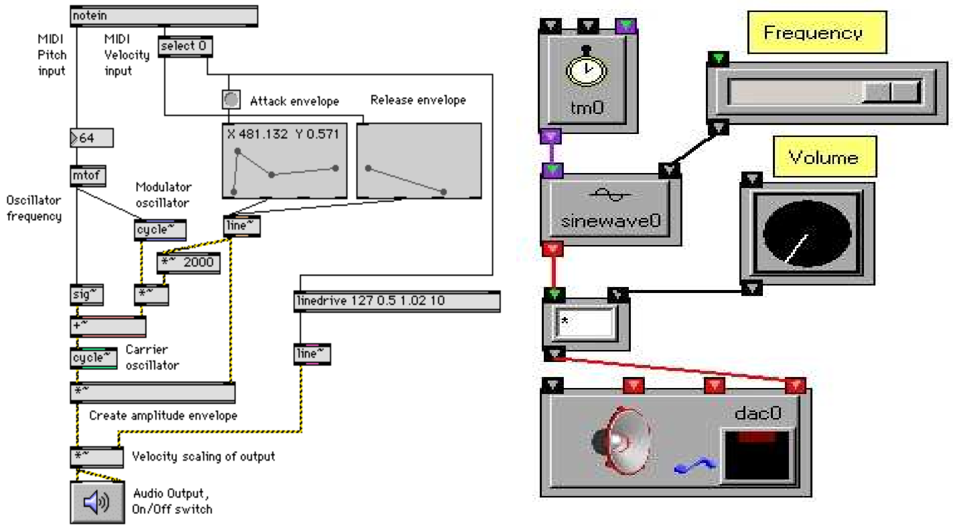


Fig. 2.3 A Max/MSP patch

leveraged the capabilities of a rich infrastructure platform (NeXTStep). It is a common observation that whereas the data-flow representation works well to describe signal-generation and transformation networks, it gets messy when used to describe control-flow or event generation and processing.

Visual editors for these environments were built from scratch. Section 3.3 describes current technology that makes developing sophisticated visual editors easier by reusing existing off-the-shelf components.

2.2.3 SuperCollider, CapyTalk/Kyma and ChucK

In essence, interpreted text-based computer-music languages described in this section are functionally identical to the visual computer-music languages described in the previous section. All three text based languages, however, currently factor out their synthesis subsystem into a separate process. The synthesis language itself is implemented as a client application that runs either as a console or as a visual environment much like an IDE (Integrated Development Environment).

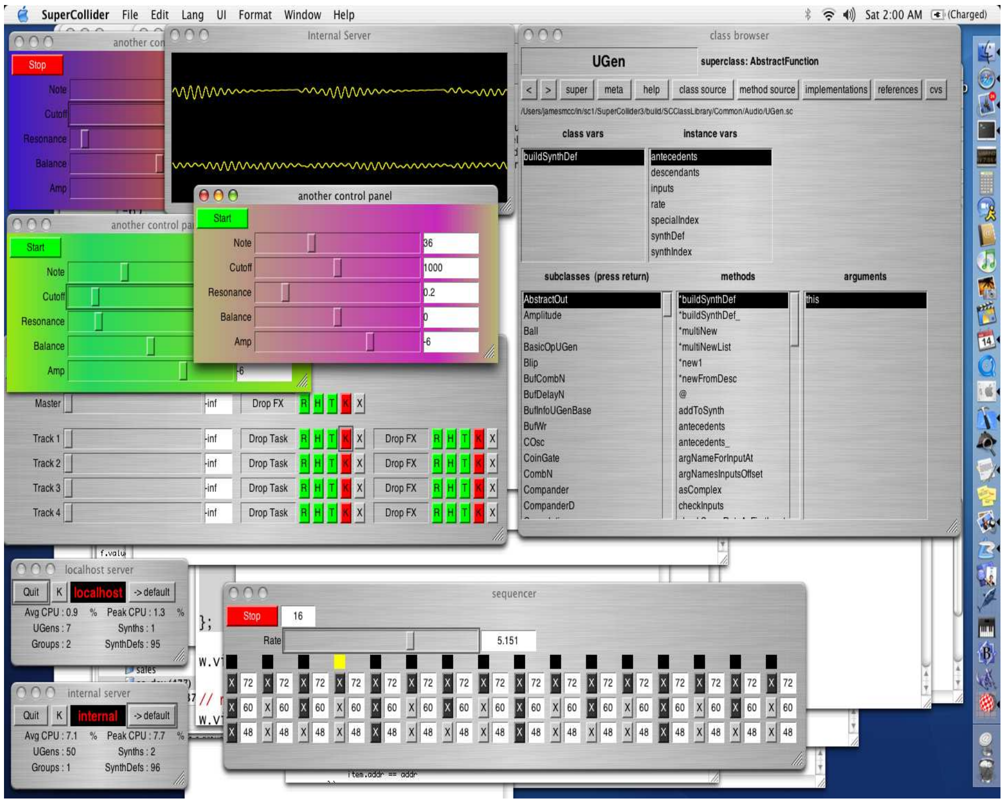


Fig. 2.4 SuperCollider Client running on Mac OS X [McCartney, 2006]

SuperCollider

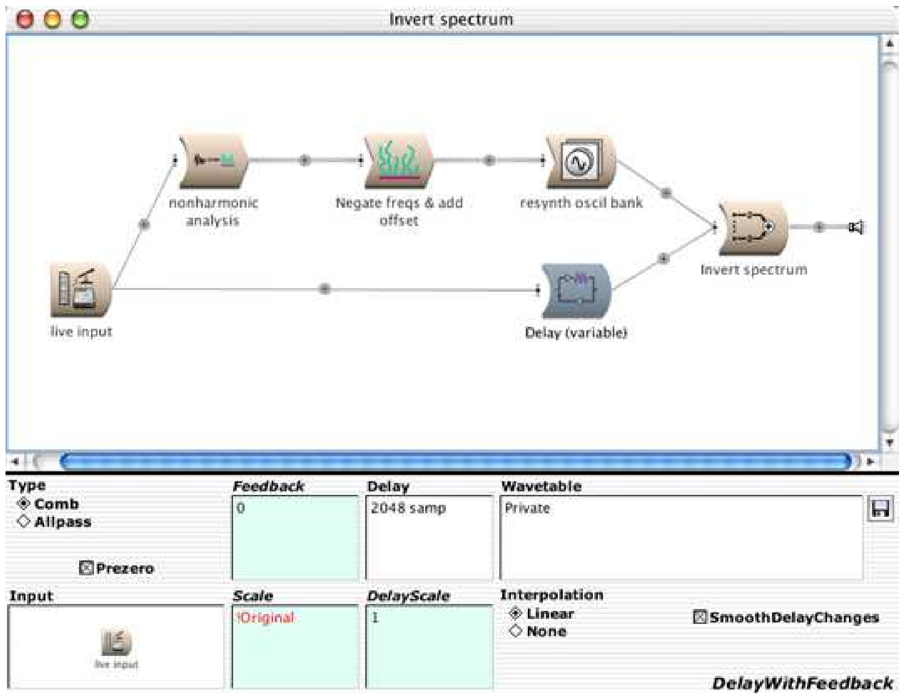
SuperCollider, chronologically the first in this family, was initially announced as a commercial product on the newsgroup comp.music.research on March 21, 1996. It ran only on the Power Macintosh and shipped as a single application that combined both the visual environment, programming language and synthesizer. The language itself was quite similar to its current incarnation and supported incremental garbage collection, first class functions (closures) and a pure object-oriented type system along the lines of SmallTalk. It supported a variety of visual interfaces to create SuperCollider objects, including patch-cord based interfaces and envelope editors.

What set Supercollider apart was that its base language was a well-structured variant of SmallTalk. This allowed for structured, object-oriented descriptions of complex synthesis

```
// play a mixture of pink noise and an 800 Hz sine tone
{ SinOsc.ar(800, 0, 0.1) + PinkNoise.ar(0.01) }.play;

// modulate the sine frequency and the noise amplitude with another sine
// whose frequency depends on the horizontal cursor position
{
        var x = SinOsc.ar(MouseX.kr(1, 100));
        SinOsc.ar(300 * x + 800, 0, 0.1)
        +
        PinkNoise.ar(0.1 * x + 0.1)
}.play;

// list iteration: create a collection of indices multiplied by their values
[1, 2, 5, 10, -3].collect { |item, i| item * i }

// factorial function
f = { |x| if(x == 0) { 1 } { f.(x-1) * x } }
```

Fig. 2.5 A SuperCollider script

processes. Event processing is described using object-oriented text-based code, and signal processing networks are edited using the patch-cord interface. The dynamic, interpreted nature of SmallTalk also facilitates interactive prototyping, with changes in the synthesis code reflected in real-time changes to the audio output, much like Max/MSP. SmallTalk was designed specifically with the purpose of supporting event-driven programming (although in the context of graphical user interfaces). SuperCollider integrates real-time audio processing into the Object-Method-Message model provided by SmallTalk, which works rather well to describe the evolution of state by asynchronous interaction between many concurrent processes. Further, SuperCollider also provides for a functional style for description of unit generators, which is perhaps a more natural style for specifying these connections as explained in Dannenberg [2002]. Figure 2.5 shows a SuperCollider script.



Type
◈ Comb
◇ Allpass
☒ Prezero

Feedback: 0

Delay: 2048 samp

Wavetable: Private

Input: live input

Scale: !Original

DelayScale: 1

Interpolation
◈ Linear
◇ None

☒ SmoothDelayChanges

DelayWithFeedback

Fig. 2.6 Patch-cord interface in Kyma

CapyTalk/Kyma

The Kyma/CapyBara is a combination of hardware and software for sound synthesis and sequencing designed and sold by Symbolic Sound Corporation. The Kyma is the software portion of the unit. Similar to SuperCollider, the scripting language for the Kyma environment is a variant of SmallTalk, named CapyTalk. It was originally written as the synthesis language for a computer music workstation designed at the University of Illinois at Urbana Champaign called Platypus [Scaletti, 1989]. Similar to SuperCollider, Kyma also provides alternate ways to create synthesis and sequencing objects. They may be created by writing CapyTalk code or by using a patch-cord based interface as shown in Figure 2.6. The differentiating feature of the Kyma is that it can run on a dedicated sound synthesis hardware system called 'CapyBara'. The CapyBara is a rack of DSPs that can be scheduled to run Kyma patches and CapyTalk objects.

ChucK

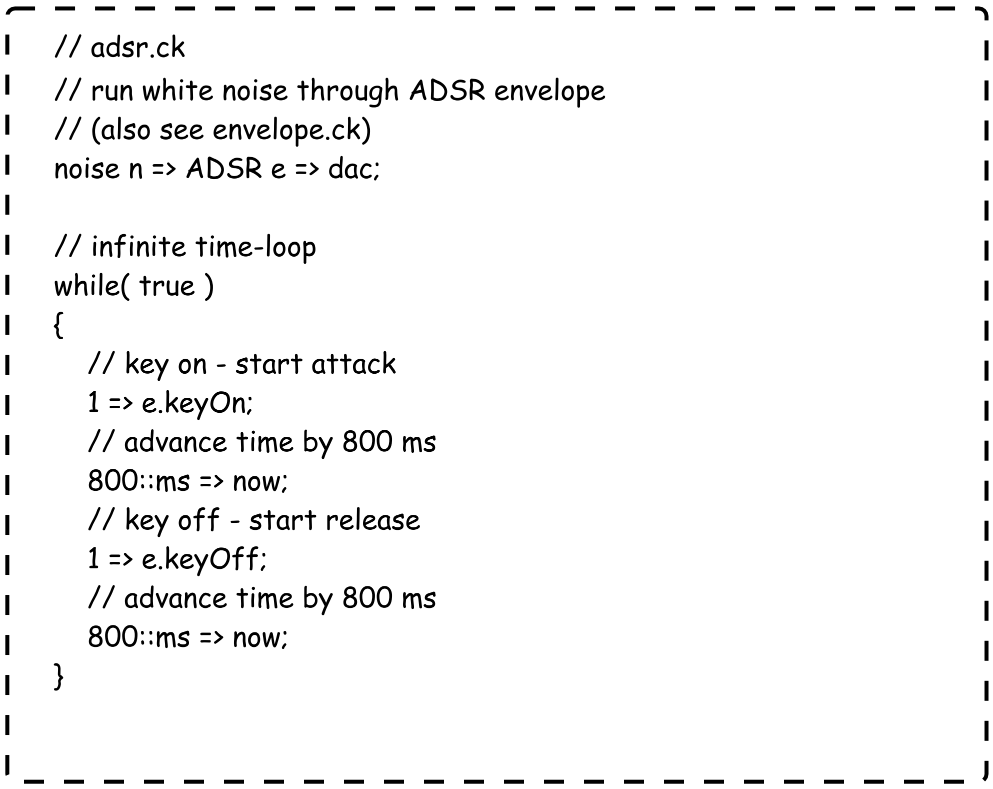
ChucK is a C-like interpreted language for audio synthesis developed at Princeton University [Wang et al., 2004]. The authors describe it as a 'concurrent, strongly-timed audio programming language for real-time synthesis, composition, and performance'. Historically, the popularity of programming languages has been greatly increased by providing syntax and semantics familiar to existing popular languages. C++, for example, was a language based on Simula that gained a huge user base partly because of its C-programmer friendly features. ChucK's C-programmer friendly syntax and reuse of commonly used C programming language constructs are attractive to programmers familiar with C and C++[2] who are looking for an interpreted audio-synthesis language.

The ChucK language introduces the => (chuck) operator. The operator is overloaded to support many different semantics. It can be understood as a message invocation operator on objects, thus making the use of the ChucK operator somewhat similar to the act of sending messages in other languages supporting dynamic-binding like SmallTalk (and hence also SuperCollider and CapyTalk), Objective C, Python, Ruby and Lisp. The => operator also seems to suggest a connection, and ChucK-ing one object to another may in many instances cause such a topological relationship to be formed between the objects. Figure 2.7 is an example of a ChucK script. Along the lines of the architecture of SuperCollider and Kyma systems, the ChucK language is separated from the ChucK Virtual Machine (VM) which is much like the 'server' component of SuperCollider 3 (SC-3 Server). The language is also supported by an integrated development environment called the Audicle. The Audicle is an editor, visualizer and compiler-invocation tool for ChucK scripts. It was written from scratch using OpenGL.

In terms of construction, a trait shared by all three systems in this family is that they were built from scratch. This is possible because these tools aspire to be 'Category B' tools. Systems like SC3 Server, the Chuck VM and the CapyBara themselves facilitate the inexpensive construction of an entire family of 'Category A' tools that integrate these sound and audio language engines with various other components. Chapter 3 describes such components designed in other domains (as well as infrastructure frameworks) that

[2]In terms of syntax alone, Java may also be considered to belong to the C family (also known as the curly bracket programming languages).

may be reused to ease the task of creating a new prototype language and environment for audio synthesis that has been tailored to serve a particular end-user context. Section 3.3 describes tools that can be used to allow the end-user herself to create these languages and environments. Such creation and customization by domain-experts is already commonplace in domains like video game design.



```
// adsr.ck
// run white noise through ADSR envelope
// (also see envelope.ck)
noise n => ADSR e => dac;

// infinite time-loop
while( true )
{
    // key on - start attack
    1 => e.keyOn;
    // advance time by 800 ms
    800::ms => now;
    // key off - start release
    1 => e.keyOff;
    // advance time by 800 ms
    800::ms => now;
}
```

Fig. 2.7 A ChucK script

2.2.4 Common Music, Nyquist, Elody, OpenMusic, HMSL/JMSL

This section completes the catalogue of dynamic interpreted music synthesis languages. The languages described in this section are functional languages, although some, like Common Music are written in an object facility implemented on top of the base functional language.

```
(defun jazz-combo (measures changes tempo scale)
  (let ((roots (new cycle :of changes))
        (ampl 1))
    (process for meas below measures
             for root = (next roots)
             if (= 0 (mod meas 12))
             set ampl = (between .5 1)
             sprout (jazz-piano scale root tempo ampl)
             sprout (jazz-cymbals tempo ampl)
             sprout (jazz-high-hat tempo ampl)
             sprout (jazz-drums tempo ampl)
             sprout (jazz-bass scale (transpose root -12)
                          tempo ampl)
             wait (rhythm 'w tempo))))

(events (list #&combo
              (jazz-combo 48 jazz-changes
                          jazz-tempo jazz-scale))
        "jazz.mid")

(events (list #&combo
              (jazz-combo 48 jazz-changes
                          jazz-tempo jazz-scale))
        "jazz.mid")
```

Fig. 2.8 A CM function that describes a jazz combo

Common Music

Common Music is a music composition language written using an object facility on top of Common Lisp or Scheme [Taube, 1991]. Figure 2.8 shows a CM function that uses other functions (not shown) to generate the sounds of a jazz combo. Common Music leverages the fact that all domain-specific languages expressed in Lisp have the advantages that come with the Lisp language and environment, namely a dynamic, extensible base language with a rich set of orthogonal operators that can be combined to express different kinds of semantics. In particular functional languages have a rich repertoire of operations for defining advanced control constructs using facilities like first-class continuations that make the control flow semantics of the base language programmable (i.e., customizable by the domain-specific language designer) [Danvy and Filinski, 1990]. Programming unit generators and filters, for example, can be implemented efficiently using vectorized operations that can benefit from the kinds of semantics supported by streams which in turn can be programmed using continuations[3].

Aura, Nyquist and Serpent

Nyquist is a functional language for audio and music programming written by Roger Dannenberg at CMU [Dannenberg, 1992]. Programming in Nyquist centers around abstractions known as ‘behaviours’ and ‘environments’. The notion of an environment is central to functional programming, and Nyquist defines a set of musically relevant variables that define the environment in which a ‘behavior’ is evaluated to produce a sound. These are: ‘warp’, ‘loud’, ‘transpose’, ‘sustain’, ‘start’, ‘stop’, ‘control-srate’ and ‘sample-srate’. Figure 2.9 shows the usage of some Nyquist behaviors.

Aura is an object-oriented framework for event-processing implemented as a C++ library written by Dannenberg and Brandt [1996]. Its markable features include a dual-model that supports both an object-oriented and a functional view of the Unit-Generator (UG) and Control networks. Serpent is the scripting language for Aura [Dannenberg, 2002]. It is a dialect of Python designed for Aura. It has the added advantage of a responsive, real-time garbage collector tuned to the needs of audio applications. Garbage collectors generally get scheduled “when needed”, and this can cause unpredictable delays that are unacceptable in real-time applications. Real-time garbage collectors address this need. Aura and Serpent also have GUI interfaces developed using wxWindows that allow for graphical patch builders and other control interfaces to be built[4].

Elody

One of the most innovative of all the functional languages developed for music, Elody is described by the authors ([Fober et al., 1997]) as a “music composition environment based on a visual functional programming language, a direct manipulation user interface and internet facilities”.

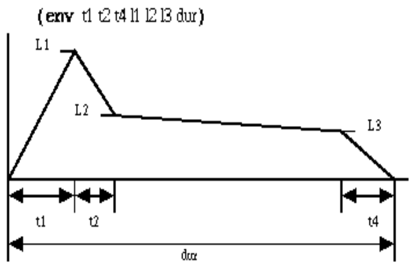
[3]Elsewhere, however, and especially so in AI, complex control structures have been criticized in recent times, and simple message passing has been favored over them [Agha and Hewitt, 1987]. This is also referred to as the ‘hairy control structure’ problem [Sussman and McDermott, 1972].

[4]The current work includes the implementation of an object-oriented event-processing framework implemented by extending STK (Section 2.2.5) and a simple reference-counting garbage collector for memory management. The OO messaging facility can itself be used to implement functional idioms on top of it.

①

```
(play (seq (note c4 i)
           (note d4 i)
           (note f4 i)
           (note g4 i)
           (note d4 q)))
```

②



```
; env-note produces an enveloped note.  The duration
;  defaults to 1.0, but stretch can be used to change
;  the duration.
;
(defun env-note (p)
  (mult (note p 1.0)
(env 0.05 0.1 0.5 1.0 0.5 0.4)))
```

③

```
play (scale 0.5 (sim (note c4 q) (note d4 i))))
```

④

```
(defun snds (dly)
  (sim (at 0.0 (cue a-snd))
       (at 0.7 (cue a-snd))
       (at 1.0 (cue a-snd))
       (at (+ 1.2 dly) (cue a-snd))))
```

Fig. 2.9 Nyquist functions. 1)Sequential Behavior 2)Envelope using a Transformation 3)Simultaneous Behavior 4)Nested Behavior

The alternative approach realized in Elody involves extending music notation systems to also double as general-purpose programming languages. It is thus a special case of a Category B framework in the guise of a Category A tool. It provides a general-purpose extensibility framework based on the lambda calculus, for the purpose of extending the domain-specific language of musical notation.

Fober et al. [1997] refer to this approach as 'homogeneous programming', referring to the fact that both domain expertise (musical knowledge) as well as extensibility expertise (software design and system modeling) are captured using the same extended version of musical notation developed by them. This allows for end-users with little knowledge of general-purpose programming languages to extend the system by using the music-notation like formalism that is provided with Elody. As the authors describe it, "Elody propose (*sic*) lambda-abstraction on musical structures as a fundamental mechanism to represent user-defined musical concepts and compositional operations.” These abstractions also have visual representations as shown in Figure 2.10, providing for intuitive discovery, by end-users, of programming-language like facilities in the composition notation.

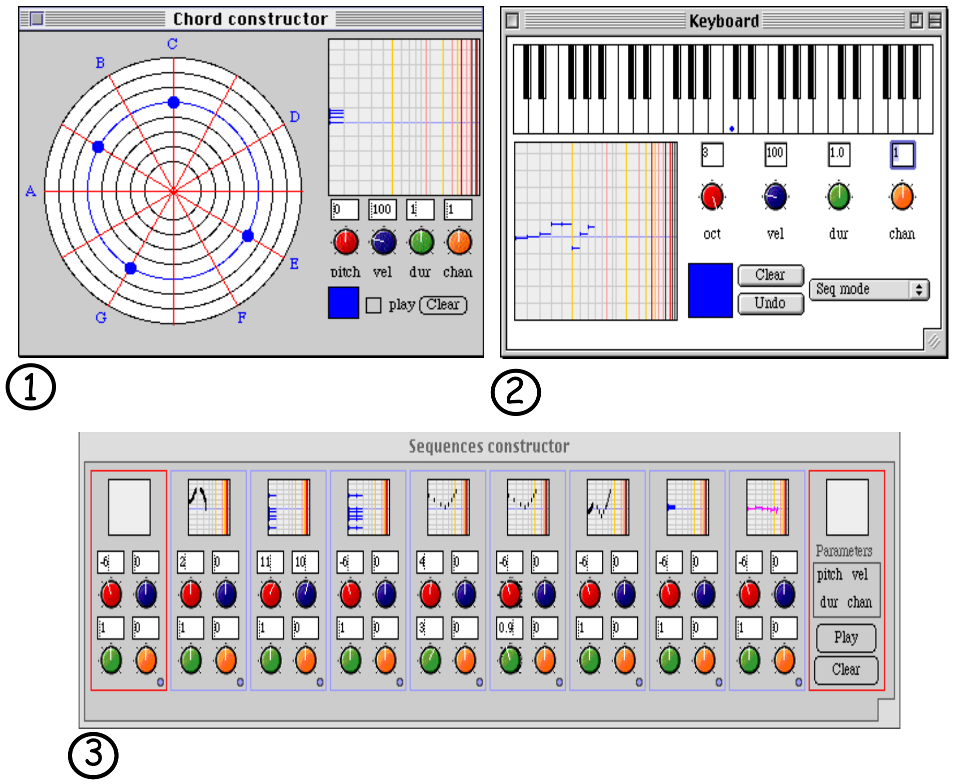


Fig. 2.10 Visual Constructors in Elody for (1) Chords (2) Keyboard Objects (3) Sequences

Elody objects can be created by end-users using a facility known as a ‘Visual Constructor’. Visual Constructors are widgets for creating new expressions that are atomic expressions or composites using combinations of existing expressions. It is worth noting that Elody itself is built on top of Grame’s MidiShare [Grame, 2000], which was designed to be a “real time operating system for musical applications” [Fober, 1994]. It is likely that consumer operating systems like Mac OS X and Windows will assimilate such functionality into future versions[5].

HMSL/JMSL

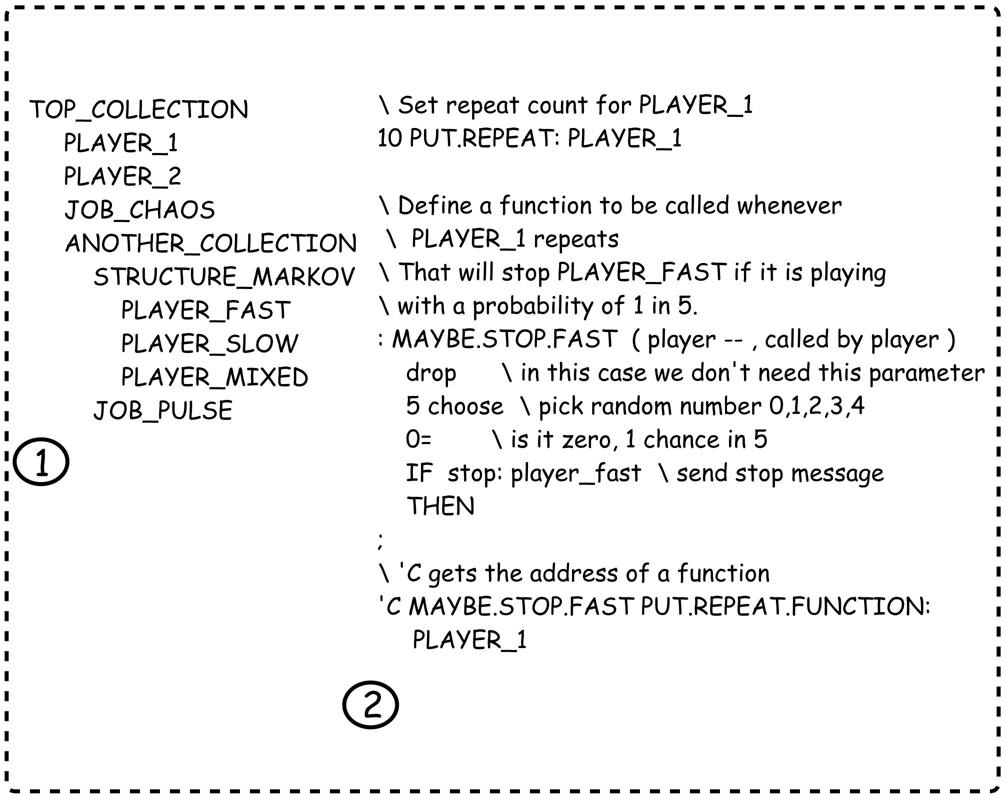
HMSL (Hierarchical Music Specification Language) is a superset of the interactive, incrementally compiled/interpreted language Forth. It was developed by Phil Burk, Larry Polansky and David Rosenboom at Mills College Center for Conteporary Music in 1980, and has been extended and maintained since then [Rosenboom and Polansky, 1985]. JMSL is a more recent, Java version of HMSL that is better maintained. Being a superset of Forth, HMSL allows user defined extensions to the language and HMSL itself is realized as a Forth extension. It is thus a Category B framework realized using another Category B infrastructure framework.

HMSL introduces the concept of a musical 'shape'. A 'shape' models an abstract musical structure like melody, a profile of harmonic complexity or other user defined structure/object. A hierarchical collection of these structures can be created and scheduled for execution. Elements in the collection can be executed consecutively or simultaneously. Hierarchy combined with the concurrent and consecutive scheduling semantics provide for the description of a very large variety of musical processes. Figure 2.11 shows HMSL scripts. HMSL also has a MIDI toolbox and a GUI toolkit for programming user interaction.

As with all the other tools the components of HMSL are built mostly from scratch. However, it reuses the Forth compiler/interpreter to implement the programming language itself. This is in the spirit of the design methodology advocated by the current work. Current technology also makes it possible to build visual and textual components into a seamlessly extensible language framework as described in Section 4.3.2.

JMSL is a more recent, Java rewrite inspired by HMSL [Didkovsky, 2004]. Unlike HMSL, which is written as an extension to Forth, JMSL is implemented as a Java library. Although Java is not an extensible language environment like Forth, a simpler form of extensibility is provided by allowing the user to extend the Java API provided by JMSL. The use of Java, on the other hand, makes it easier to deploy JMSL applications over the web. JMSL reuses

[5]This is also the case with video game consoles from product lines like Nintendo, PlayStation and XBox which often evolve independent infrastructure frameworks



```
TOP_COLLECTION
    PLAYER_1
    PLAYER_2
    JOB_CHAOS
    ANOTHER_COLLECTION
        STRUCTURE_MARKOV
            PLAYER_FAST
            PLAYER_SLOW
            PLAYER_MIXED
        JOB_PULSE
```

```
\ Set repeat count for PLAYER_1
10 PUT.REPEAT: PLAYER_1

\ Define a function to be called whenever
\  PLAYER_1 repeats
\ That will stop PLAYER_FAST if it is playing
\ with a probability of 1 in 5.
: MAYBE.STOP.FAST  ( player -- , called by player )
    drop      \ in this case we don't need this parameter
    5 choose  \ pick random number 0,1,2,3,4
    0=        \ is it zero, 1 chance in 5
    IF  stop: player_fast  \ send stop message
    THEN
;
\ 'C gets the address of a function
'C MAYBE.STOP.FAST PUT.REPEAT.FUNCTION:
    PLAYER_1
```

Fig. 2.11 An HMSL hierarchy (1) and functions to dynamically call the hierarchy (2).

the JSyn library [Burke, 1998] for synthesis. A JSyn plugin is available for integration with browsers to run JMSL applets over the web.

JMSL also uses the rich GUI facilities provided with Java applications and applets to implement various graphical object editors, for example, ‘transcribers’ to notate algorithmically generated music. It also provides a plug-in or extensibility API that allows for extensions to be build to these graphical editors. These graphical editors are thus Category A components, and JMSL itself is a Category B tool with which they are built. This makes for a good combination. Traditional musical scores produced using the Transcriber can also be exported to Finale®, a popular commercially available music-notation based authoring system.

OpenMusic

OpenMusic is an object-oriented/functional Lisp system that has a visual interface to the Common Lisp Meta Object Protocol (MOP) and a constraint programming system for music [Assayag et al., 1997; Agon et al., 1998; Assayag et al., 1999a;b; Truchet et al., 2001; Bresson et al., 2005; Assayag et al., 2000a;b]. Figure 2.12 shows a composition

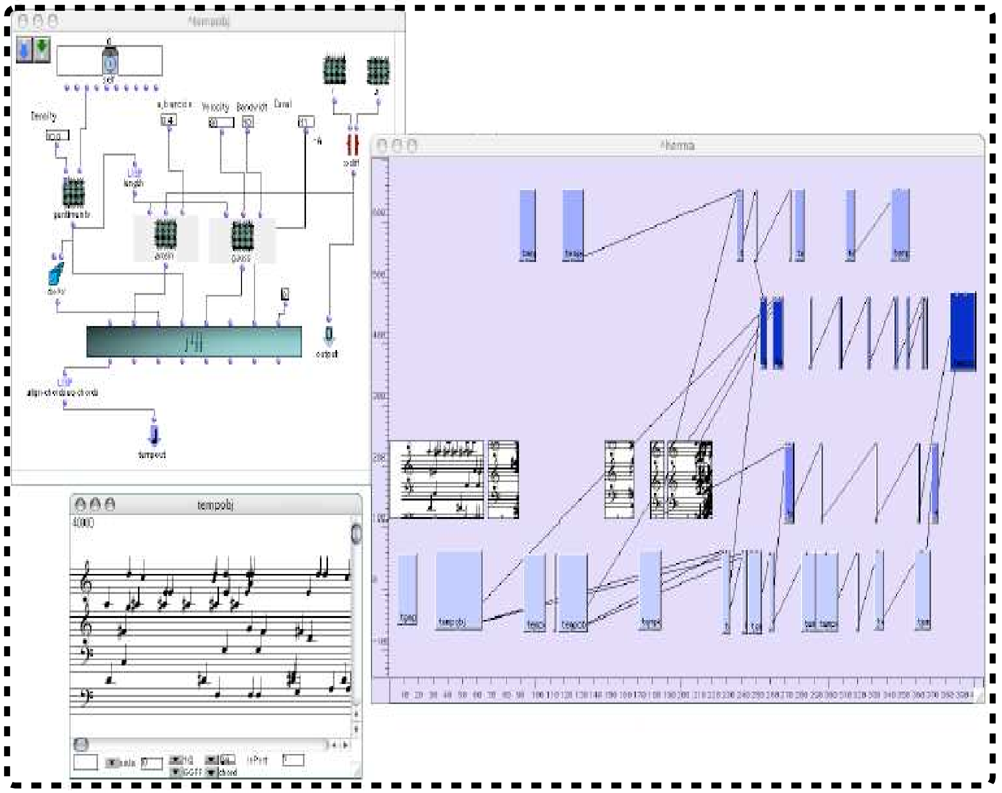


Fig. 2.12 A composition by Iannis Xenakis (Herma) opened in OpenMusic

created in OpenMusic. OpenMusic, being a successor to Elody, bears some resemblance to Elody. Similarly to Elody, musical S-Expressions can be created using visual constructors. Additionally, OpenMusic also provides access to the object-oriented features of the Common Lisp Object System (CLOS) as well as visual metaprogramming via the Common Lisp Meta Object Protocol. This is similar to extending and customizing a Forth system as done in HMSL. Using the MOP allows customization and extension of the semantics of the object model used by CLOS itself, and is a very powerful way to create domain-specific abstractions[6]. Section 3.3 visits current advances in visual meta programming and meta modeling systems. Chapter 4 documents the results of design experiments conducted to explore the possibility of reusing components resulting from these advances in building music authoring systems.

2.2.5 STK, CLAM

The synthesis components described in this section are all distributed as libraries. They do not define their own programming language. Instead, they may be called from applications written in general-purpose programming languages.

2.2.6 Synthesis ToolKit (STK)

The Synthesis ToolKit is a C++ library for audio signal processing and algorithmic synthesis designed and implemented by Perry Cook and Gary Scavone [Cook and Scavone, 1999]. STK is highly portable and builds under a wide variety of operating systems, including realtime support on SGI, Linux, Mac OS X and Windows. The design of STK leverages the object-oriented facilities of the C++ Programming Language. It is both a library of reusable synthesis and signal processing components, as well as a framework within which to design new synthesis and signal processing elements. Figure 2.13 shows a portion of the STK Class Hierarchy. Using STK as a library involves writing a C++ program like the one showed in Figure 2.14. The code generates a sinewave and writes it to a wav file. STK may also be used as a framework to build new synthesis classes. This involves 'inherit'ing a class from one of the base classes in the hierarchy and implementing customized functionality to produce a new type in the hierarchy.

STK is particularly popular because of the ease with which it may be used to build C++ applications. C++ has been a first choice for many years for building media-rich and computationally intensive applications and STK addresses the need for a synthesis infrastructure framework for C++ programmers. In recent times, advances in processing

[6]As also a powerful way to shoot oneself in the foot. Creating useful and meaningful new object models and semantics is not for every end-user. However, 'power'-users may leverage such facilities for advanced customization. When using Category B frameworks like OpenMusic to realize Category A tools and systems, it is thus advisable, from a usability perspective, to expose these facilities in a manner that does not obfuscate simple usage scenarios. The Perl design philosophy of 'Make easy things easy and difficult things possible' is especially relevant in this context.

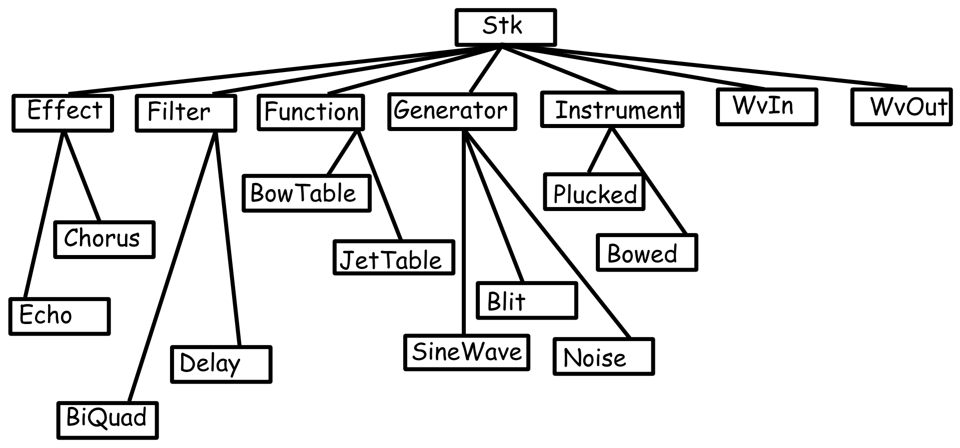


Fig. 2.13 Portion of the STK Class Hierarchy.

power, dedicated sound processing hardware, as well as advances in optimized compilers and garbage collectors has made garbage collected languages like Java and C# and dynamic languages like Lisp, Python, Perl, Ruby and SmallTalk also viable vehicles for designing audio synthesis libraries. Often times, however, the computationally intensive portions of such libraries are also written as C/C++ externals[7]. Additionally, STK implements innovative methods for synthesis like Physical Modeling [Smith, 1996] that may be reused simply by instantiating the relevant STK class.

2.2.7 CLAM

CLAM is a C++ Library for Audio and Music [Amatriain et al., 2002]. It implements the core technology used by a tool suite and platform for audio computing developed at the Universitat Pompeu Fabra [Arum and Amatriain, 2005]. The functionality offered by the library is similar to STK. However, CLAM includes functionality both for analysis and for synthesis of audio. The features of the library that are highlighted by the authors are the quality of the C++ code, which follows current best practices advocated by the C++ community, the efficiency of the implementation, the comprehensiveness of the spectrum of functionality integrated into the library, tool suite and the extensibility of the framework. While these qualities are not particularly relevant to end-users of the Category A tools

[7]The current work includes design experiments that explore the possibility of combining a synthesis server implemented in STK with dynamic components written in other languages.

```
// sineosc.cpp

#include "WaveLoop.h"
#include "FileWvOut.h"

int main()
{
  // Set the global sample rate before creating class instances.
  Stk::setSampleRate( 44100.0 );

  WaveLoop input;
  FileWvOut output;

  // Load the sine wave file.
  input.openFile( "rawwaves/sinewave.raw", true );

  // Open a 16-bit, one-channel WAV formatted output file
  output.openFile( "hellosine.wav", 1, FileWrite::FILE_WAV, Stk::STK_SINT16 );

  input.setFrequency( 440.0 );

  // Run the oscillator for 40000 samples, writing to the output file
  for ( int i=0; i<40000; i++ )
    output.tick( input.tick() );

  return 0;
}
```

Fig. 2.14 A C++ program that uses STK.

developed using CLAM, they are directly relevant to the users that develop tools using CLAM. Increasingly, Category A tools that lie in the intersection are providing customization and extensibility features to the end-user, and this involves providing a simplified access to the internals of the tool, which effectively allows domain experts like musicians and sounds designers to program the tool using the underlying Category B functionality of libraries like CLAM and STK.

Good design of the Category B framework itself will thus affect the end-user programmability of the Category A tool that exposes its internals. The core design entities in CLAM have been formally represented as a metamodel expressed in UML [Amatrian, 2005]. Such formal representation of the internal design of a framework facilitates the construction of automatic introspection and reflection tools.

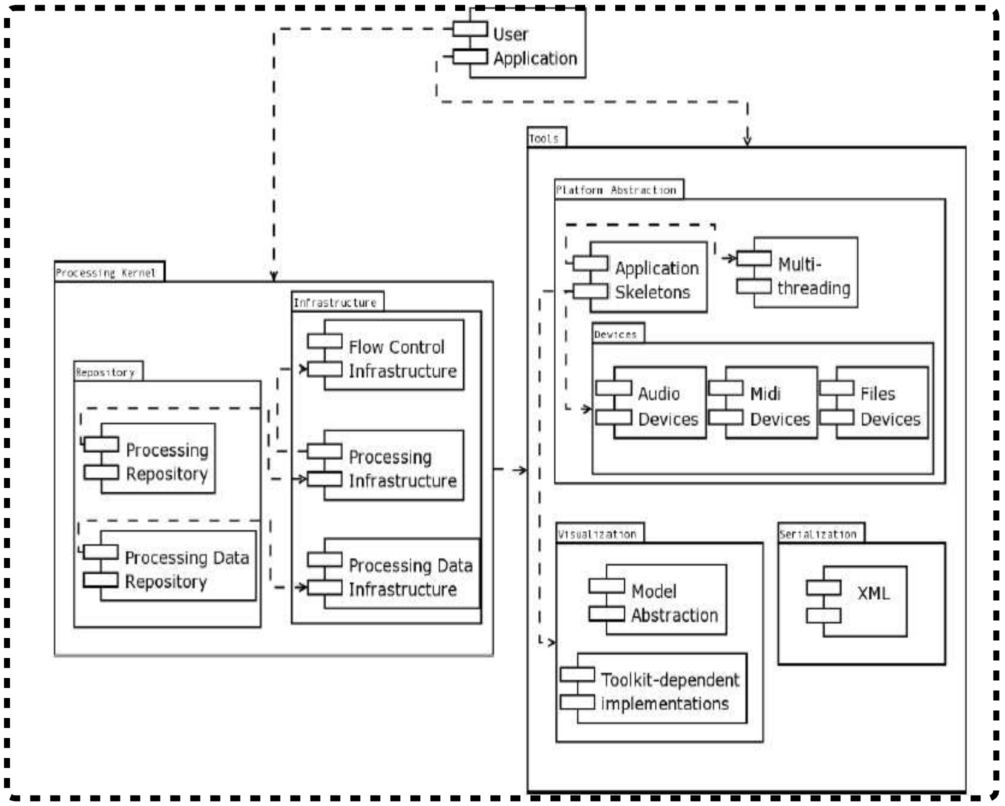


Fig. 2.15 CLAM modules [Arum and Amatriain, 2005].

Figure 2.15 shows the organization of CLAM modules. Besides the infrastructure framework that CLAM implements in the form of a library, it also provides rapid-prototyping tools in the form of graphical tools like the Network Editor and the Prototyper. Developing an application with CLAM involves defining the processing network using an application called the Network Editor and defining the UI and interaction using an application called the Prototyper.

Figure 2.16 shows a processing network being edited with the Network Editor and an application screen being edited using the prototyper. The Prototyper itself has been built by customizing a Category A tool bundled along with the QT library called QT Designer. It is a number of extensions, additions and customizations to QT Designer for CLAM specific applications. This is in the spirit of the methodology for authoring tool design recommended by the current work. Section 3.3 presents advanced customizable modeling environments

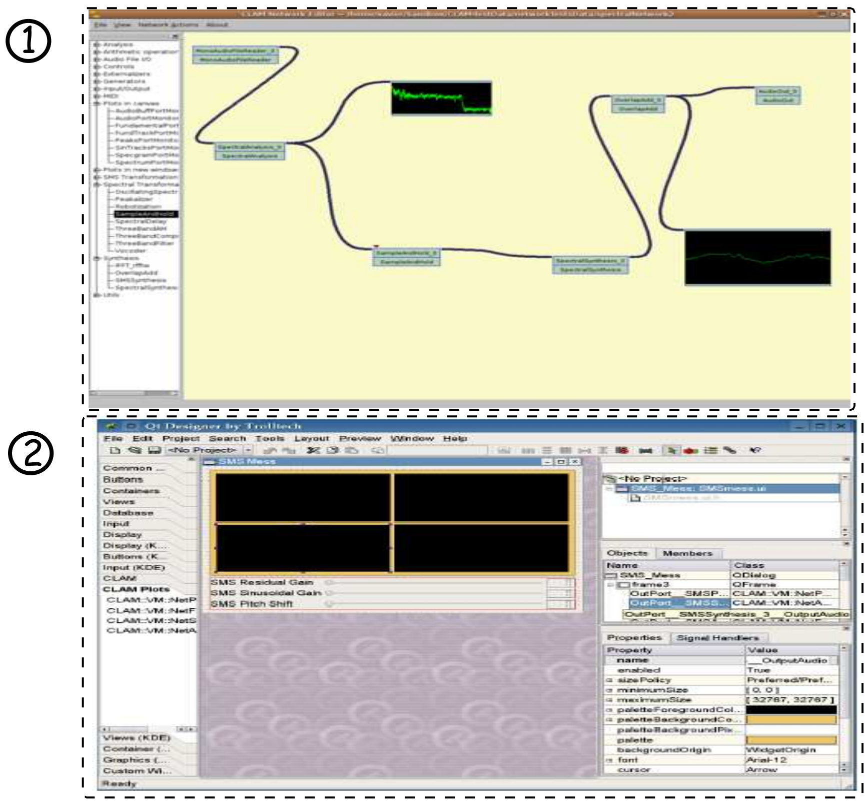


Fig. 2.16 CLAM Network Editor (1) and CLAM Prototyper (2) in action.

that can be used to generate applications like the Network Editor automatically, from UML specifications.

2.3 Studio Tools/Recording and Sequencing Software

Section 2.2 explored languages and frameworks for describing, representing and realizing the structure of processes that generate sound and the structure of processes that generate music. This section explores tools that have been developed in the context of the more conventional roles of computers in the music industry, that of the computer as a recording and editing machine in a studio setting.

Also referred to as Digital Audio Workstations [Wikipedia, 2006c], the tools discussed in this section have somewhat similar functionality, though there are markable differences in the quality and performance of the tools, their suitability for particular tasks, and the number of features offered. Features offered by these software include:

1. The ability to record sound.
2. The ability to transform recorded sound using a comprehensive set of signal processing blocks.
3. The ability to simulate acoustic instruments and analog electronics (instruments, amplifiers and processors) in software.
4. The ability to sequence recorded sound as well as control events.

2.3.1 Audacity, Apple Logic, PropellorHead Reason, DigiDesign ProTools, Steinberg Cubase, TwelveTone Systems CakeWalk Pro

Apple Logic Pro

Logic Pro aims to be a 'complete music studio' and offers a wide range of functionality, including the ability to record sounds, sequence recorded sound (as well as control events in the form of MIDI/OSC messages), transform recorded sound with a comprehensive set of signal processing blocks, and modeling virtual instruments (both pre-created and user build) in software [Wikipedia, 2006g]. Figure 2.17 shows screenshots from Apple's Logic Pro software. The Sculpture interface (lower box) offers the user a wide variety of parameterized models of musical instruments and components of musical instruments (like strings, strikeable surfaces, air columns and so on). This is effectively a parameterized 'CAD' for sound design, analogous to the Category A Bicycle CAD application described in Section 1.1. The underlying models may be complex though, as the Logic Pro online documentation hints:

> The sophisticated algorithm at the core of Sculpture combines different models of vibrating natural material: It can be glass, steel, nylon or wood or a mixture of all of them. You can even morph between them, starting with a nylon string in the attack phase of the sound and then decaying to the wood or metal bar of a xylophone. A second element of the algorithm describes the way the vibration is initiated. This means that you have the choice between a vibration that is bowed, plucked or blown, or variations of these. You can also determine where the pick-up is positioned.

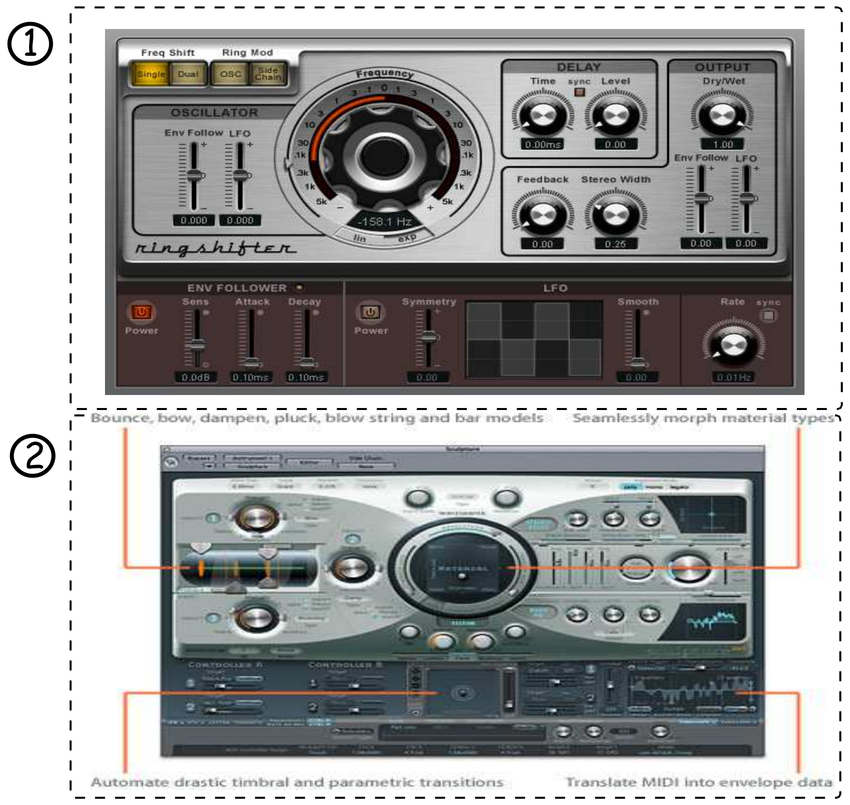


Fig. 2.17 Logic Pro's Sculpture (1) and Ring Shifter (2) tools.

All of the functionality available in Logic is neither scriptable nor re-usable in standalone applications independant of the Logic Pro product. It is noteworthy that Garage Band, another music authoring and production tool from Apple[8], also uses the Logic Pro Audio Engine [Wikipedia, 2006g]. Going by the trends in the 3D animation industry[9]the economics suggest that the development of a reusable, professional-quality, commercially supported (or community supported) audio engine would be very valuable. Such an engine would constitute a Category B framework that could be used to build a whole family of interoperable Category A tools for music authoring, as well as stand alone Category A applications (that do not belong in the intersection A∩B).

[8]Targetted primarily at amateur musicians.

[9]The classic examples in this category are the 'Quake' and 'Unreal' 3D engines, which were originally designed specifically for the games with the same name. Eventually, both engines were made available for use in other games and applications [Wikipedia, 2006f].

Propellerhead Reason

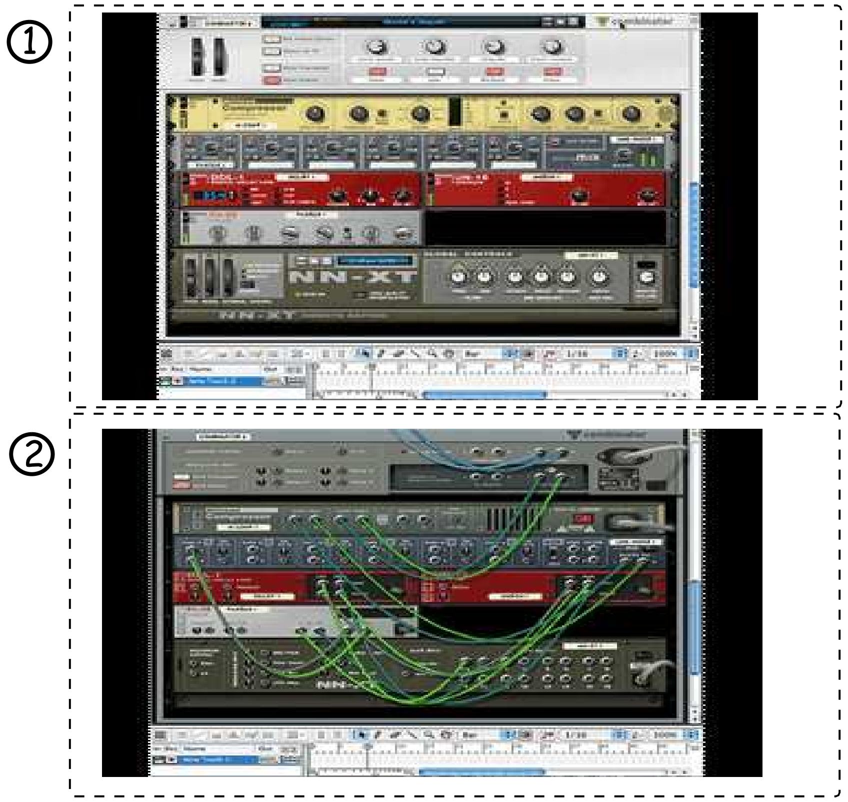


Fig. 2.18 (1) Reason's Combinator and (2) 'Rear View' with a realistic patch cord interface.

Figure 2.18 shows screenshots from the Reason software. Reason is notably different in that it cannot to be used to record audio, or be expanded using third-party plugins. The main feature offered by Reason is simulation of synthesis hardware in software, and the ability to plug many of these modules together in an intuitive patch cord interface. Reason takes the patch cord interface to new literal heights with a realistic rendering of cords as shown in the 'Rear View' Figure 2.18 (lower box). The upper box shows the regular view of the Combinator, exposing the parameters and controls of various modules that have been combined. It is a good example of an intuitive end-user programming interface. Section 3.1 discusses currently available GUI frameworks that make it possible to provide facilities for designers to add new professional-quality intuitive interfaces that leverage the state of the art in 2D and 3D graphical user interfaces.

DigiDesign's ProTools, Steinberg's Cubase and TwelveTone Systems' CakeWalk Pro provide somewhat similar functionality to Logic and Reason, though they have their individual strengths and weaknesses. GarageBand and FruityLoops are similar tools targeted at amateur users.

2.4 Plugin Application Programmer's Interfaces (APIs)

These APIs were originally designed in the context of extension module Software Development Kits (SDKs) for effects processing in recording software (see Section 2.3). For this reason, they are not specific to the problem of authoring tools. They provide an API to describe new unit-generators and effects-processing algorithms. However, such APIs could be re-used in the context of extension-APIs for an authoring tool. In particular, a subset of the synthesizer extension API could support VST and LADSPA plugins to reuse existing components written for these APIs.

2.4.1 Steinberg VST

The VST plug-in API was developed in relation to Steinbergs Cubase software. It is among the most popular effects and synthesizer plug-in APIs. The VST SDK documentation describes it thus:

> In the widest possible sense a VST-Plug-in is an audio process. A VST Plug-in is not an application. It needs a host application that handles the audio streams and makes use of the process the VST plug-in supplies. Generally speaking, it can take a stream of audio data, apply a process to the audio and send the result back the host application. [...]: The host does not maintain any information about what the plug-in did with the last block of data it processed. From the host application's point of view, a VST Plug-In is a black box with an arbitrary number of inputs, outputs, and associated parameters. The host needs no knowledge of the plug-in process to be able to use it.

The control structure of VST is known to be limited [Zbyszynski and Freed, 2005]. However, it is well suited to describe effects plug-ins as well as as a subset of software synthesizer plug-ins.

2.4.2 Linux Audio Developers Simple Plugin API (LADSPA) and DSSI (Disposable Soft Synth Interface)

LADSPA developed primarily as an open alternative to VST and DirectX plug-ins on the Linux platform. The effects/filters plug-in funtionality that LADSPA aims to standardize is very similar to VST and DirectX, and it brings these benefits to linux audio developers. It has similar drawbacks. Development on LADSPA has been slow, as noted in the LADSPA home page [Furse, 2006a]. LADSPA has tried to address the lack of a GUI standard for plug-ins by coming up with an XML based GUI Specification [Furse, 2006b]. Along similar lines, Chapter 4 describes design experiments that demonstrate the manner in which an advanced XML based GUI specification (XAML) could be used to specify visual interaction with authoring tool extensions.

2.5 Next Generation Music Software

This section presents music software that are in some sense, substantially different from the software presented in the previous two sections. Both these pieces of software are distinguished by the fact that they go above and beyond simple musical tasks and provide the user with a palette of functionality that can be used to achieve complex musical tasks. Cypher is a Category B framework that belongs in the intersection and HyperScore is a Category A tool that does not belong in the intersection.

2.5.1 Cypher

Cypher is a C++ library of classes and algorithms for the Macintosh (also usable as Max externals), that are a realization of an approach to interactive music systems proposed by Rowe [Rowe, 1991]. It is also a complete real-time interactive music system. Rowe's work is inspired by Marvin Minsky's ideas from Society of Mind [Minsky, 1986]. The implementation of Cypher is based on a societal architecture, in which small, relatively independent agents co-operate to realize complex behaviors. In particular, the compositional part of Cypher's design centers around such independent musical agents, known as 'compositional critics'. These critics are processes that evaluate the output of a segment of composition before it is sounded. The segment is then modified to conform to the aesthetic criteria that the critic defines.

In his book Machine Musicianship [Rowe, 2001], Rowe explains the idea thus:

> Musicians begin formal training by acquiring a body of musical concepts commonly known as musicianship. These concepts underlie the musical skills of listening, performance, and composition. Like humans, computer music programs can benefit from a systematic foundation of musical knowledge ... The resulting applications can be used to accomplish tasks ranging from the solution of simple musical problems to the live performance of interactive compositions and the design of musically responsive installations and web sites.

Cypher builds abstractions for musical processes on top of sequences of MIDI[10] events. However, it goes above and beyond the paradigm of viewing music as a combination of sound design and sequencing[11]. The musical processes modeled by Cypher's compositional critics embody a rule-based representation of aesthetic principles that enables it to respond to high level features in the incoming musical stream in intelligent ways. Figure 2.19 shows a portion of the implementation of a high-level listener object in Cypher. Cypher's listening and interpretation engine could be reused in intelligent authoring tools that provide the composer with hints computed by the 'compositional critic' processes.

2.5.2 HyperScore

Like Cypher, HyperScore has its roots in the Hyperinstruments group at the MIT Media Lab. Originally designed by Mary Farbood as part of her doctoral work [Farbood et al., 2004], it was eventually made into commercial software, sold through Harmony Line Music. HyperScore is designed as an intuitive, easy-to-use visual environment along the lines of a sketching program. It is intended to teach children and amateurs the basics of composition through first-hand exploration. Figures 2.20 and 2.21 show screen shots from the software.

The basic visual metaphor in HyperScore is that of the Harmony Line (box 7). It consists of a bendable line that is associated with musical objects. The association between the

[10]Rowe acknowledges the limitations of MIDI, and subsequent work has focused on what he terms 'Personal Effects' or a library of sound and compositional building blocks that computer-based music artists construct, effectively creating a personalized instrument [Rowe, 2005]. This concurs with the views of Tanaka described in Section 2.1.1 as well as the approach proposed in the current work.

[11]Here, and throughout this thesis, the term 'sequencing' is used in a general sense, to mean the production or transformation of any sequence that eventually controls an audio process.

```
void TriadListener::MakeUpEvent(void)
{
 int pcs[3]    = { 0, 0, 0 };
 int chordSize = 3;

 Incoming = Incoming->Next();
 Incoming->SetChordSize(chordSize);

 while ((pcs[0] == pcs[1]) || (pcs[0] == pcs[2]) || (pcs[1] == pcs[2]))
   FillTriad(pcs);
 for (int i=0; i<chordSize; i++) {
  register Note *n = Incoming->Notes(i);
  n->SetPitch(pcs[i]+60);
  n->SetVelocity(60);            // with a constant velocity
  n->SetDuration(1000L);         // and a constant duration
 }
 long here  = gClock->currentTime;
 Incoming->SetTime(here);
 Incoming->SetIOI(here - lastAttack);
 lastAttack = here;
 Process();
}
```

Fig. 2.19 Portion of the implementation of the 'TriadListener' class in Cypher.

musical object and the line is made when the object is created. The creation of the object itself is similar to Elody and OpenMusic, with the aid of 'Visual Constructors' (boxes 1,2). The expression of Looping, Transposition and Sequencing are all thus subsumed within the language of distinct, bendable lines. This is an intuitive new visual syntax for the domain of sequencing that is different from conventional data-flow, object-oriented and functional notations. The semantics of the language is that of highly simplified object instantiation, composition and repetition, and it is not a Turing-complete visual language like Elody or OpenMusic. This suggests that HyperScore is intended to be a Category A tool, which does not belong in the intersection. Mark Zadel has developed a similar line-based interface for laptop performance [Zadel, 2006; Zadel and Scavone, 2006].

It is useful to formalize the process of interpreting descriptions represented by 'sentences' in such generalized visual languages. [Costagliola et al., 2002] have developed a classifica-

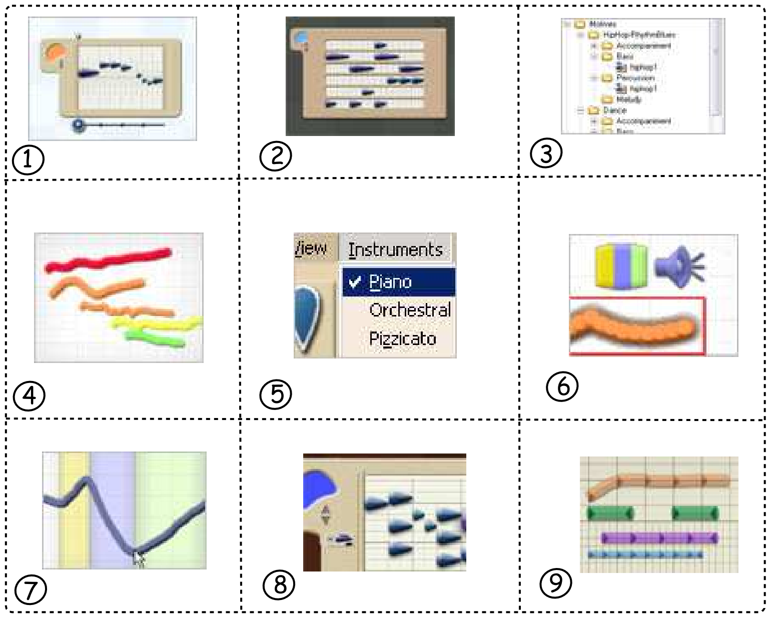


Fig. 2.20 Visual Tools in HyperScore - (1)Melody Window (2)Percussion Window (3)Music Library (4)Sketch Window (5)Instrument Sounds (6)Volume and Tempo Control (7)Harmony Line (8)Polyphonic Mode (9)Motive Loop Visual Feedback

tion framework for visual languages that can be used to design an automatic interpreter or compiler generator for such visual descriptions. Section 3.1 and Section 3.3 present frameworks that may be combined to provide an end-user extensible visual-language interpreter facility for applications with generalized, interactive, visual languages with static and dynamic, 2D/3D elements supportable in the language's concrete syntax, and innovative new semantics for expressing musical structure[12].

[12]These sections do not propose a specific new syntax or semantics as being valuable, though. They only describe the tools and frameworks that may be used to ease the process of realizing tools that provide such features.

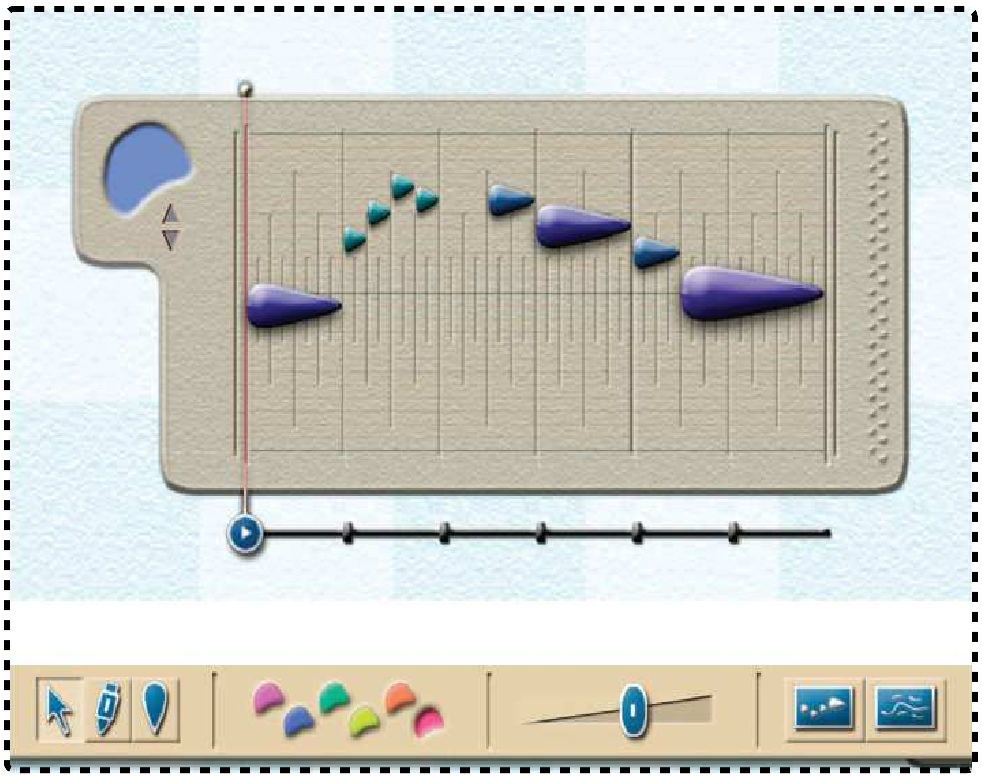


Fig. 2.21 Closeup of HyperScore's Motive constructor.

Chapter 3

Reusable Components from Other Domains

This chapter surveys the infrastructure frameworks introduced in Section 2.1.1 in greater detail. The functionality provided by these frameworks can be classified as belonging to one of the three categories of a) Language b) User Interface and c) Extensibility. These frameworks thus offer 'support functionality' that is needed in a usable and productive authoring environment. While these do not contribute directly to the end goals of describing and realizing musical and sound structures, they are as essential as the sound and music engines described in the previous chapters. The survey is conducted from a high-level design perspective. The focus here is to identify strategies with which to combine functionality from general-purpose off-the-shelf components and libraries developed to serve many domains in a way that is valuable to the goal of building music authoring tools, and to identify currently popular frameworks for realizing languages, user interfaces, and end-user extensibility. This is intended to serve as a map or field guide for persons embarking on musical-tool-design projects, and as supporting material for the design recommendations proposed in the current work.

3.1 GUI Frameworks and Authoring Tools

Graphical User Interface (GUI) toolkits are low-level infrastructure frameworks and are not, as such, relevant to the discussion. However, new-generation graphical user interfaces are increasingly providing advanced facilities that ease the implementation of visual languages and user-interface extensibility in ways that are noteworthy and relevant to the design possibilities identified by the current work. One of the goals is to identify a framework or class of frameworks for extensible syntax design that can be used to combine textual and visual elements seamlessly. This is motivated by the fact that data-flow languages have a graph-based language for describing network topologies. These are especially suited to being described with a visual language framework. However, describing event-based behavior requires a combination of visual and textual elements, like state-charts and object-oriented descriptions.

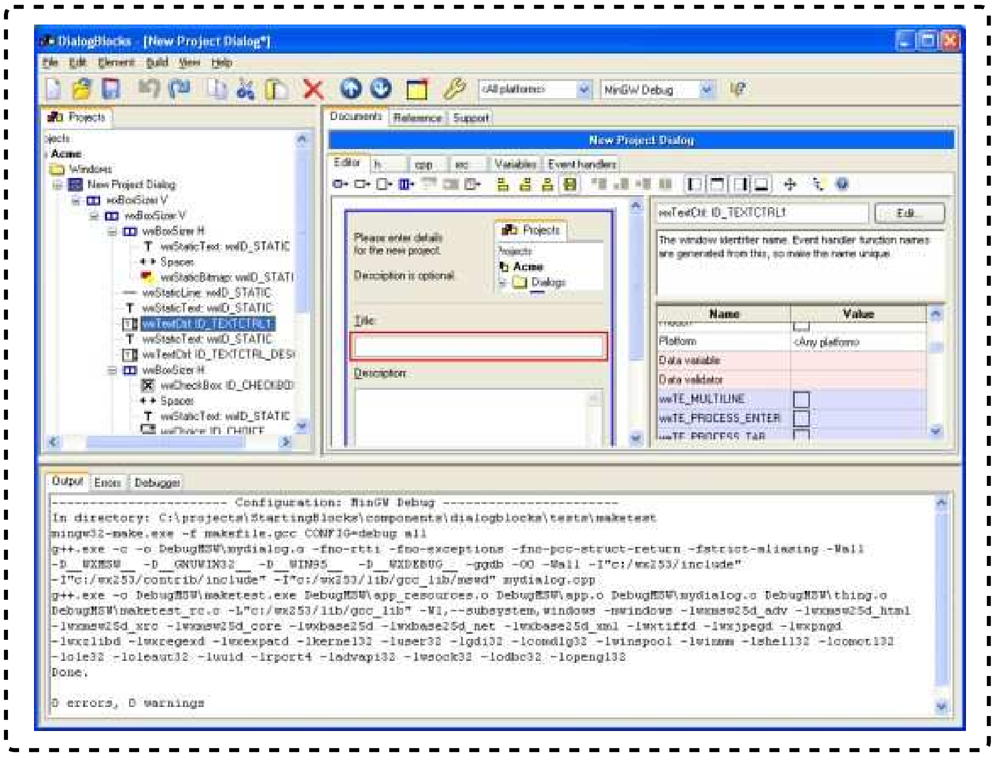
3.1.1 C++ and Java Toolkits: QT, wxWidgets, Swing, SWT

QT is a cross-platform GUI framework [TrollTech, 2006]. It is written in C++ and the native API is in C++. However, it has bindings to a number of other languages, including Python. QT implements optimized versions of its GUI using the native windowing toolkit on each platform that it supports, hence it performs as well as the native windowing toolkit on a given platform. QT introduced an innovative programming model to describe event-handlers, known as the 'signals-and-slots' mechanism. In order to implement this feature, TrollTech introduced proprietary extensions to the C++ language via macros, which is compiled using the QT Meta Object Compiler (MOC) [Wikipedia, 2006k].

wxWidgets is another cross-platform C++ GUI toolkit. It has its origins in the development of a cross-platform toolkit for the end-user programmable diagramming tool Hardy in the Artificial Intelligence Applications Institute at the University of Edinburgh[1] [Smart, 2006]. It also implements a thin wrapper around native windowing toolkits that makes it possible to write against the wxWidgets interface and compile on many different GUI platforms.

Both QT and wxWidgets implement 'designer' tools that may be used to design the GUI. This is an important consideration in choosing GUI frameworks. QT ships with the QT Designer tool, and wxWidgets can be enhanced with the installation of DialogBlocks, a What-You-See-Is-What-You-Get (WYSIWYG) dialog editor (Figure 3.1). The build process in developing a GUI using either QT or wxWidgets is similar and integrated into

[1] The StoryLines tool is another example of an authoring tool developed using wxWidgets also by the author of wxWidgets [Smart, 2006]).


DialogBlocks - [New Project Dialog*]
File Edit Element Build View Help
Projects
Documents Reference Support
New Project Dialog
Editor h cpp rc Variables Event handlers
wxTextCtrl: ID_TEXTCTRL1
Edit...
The window identifier name. Event handler function names are generated from this, so make the name unique.
Name Value
Platform <Any platform>
Data variable
Data validator
wxTE_MULTILINE
wxTE_PROCESS_ENTER
Output Errors Debugger

```
------------------------ Configuration: MinGW Debug ------------------------
In directory: C:\projects\StartingBlocks\components\dialogblocks\tests\maketest
mingw32-make.exe -f makefile.gcc CONFIG=debug all
g++.exe -c -o DebugMSW\mydialog.o -fno-rtti -fno-exceptions -fno-pcc-struct-return -fstrict-aliasing -Wall
-D__WXMSW__ -D__GNUWIN32__ -D__WIN95__ -D__WXDEBUG__ -ggdb -O0 -Wall -I"c:/wx253/include"
-I"c:/wx253/contrib/include" -I"c:/wx253/lib/gcc_lib/mswd" mydialog.cpp
g++.exe -o DebugMSW\maketest.exe DebugMSW\app_resources.o DebugMSW\app.o DebugMSW\mydialog.o DebugMSW\thing.o
DebugMSW\maketest_rc.o -L"c:/wx253/lib/gcc_lib" -Wl,--subsystem,windows -mwindows -lwxmsw25d_adv -lwxmsw25d_html
-lwxmsw25d_xrc -lwxmsw25d_core -lwxbase25d -lwxbase25d_net -lwxbase25d_xml -lwxtiffd -lwxjpegd -lwxpngd
-lwxzlibd -lwxregexd -lwxexpatd -lkernel32 -luser32 -lgdi32 -lcomdlg32 -lwinspool -lwinmm -lshell32 -lcomctl32
-lole32 -loleaut32 -luuid -lrpcrt4 -ladvapi32 -lwsock32 -lodbc32 -lopengl32
Done.

0 errors, 0 warnings
```

Fig. 3.1 DialogBlocks, a designer for wxWidgets.

the application build procedure. Both frameworks require that the user-interface components be laid out *at compile time*. This has implications for end-user extensibility, in that new visual components created using the designers cannot be integrated into a running application in a straightforward manner[2].

SWT and the Eclipse Platform

Swing and the Standard Widget Toolkit (SWT)[3] are popular GUI toolkits on the Java platform. They differ fundamentally in their implementation philosophy. Swing aims to be a pure Java implementation, with the entire toolkit implemented from scratch using Java, whereas SWT reuses native implementations to the extend that it can. The upshot of this

[2]This does not, of course, preclude the programmatic creation of new UI elements at run-time. It only limits the dynamic loading of UI functionality created by a user-interface designer or end-user via the designer tool.

[3]A check mark is placed besides all frameworks found to meet the design requirements for extensibility and reusability.

is that Swing implementations tend to be uniform on all OS platforms that have a Java implementation, whereas SWT widgets tend to have small differences in behavior that are determined by the behavior of the native widgets. One the other hand, Swing widgets tend to perform noticeably slower than the SWT widgets in computationally intensive applications [Wikipedia, 2006a].

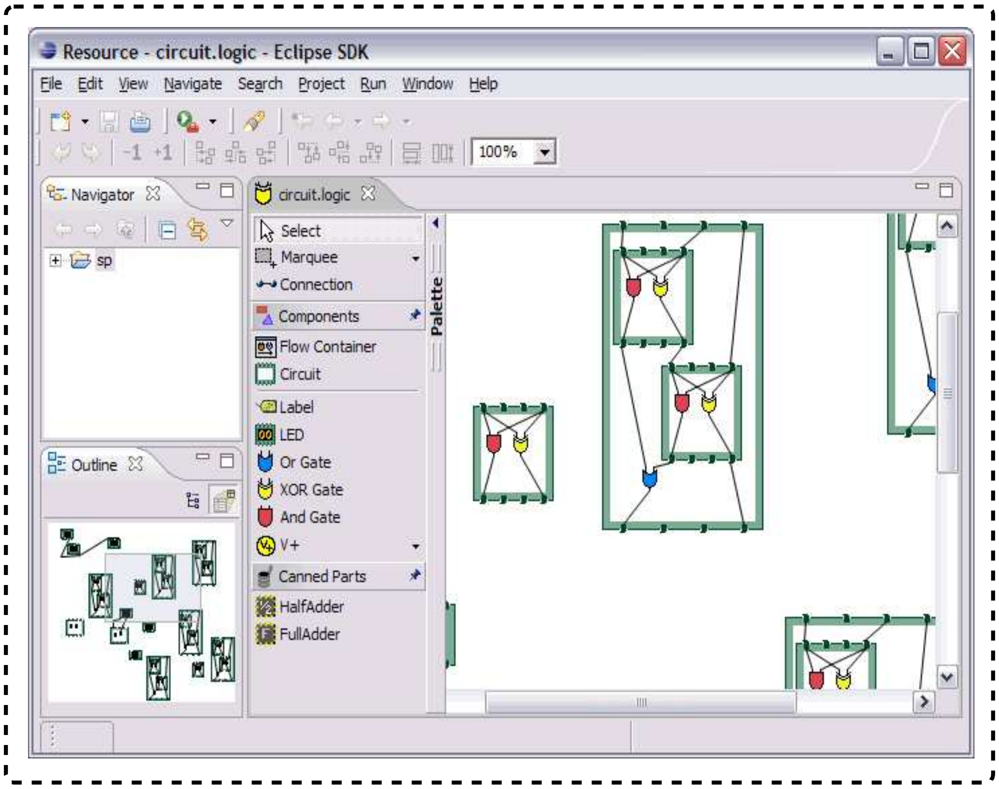


Fig. 3.2 Implementation of a Logic Circuit Editor by extending Eclipse.

SWT is part of a larger project called Eclipse, managed by the Eclipse Foundation. The Eclipse IDE and IDE workbench are very suitable for the development of extensible music authoring tools and are discussed in further detail in Section 3.3. The Eclipse platform was built with the specific purpose of being a 'universal toolset for development' [Eclipse Foundation, 2001]. It is a Category B framework (in the intersection) for language and environment design. Eclipse and SWT may be considered a high-level GUI platform for building design tools. From the perspective of the design of music authoring tools, the IDE and Language Workbench in the Eclipse platform are particularly relevant.

It is interesting to note that the Eclipse project is both a landmark in the development of reusable user interface tools, as well as in the development of extensible programming languages [Eclipse Foundation, 2001]. The toolkit is especially suitable for generating environments for visual languages like the data-flow notation used in many sound-design environments, as well as more general visual languages [Ehrig et al., 2005]. Figure 3.2 shows a Logic Circuit Editor implemented using the Eclipse platform.

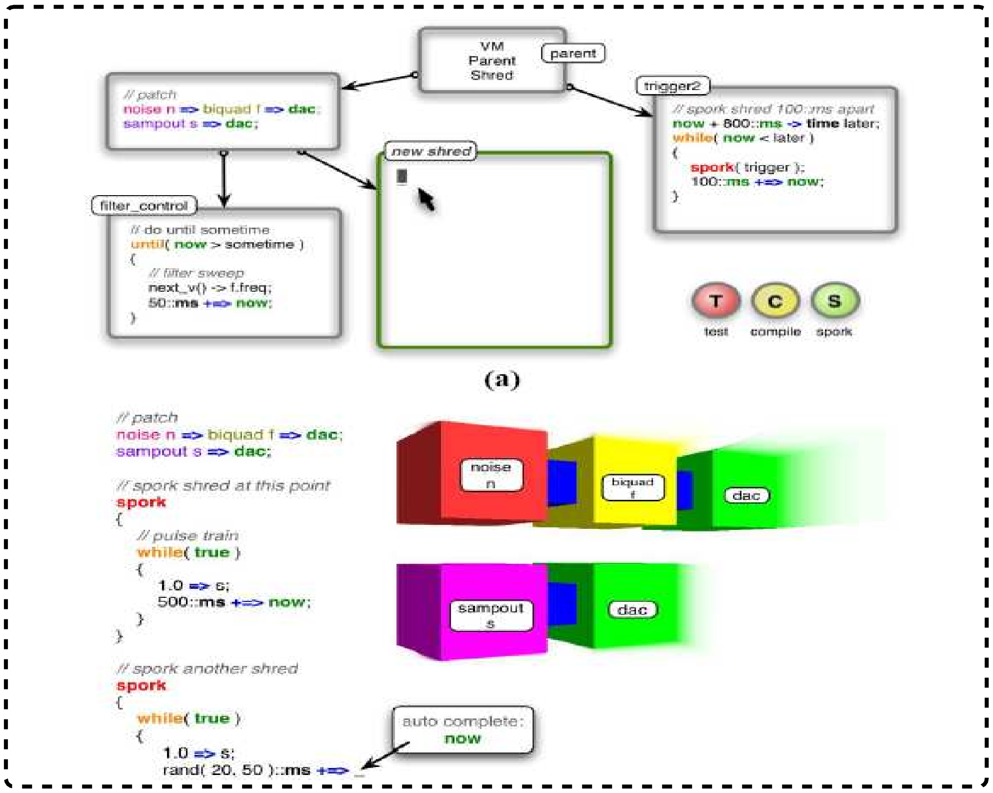
As an aside, audio performance can be retained by implementing the audio engine for authoring tools in C++ using a toolkit like STK or CLAM and interfacing with the Java based GUI and language engine via the Java Native Interface (JNI), as done in languages like JMSL.

3.1.2 Toolkits targeting the 3D space: OpenGL, Direct3D, Java3D

Using the 3D medium to design visual languages and interfaces for expressing and interacting with musical and sound structures is a sparsely researched subject. Although user interfaces for computer music software have come a long way from the state of the art at the time of Adrian Freed's 'plea' to the computer music community [Freed, 1995], there is still scope for large leaps to be made by simply reusing and integrating the improvements in 3D user interfaces and authoring tools.

Infrastructure frameworks offer graphics and user interface functionality at various levels of abstraction. Libraries like SDL and DirectX offer a thin layer of abstraction over the graphics hardware[4]. OpenGL, Direct3D and Java3D offer a scene graph API that facilitates the description of the 3D elements and transformations using a tree-like hierarchy. Environments like the Audicle (shown in Figure 3.3) operate directly at the level of the OpenGL scene-graph API to implement integrated development environments for sound design and sequencing [Wang and Cook, 2004]. This involves implementing all the functionality implemented in 2D user-interface toolkits, from scratch, in 3D. It is a non-trivial effort to implement an SWT like toolkit or an Eclipse-like IDE to leverage the additional

[4]These APIs actually offer abstraction layers over a whole range of subsystems, including graphics, audio and input devices.



```
// patch
noise n => biquad f => dac;
sampout s => dac;
```

```
// spork shred 100::ms apart
now + 800::ms -> time later;
while( now < later )
{
    spork( trigger );
    100::ms +=> now;
}
```

```
// do until sometime
until( now > sometime )
{
    // filter sweep
    next_v() -> f.freq;
    50::ms +=> now;
}
```

```
// patch
noise n => biquad f => dac;
sampout s => dac;

// spork shred at this point
spork
{
    // pulse train
    while( true )
    {
        1.0 => s;
        500::ms +=> now;
    }
}

// spork another shred
spork
{
    while( true )
    {
        1.0 => s;
        rand( 20, 50 )::ms +=> _
```

Fig. 3.3 Audicle: The use of the 3D medium to design a visual language and IDE.

possibilities afforded by the 3D medium. It is easier to implement such functionality over a 3D-engine intended for game development (Section 3.4), or with a new-generation UI toolkit like Windows Presentation Foundation (Section 3.1.6).

3.1.3 Flash

Adobe Flash is a graphics platform that evolved in the mid '90s primarily as a means for embedding vector graphics and animation on web pages [Wikipedia, 2006d]. With the introduction of the Flex group of technologies in 2004, it has evolved into a full fledged cross-platform user-interface platform for developing both web and stand-alone applications. Flex also comes with a powerful designer tool that generates an XML description of the user-interface[5] designed in a way that permits the UI design to be decoupled from the control

[5]The XML description format used by Flex is called MXML.

or application functionality.

User-interfaces that are deployable both on the desktop and over the web are particularly relevant to application extensibility, because web-deployable applications are designed to be installation-free. This means that new functionality can be added and deployed in a web-application even while it's running. Combined with the fact that the application is automatically accessible by millions of users over the internet, this makes a very potent architecture for end-user extensibility. Music and other design-intensive domains, in particular, are well suited to an end-user extensible application architecture that is deployed over the web, as application extensions and design-artifact additions by members of a design community are instantly accessible to everyone over the internet. Desktop applications, on the other hand, have full access to various resources and privileges not available to web applications. Being able to use the same resources across both implementations shortens development time.

3.1.4 Mozilla/Javascript

With the increase in popularity of the Mozilla Firefox browser, the runtime for the user interface[6] and component/extensibility toolkit designed to implement the Firefox browser became universally available. This made the Mozilla platform a viable way to develop both user interfaces and application extensibility [Boswell et al., 2006]. The component technology implemented for the Firefox browser, which is the foundation for its scriptability and extensibility features, is called XPCOM. The graphics and layout engine, Gecko, (implemented for Firefox) is reusable as XPCOM components. Like Flex, Mozilla defines a declarative XML-based GUI description format known as the XML User interface Language (XUL). Of particular interest is the fact that the Mozilla framework allows for user-interface overlays to be dynamically de-serialized from XUL files at runtime. This means that a running Mozilla application can be extended with new UI components that have been serialized into XUL files. This brings the Mozilla platform up to the level of an infrastructure framework that facilitates the development of an extensible application as advocated by the current work.

[6]Originally known as the Cross Platform Front End (XPFE).

3.1.5 SVG/DHTML/Javascript (a.k.a AJAX)

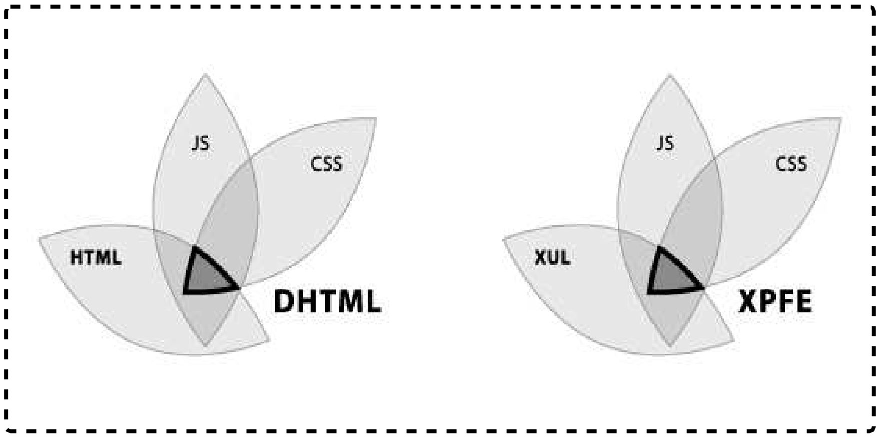


Fig. 3.4 A comparison of major components in a Mozilla based application with traditional DHTML/Javascript applications.

The years 2005 and 2006 have seen the resurgence of a user-interface and web-application model that predates the Mozilla framework - a combination of DHTML and Javascript. In its current form, known as 'Asynchronous Javascript And Xml' (Ajax), the principle advantage claimed by the model is a standards-based replacement to proprietary user-interface technologies for the web, like Adobe's Flash/Flex [Garrett, 2005]. The Ajax model involves using a combination of Dynamic HTML (DHTML) (as shown in Figure 3.4), Cascading Style Sheets (CSS) and ECMA-262 Script (Javascript) to develop rich user interfaces for applications that run inside sundry browsers[7]. All three of these technologies have been standardized by the W3C. The importance of Ajax is its portability across browser technologies. As such, there is no guarantee that the most competitive UI technology available would adhere to these standards. It is appropriate when designing a tool where portability is of the utmost concern, and the lowest-common-denominator functionality that the standards guarantee across browser platforms is adequate. Recently, entire windowing toolkits have been developed for the Ajax platform, thus creating new user-interface platforms on top of this virtual user-interface platform. Of course, these windowing toolkits bring with them their own set of incompatibility problems while retaining compatibility with web-browser standards.

[7]As compared with Mozilla technologies, which require the Mozilla browser.

3.1.6 Windows Presentation Foundation (WPF)

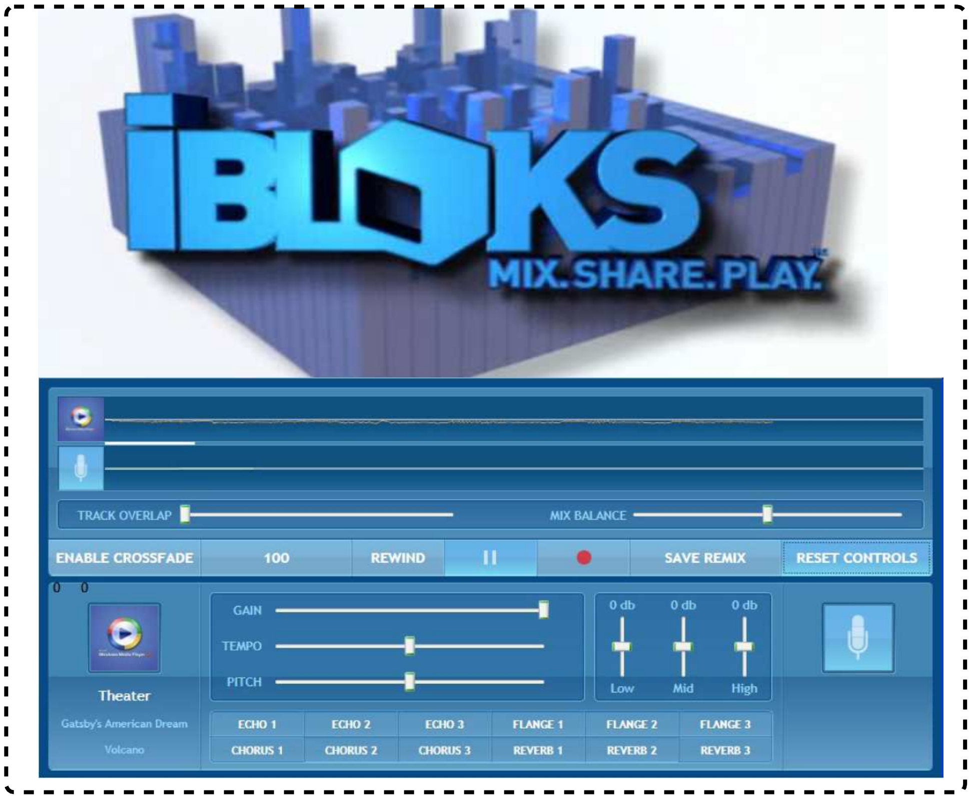


Fig. 3.5 iBlocks, a music remixing application written using WPF.

Windows Presentation Foundation is the graphics sub-system of the Windows Vista family of operating systems. Figure 3.5 shows portions of the UI of a music remixing application written using WPF. The family itself is scheduled to be released in January 2007[8]. WPF assimilates many of the improvements in SWT, Flash and Mozilla while adding other useful features. Like Flex/Flash's MXML and Mozilla's XUL, WPF uses an XML based application description language developed for Windows Vista called the XML Application Markup Language (XAML). Similar to Mozilla, XAML descriptions of user-interfaces can be built into applications that are deployed either on the desktop or via the

[8]The information on WPF included in the current work is based on design experiments with the WinFX Beta SDK version 2.0 Build 50215. As the design experiments only aim to reach conceptual conclusions about the nature of the programming model supported by the WinFX SDK, the Beta status of the SDK does not influence the relevance of the conclusions made.

web. Like Mozilla, WPF supports user-interface overlays, and allows designer-generated XAML to be de-serialized dynamically to load user-interface components into a running application. WPF applications can be scripted and extended using any .NET language, which includes C++/CLI[9], C#, Visual Basic and Python.

WPF supports some features unique to all the user-interface frameworks surveyed so far. One such feature is 'control/content composition'. This enables *any* element of the user-interface to serve as a container for any other element. For example, this would allow a text editor control to allow arbitrary shapes to be laid out next to letters, or for list controls to contain lists of complex shapes, which in turn could be trees of other complex shapes. Another feature is what is known as 'lookless controls'. All WPF controls have a well defined abstract structure and behavior. However, the visual appearance of the controls can be arbitrarily customized, beyond basic "skinning" facilities provided by most frameworks. WPF simplifies the programming of interactivity greatly by supporting hit testing of arbitrary elements in the GUI and declarative specification of animation. The hit testing also works with the animation turned on. All of these features also have 3D versions and the elements actually go through the 3D rendering pipeline. This, combined with the availability of a good number of 2D and 3D designer tools[10] for WPF makes it a great choice for developing extensible, end-user scriptable authoring applications.

3.2 Scripting Languages and Embedding/Extension Facilities

Every so often, users of a Category A tool find themselves repeating a complex sequence of operations many times with different values. Though the structure of the operation remains the same, it operates on different values. This may involve copying a certain set of objects and applying some transformation to all of them, generating a sequence or creating a set of objects based on a sequence of data. An early solution to these problems involved allowing the user to record his actions involving the user-interface as 'macros' or repeatable operations, and to 'play' the sequence of operations back when it needed to be repeated.

[9]CLI stands for Common Language Infrastructure. C++/CLI includes proprietary extensions to ISO/C++ for .NET

[10]At the time of this writing (July 2006), Microsoft Expression and Microsoft Acrylic products are available for 2D interface design. Xamlon, Zam3D and Maya support XAML output for 3D models and interfaces.

This solves the problem of repeating the operations, but still leaves the problem of changing the data for new invocations of the macro. A common solution to this problem involves giving access to the underlying implementation of the tool via a programming language (a Category B framework). The access is given by exposing this interface via the user interface itself. Thus, the user can record a macro, and then modify the 'script' corresponding to that macro via a macro editor. Additionally, she can modify this script to prompt for the data required for the particular invocation of the script. This concept of editing a 'script' of operations eventually led to the use of scripting languages to allow the end user to automate high-level tasks. Commonly used scripting languages include Javascript, Python, Perl and Basic. Less common but important languages include Lua, described by the authors as an 'extensible extension language' [Ierusalimschy et al., 1996] and Guile, a light-weight dialect of Scheme.

These languages are relevant in the light of an additional design requirement for music authoring tools - designing in advance for end-user scriptability. How can we architect these tools to be scriptable from the bottom-up? There are many frameworks available that facilitate automatic addition of scriptability to systems. One method is to use a 'wrapper generator' like the Simplified Wrapper and Interface Generator (SWIG) [Beazley, 2003] . Another method is to design all the components of the system to be Component Object Model (COM) components or Common Object Request Broker Architecture (CORBA) components. More recently, the .NET platform allows all .NET components to be scripted using any .NET language. There are implementations of a variety of scripting languages for .NET, with the most supported languages being Visual Basic Scripting Edition and IronPython. Irrespective of the method of generating bindings of the internal interfaces of the application to scripting languages, the application design needs to be fundamentally altered to support the kinds of operations that are needed to allow the end user to automate application tasks. Section 4.2 illustrates the design process involved in going from a C++ library for synthesis (STK) to a simple, scriptable, object-oriented software synthesizer (Grease).

3.3 Visual Modeling Tool Generators

Text-Based scripting languages have been the traditional way to support automation of repetitive tasks in many application domains. Computer music, on the other hand, has been well served by visual scripting languages like Max/MSP and OSW. Requirements from a very different domain have driven the development of reusable and reconfigurable visual language environments - the field of Modeling and Simulation. Principally consisting of researchers studying complex physical systems composed of a combination of discrete and continuous behavior, this community has driven innovation in formally-described generators for automatically generating visual environments that are tailor-made for editing and compiling visual models. These visual models describe systems in a very narrow and well-defined field of enquiry, for example, logic circuits.

In recent times, the tools produced from the Model Driven Architecture (MDA) school of software engineering have converged with those produced by the physical systems community resulting in tools like the Generic Modeling Environment (GME). Another closely related approach is the 'Software Factory' approach proposed by Jack Greenfield and Keith Short [Greenfield and Short, 2003], which has ultimately resulted in the Domain Specific Languages (DSL) toolset in the Visual Studio Team System product from Microsoft. The technical differences in methodology are not relevant to the current work. Some of these tools, and the manner in which they can be used for automating the implementation of portions of a music authoring tool, are described below.

3.3.1 AtoM^3

AtoM^3 is a tool for modeling and meta-modeling developed at McGill University [Vangheluwe and de Lara, 2002; 2003]. It consists of a graph-transformation kernel and a graphical editor engine, developed in Python and Tcl/Tk. AtoM^3 can be used to automatically generate visual modeling environments for a wide variety of visual languages. Figure 3.6 shows a visual environment for modeling road traffic, generated using AtoM^3. The approach used in AtoM^3 can be compared to generalizing the process of generating compilers for a textual language. AtoM^3 and GME extend the idea of compiler-compilers like yacc and SableCC to the realm of visual languages. The process of implementing a visual language environment using AtoM^3 involves specifying the entities and relationships in the abstract syntax of the

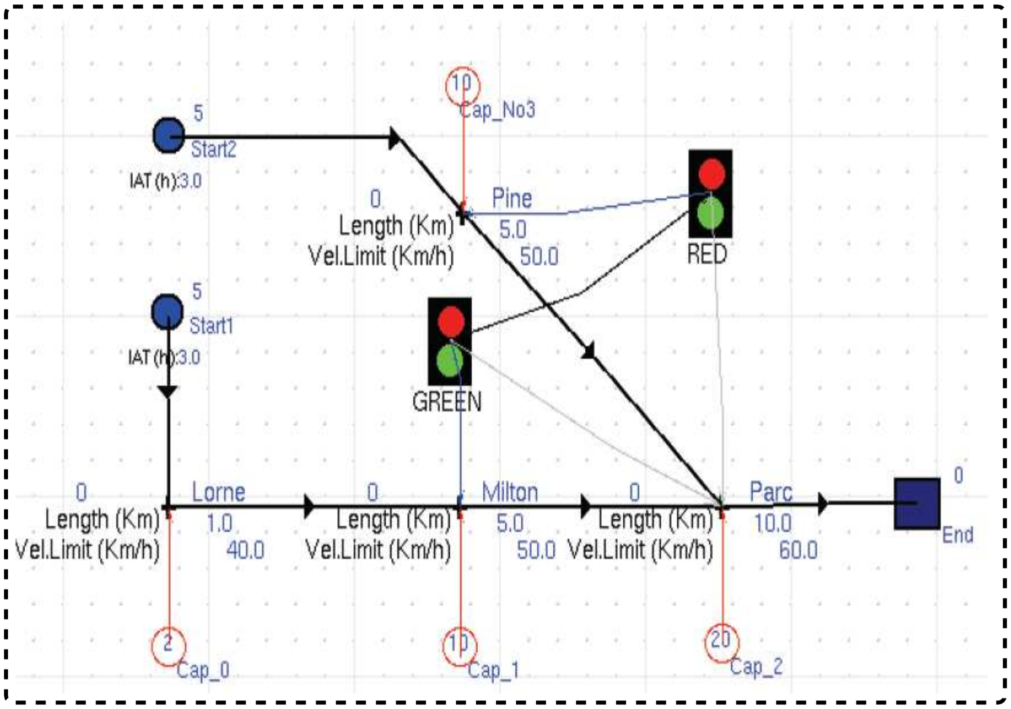


Fig. 3.6 A visual modeling environment generated by AtoM^3 [Sun, 2005]

visual language using a meta-formalism like Entity-Relationship (ER). The semantics of the visual language are then formally specified by defining graph transformations that map the visual models to another, well-known formalism like Discrete Event Specification (DEVS). The approach of assigning a semantics to a language by mapping it to another language is known as Denotational Semantics [Stoy, 1977]. The language can also be interpreted by assigning an 'operational semantics' by generating code that runs on some model of computation. Such a model could be, for example, a C++ program or MIDI/OSC commands to a synthesizer. AtoM^3 provides an intuitive, declarative, rule-based system for defining such transformations.

3.3.2 GME - A Generic Modeling Environment

GME is a tool-suite for building visual environments and interoperable tool chains for many different domains. Like AtoM^3, GME has a generic graph engine as well as a configurable editor engine. The editor engine, also known as the 'GUI client' to the GME engine,

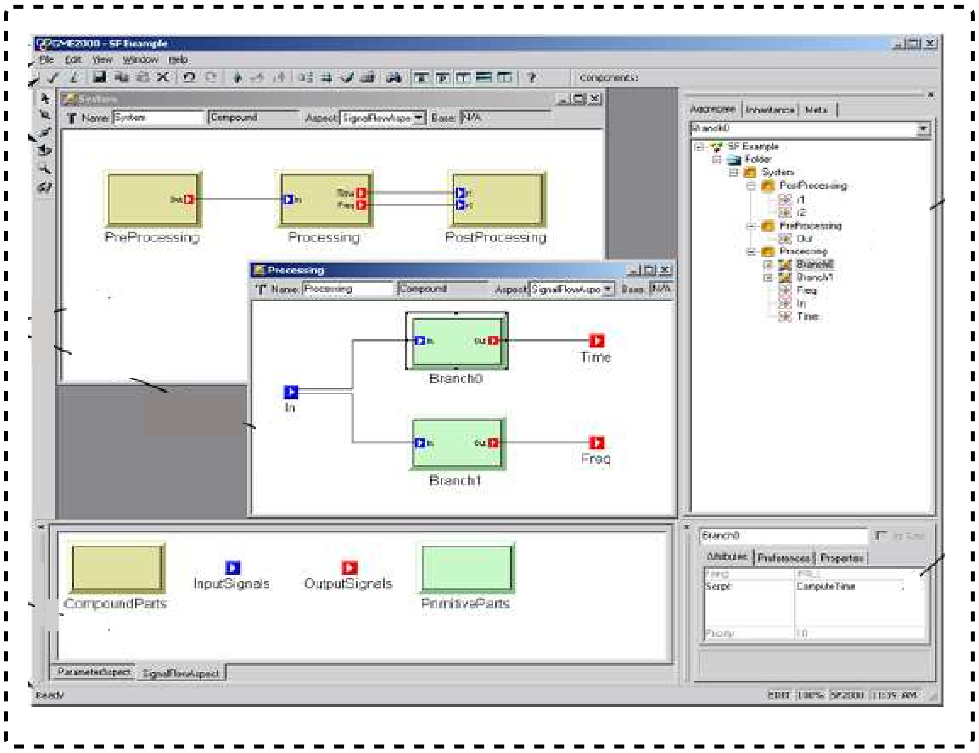


Fig. 3.7 A visual modeling environment generated by GME [Institute for Software Integrated Systems, 2006]

can be configured by meta-models that are specified using UML, similar to metamodeling with Entity Relationship in AtoM^3. The process of using GME to generate a visual modeling environment is slightly different than in AtoM^3. The meta-modeling phase is common, but GME metamodels are specified using UML[11]. Metamodels in GME are known as ‘paradigms’. Once a visual formalism (for example, data-flow networks) for the domain has been defined as a GME paradigm in UML, it can be used to generate a visual environment. Figure 3.7 shows a simple data-flow modeling environment generated using GME. Once the formalism has been finalized, it can be used to generate a skeletal interpreter using GME’s component creation tool. GME exposes its graph engine core and graphics engine as COM components, hence the generated skeletal interpreter can be extended using any

[11]The fact that the meta-modeling formalism (UML or ER) can itself be customized or modeled using a meta-meta-model is illustrative of the completeness of the methodology of meta-modeling.

programming language that supports programming against COM[12]. Using this interface, interpreters can be written that transform or 'run' the visual models that are created by a user using the generated modeling environment. The GME tool is further explored in Section 4.1. GME also provides a visual language for graph transformation called 'Graph Rewriting And Transformation (GReAT)' [Agrawal, 2003], that bears some similarity to the environments provided by Elody and OpenMusic[13]. It also bears some similarity to AtoM^3's rule-based transformation facility in that it (GReAT) is also rule-based.

3.3.3 EMF - Eclipse Modeling Framework and GEF - Graphical Editing Framework

The Eclipse framework (introduced in Section 3.1.1) provides Java-based components for building extensible development environments. Among these is a recent framework for interchanging various representations of an underlying model, called the Eclipse Modeling Framework, or EMF. EMF allows programmers to generate Java code from models, extract models from annotated Java code, and interchange these representations with XML and UML representations.

The Graphical Editing Framework, on the other hand, is a toolkit for building graphical editors using Eclipse components. Together[14], the EMF and GEF can be used to build visual modeling environments. Figure 3.8 shows AcmeStudio, a visual tool for architectural[15] modeling developed at Carnegie Mellon University [Schmerl and Garlan, 2004]. The development model afforded by EMF/GEF (which are Category B frameworks for building visual languages) is not as simple and straightforward as programming with tools like AtoM^3 and GME (which are Category A tools for building visual language interpreters and compilers). On the other hand, building complex generalized visual languages comparable with those that can be built with WPF (see Section 3.1.6) turns out to be too much of a

[12]Currently, however, the skeleton generation tool itself supports only C++ and Visual Basic, so this is a practical limitation.

[13]This similarity is purely from the point of view of the semantics of the visual language, and has nothing to with musical capability. GME and GReAT are completely unaware of music domain specific functionality.

[14]There is no dependence between EMF and GMF, and they may be used independently of each other as well.

[15]"Architecture" here refers to the architecture of complex systems in general, not just buildings, as used in the conventional sense.

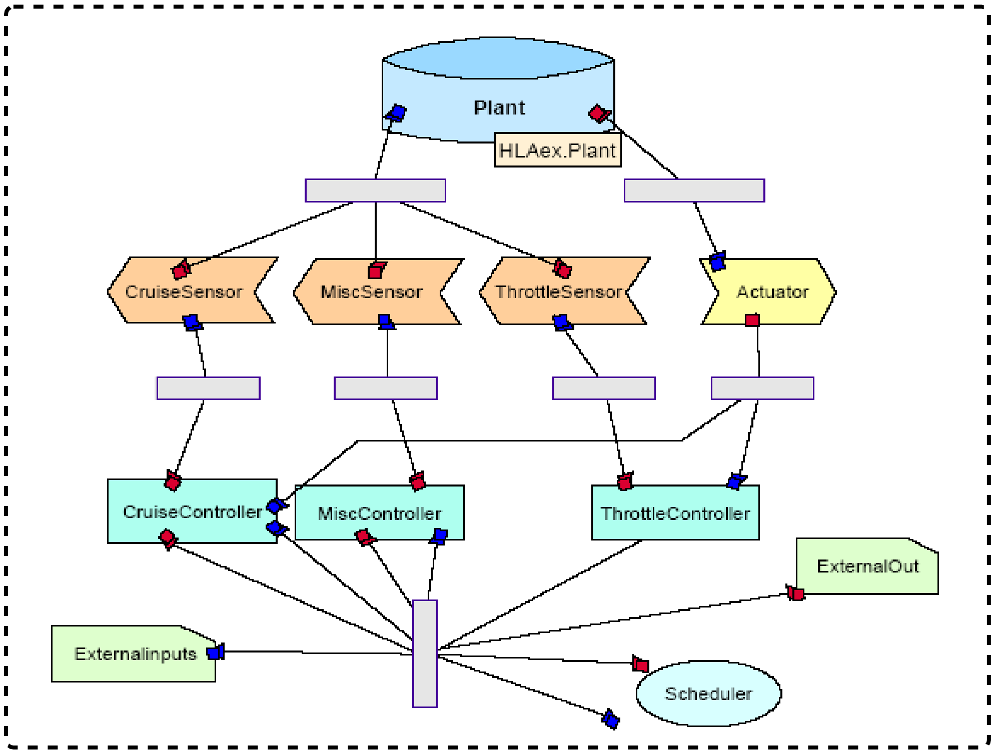


Fig. 3.8 Diagram created with AcmeStudio, a visual modeling environment designed using GEF

hack with Category A tools, and frameworks like WPF and EMF/GEF are more suitable in such cases.

3.4 Reusable Technology from the Video Game Industry

The computer gaming industry has evolved an entire methodology and set of tools for dealing with the fact that a large part of the game implementation process involves artists and designers who need to produce design and implementation artifacts that go into the game. These artifacts must be in a form process-able by machine. Yet, they cannot be created by programmers[16] because they require artistic expertise that they do not posses. Visual design and animation, for example, is a highly technical field with its own set

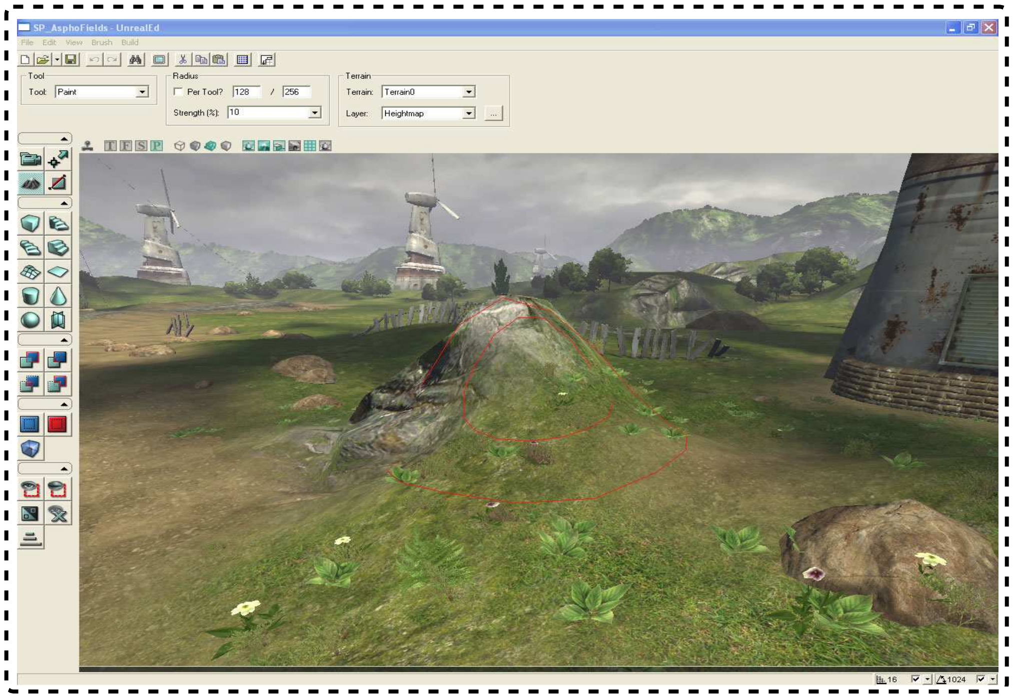
[16]Although a lot of modern large-scale games separate the activities, game design and programming were, in the beginnings of the game industry, often done by the same person. Notable designer/programmers are Sid Meier, Chris Sawyer, Will Wright and John Carmack [Wikipedia, 2006e].

of concepts and patterns. While programmers and software architects design the 3D and sound engines that actually render the models, how can the same engines be used by artists who have little knowledge of the programmatic interfaces to the engine? The problem extends all the way to the end users, with demand for tools that allow gamers, who are end users that have neither specialized artistic skills nor extensive programming knowledge, to customize games and produce their own modifications or 'mods' as they are known in the gaming industry. Such modifications are valuable to some community of gamers, and often add great value to the gaming experience. The following sections describe tools that address a number of these problems [Wikipedia, 2006f]. Game engines are Category B infrastructure frameworks for realizing games that build on various low-level APIs. Game engines are often designed with components that are years ahead of the technology that is used in business applications. This is usually because failure in games does no harm beyond causing a poorly selling game. For this reason, the returns on the success of a cutting-edge technique in games are usually much more than the risks associated with its failure. This is also the case in the domain of music authoring applications. The technology developed for game engines, is thus a good source for advanced, re-usable components for music authoring tools. Examples of popular game engines include Quake, Unreal Engine 3, RenderWare and Visual3D.NET

3.4.1 Game Content Editors

3.4.2 Game Engines

Game content editors are authoring tools for game assets that include all assets that can be rendered using the various components of the game engine. These include 2D/3D visual art, physical models and processes, rules used by the Artifical Intelligence (AI) engine, and so on. A good example is the Unreal Engine 3 Content Editor [Epic Games, 2006], UnrealEd. UnrealEd is a WYSIWYG editor for game assets that can be rendered using the Unreal Engine. It provides high level tools to edit 3D models, textures, animation and other assets. It also serves as a development environment for the scripting language supported by the Unreal Engine, called UnrealScript. Figure 3.9 shows a terrain editor that is part of UnrealEd [Epic Games, 2006].



Tool
Tool: Paint

Radius
Per Tool? 128 / 256
Strength (%): 10

Terrain
Terrain: Terrain0
Layer: Heightmap

Fig. 3.9 A terrain editor for Unreal Engine 3

3.4.3 Visual Scripting in Game Design

Describing game assets requires more than just creating static models and other art-work using a content editor. It is easy for programmers to encode the dynamics of game processes using general-purpose programming language interfaces that the game engine exposes. How can the same processes be described by concept-design experts who possess a deep understanding of the game's 'concept', but do not possess sufficient programming skills? Describing dynamic processes and rule-based systems have become an integral part of the conceptualization of various aspects of the game and the need for an intuitive language that can be used by game designers to describe the *dynamic* entities in game concept design has led to visual scripting environments for game design. Figure 3.10 shows Virtools Dev, a popular visual environment for game concept design [Virtools Inc, 2006]. The visual language complements Virtools Scripting Language, the text-based scripting language for the Virtools engines.

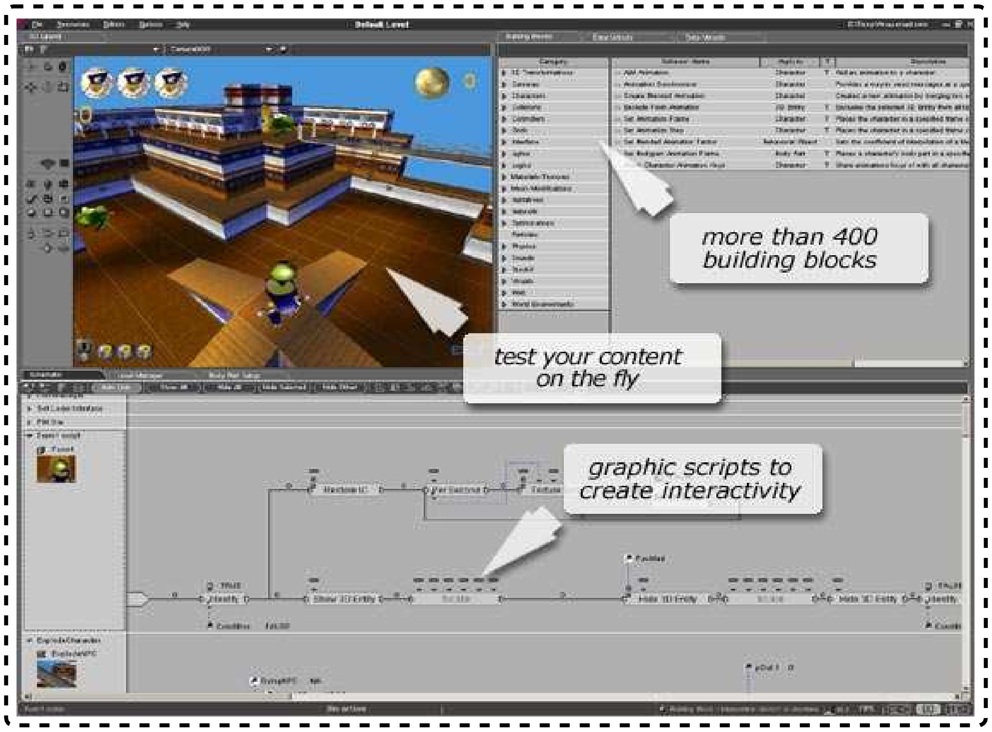

more than 400 building blocks
test your content on the fly
graphic scripts to create interactivity

Fig. 3.10 Virtools Dev Authoring Environment with Visual Scripting.

3.4.4 'Mod' Authoring Tools

'Mod' or modification authoring tools put the power of modifying games into the hands of the ultimate end users, the gamers themselves. The idea is to model variations in game play and expose this via a Category A tool. Often, however, the Category A tool also exposes a Category B framework that gives advanced mod designers access to the underlying game engine interfaces via a scripting language. Between game content editors, designed for seasoned designers and artists and 'mod' authoring tools designed for both amateurs and professionals, there are many architectural patterns in extensible and customizable authoring tools to be adopted from the video-game industry.

Chapter 4

Concept Demonstrations

This chapter presents the design and implementation of proof-of-concept demonstrations of the architectural styles for music authoring applications advocated by the current work. The manner in which combinations of a chosen set of off-the-shelf tools and frameworks can be used to produce extensible authoring environments is demonstrated. The objective is to use proven existing components and to identify the manner in which they could be glued together to interoperate. The components fall under 4 categories: a) Synthesis and audio signal processing frameworks and engines; b) Graphics and user interface frameworks; c) Visual modeling tool generators and engines; d) Scripting language engines.

The goal is not to produce a quantitative evaluation. Rather, the concept of integrating various *classes* of tools and frameworks for the purpose of building end-user extensible authoring environments is sought to be validated by demonstration. However, the choice of components in each class has been influenced by literature and online community reviews (surveyed in the previous chapters) with regard to the following criteria: 1) suitability for the purported role of the component in the overall architecture of an extensible authoring environment 2) performance 3) commercial and/or community support 4) ease of integration with other quality components.

It is noteworthy that in each class, many components, including those surveyed in the previous chapters, meet the aforementioned criteria and may be more suitable in a given context. In the context of the demonstrations, the choice of some components (for example, the use of IronPython as a scripting engine) has been influenced by the need to integrate with other components (like Windows Presentation Foundation). The design rationale and implementation details involved are further elucidated in the sections that follow. The manner in which the same goals can be attained with comparable tools and frameworks is highlighted where relevant.

4.1 Experiments with GME

Early on, the Generic Modeling Environment (GME) toolset was identified for use as an extensible visual-language interpreter/compiler for interpreting multiple visual notations[1]. The environment was introduced in Section 3.3.2. The implementation details involved in using GME are detailed here. The manner of extending GME to support a given visual notation is prototyped for the case of signal-flow graphs and hierarchical sequences. The aim is to illustrate the manner in which these environments can be automatically generated from formal descriptions. The visual language is mapped to C++ code that uses the Synthesis ToolKit (STK), introduced in Section 2.2.6.

4.1.1 Steps involved in using GME

Figure 4.1 shows the steps involved in generating a new visual authoring environment using GME. The process is quite similar to using a 'compiler-compiler' like yacc, bison, ANTLR or SableCC [Wikipedia, 2006a] to generate a compiler for a textual language. In the case of environments like GME, the 'grammar' of the visual language is specified using a declarative meta-language, UML. Unlike the context-free Backus Naur Form (BNF) meta-language traditionally used to describe the syntax of textual languages, the meta-language used by GME is the object-oriented Unified Modeling Language (UML), in conjunction with the Object Constraint Language (OCL)[2].

Once the structure and constraints of the visual modeling language and environment are defined as a 'meta-model' in UML, a tool called the 'MetaGME interpreter' is invoked to generate the visual environment using these descriptions. This environment can now

[1]The initial choice was AtoM[3]. The choice of GME was motivated both by the possibility of implementing 'Decorators' (custom visualization for types programmable using ActiveX technology) as well as the author's familiarity with the C++ programming language, the language best supported by GME. Thanks to Hans Vangheluwe of the School of Computer Science at McGill for referring the author to GME.

[2]The UML and OCL are standards defined and maintained by the Object Management Group [Wikipedia, 2006b; Object Management Group, 2006]

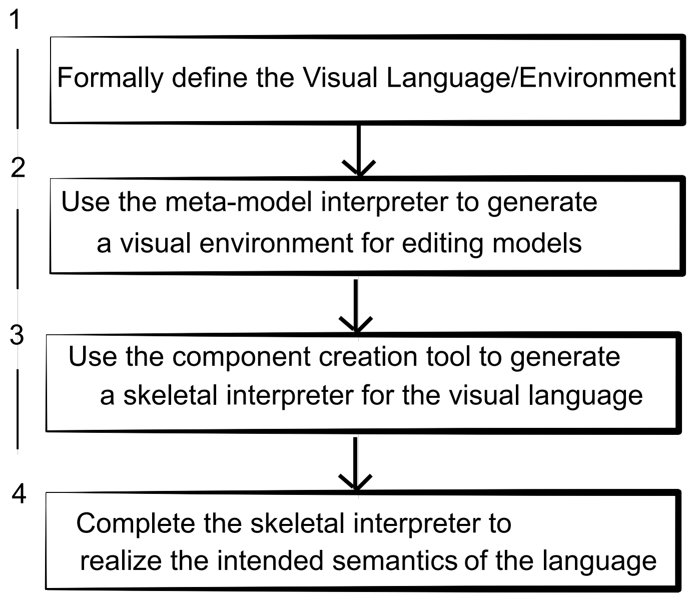

1
Formally define the Visual Language/Environment
2
Use the meta-model interpreter to generate a visual environment for editing models
3
Use the component creation tool to generate a skeletal interpreter for the visual language
4
Complete the skeletal interpreter to realize the intended semantics of the language

Fig. 4.1 Steps involved in designing a new visual environment in GME.

be used to create visual models conforming to the formally defined visual language. Next, a component creation tool is used to generate a skeletal COM-based interpreter for the visual language with do-nothing operations associated with each node type. The skeletal interpreter may then be modified to map the visual language models to other models, code, or real-time audio. The skeletal interpreter is generated as a set of C++ classes that represent the actions to be taken by the interpretor when visiting each node type. The class-implementation stubs may now be filled out to implement the semantics of the language. The next two sub-sections demonstrate this process for two visual languages, one for sound design and another for sequence design.

4.1.2 Generating a sound design environment using GME

The steps are illustrated here for the case of designing a sound design environment that allows a set of simple primitives to be combined to design sound synthesis and processing networks.

Step-I Figure 4.2 shows an extract from a formal meta-model of the environment in UML[3]. The meta-model expresses the elements of the visual syntax, and

[3]The complete model includes around 20 types and this model can be constantly updated to add new

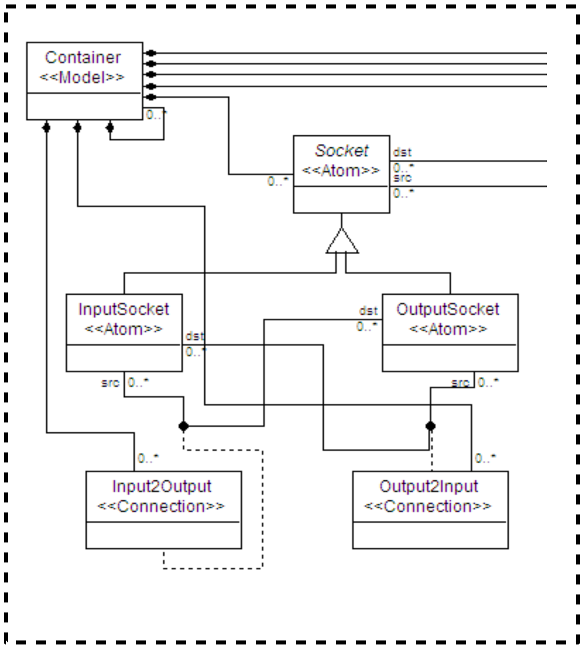


Fig. 4.2 UML extract from the meta-model for the sound design environment.

the ways in which they may be connected together. UML allows set and subset relationships to be expressed. For example, the types 'input-socket' and 'output-socket' are both defined to be specializations of the super-type 'socket'. This allows for assertions about sockets to be made, that apply to both input and output sockets. Types may also have attributes, with constraints specified on them. For example, a 'mixer' type can be defined such that the gains on each channel are non-negative. Object attributes can be edited in a toolbar in the generated visual environment. Constraint implementation is further discussed in Section 4.1.3. Types may also be associated with a simple image to be displayed as an iconic representation of objects of the type. For more advanced use-cases, a visualizer class may be implemented using the GME Software Development Kit (SDK) for objects of the type. The visualizer can be implemented in any language that supports ActiveX/COM programming. The visualizer can utilize the structure and content of the object to appropriately visualize it. For

features to the language. However, consistency conflicts with the models defined in previous versions of the language must be taken into account before making such modifications.

example, a frequency response may be visualized by drawing a spline to fit the peaks and troughs of the response function. Decorators are not explored in this demonstration.

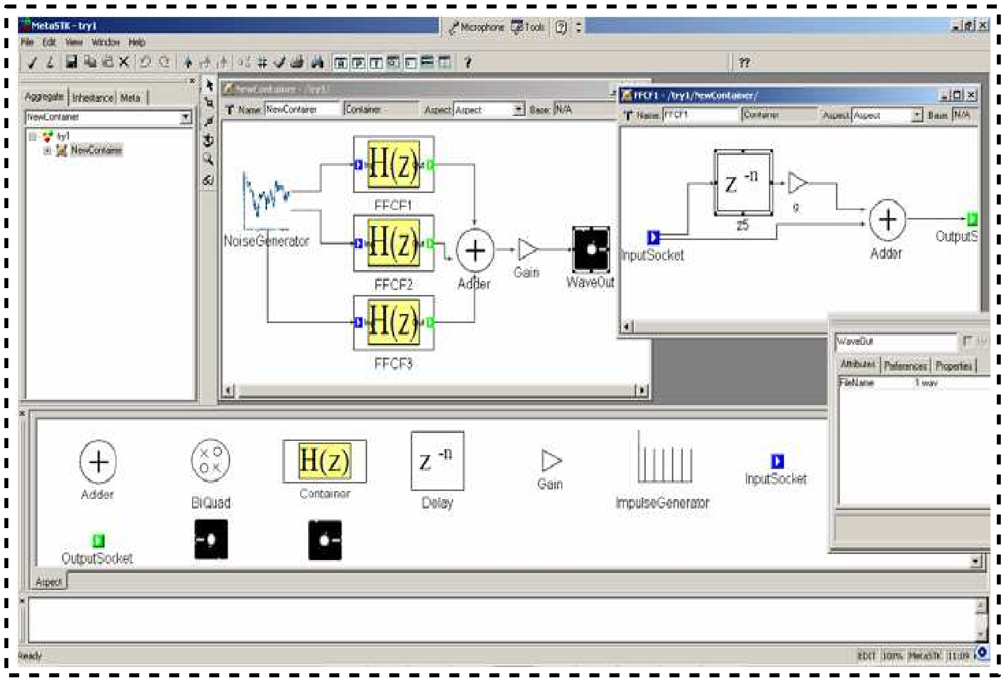
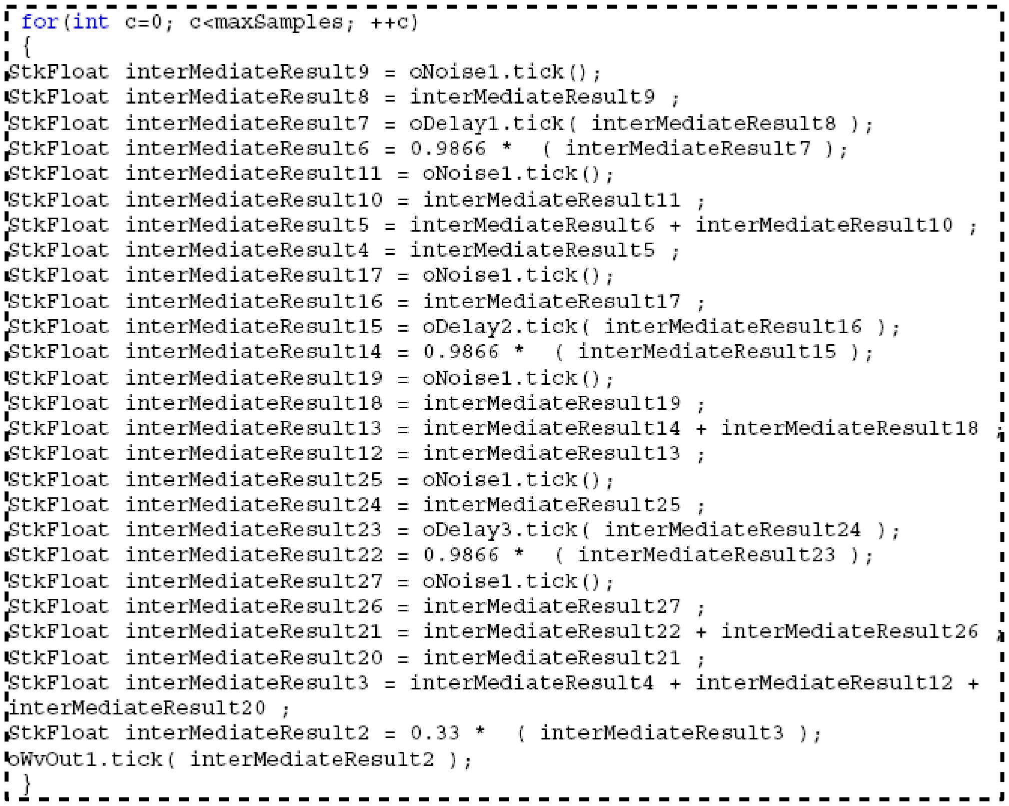


Fig. 4.3 The generated sound design environment.

Step-II Once the environment was formally described, the MetaGME interpreter was invoked to automatically generate a visual editor for language. The environment so generated was then used to create a number of models to determine whether it actually allowed the users to create the class of visual models intended to be meta-modeled in MetaGME. The meta-model is modified and Step 2 is repeated until this is true. As many iterations as are required to produce the final environment may be carried out to arrive at the final environment[4]. This is important as the meta-modeling process is prone to conflicts arising from evolving the meta-model. The user-interface of the visual design environment is much

[4]This is, of course, not a rigorous way to test whether the environment meets the specifications. A discussion of the design of a suite of test-cases to automate the testing of the visual environment is outside the scope of the thesis investigation, and is specific to the requirements of the particular environment being designed.

like many classic sound design environments like Max/MSP, OSW and Synth-Builder. Unlike many of these environments, though, the visual language of the generated environment can be customized to a large degree by re-designing the meta-model of the environment.



```
 for(int c=0; c<maxSamples; ++c)
 {
StkFloat interMediateResult9 = oNoise1.tick();
StkFloat interMediateResult8 = interMediateResult9 ;
StkFloat interMediateResult7 = oDelay1.tick( interMediateResult8 );
StkFloat interMediateResult6 = 0.9866 *  ( interMediateResult7 );
StkFloat interMediateResult11 = oNoise1.tick();
StkFloat interMediateResult10 = interMediateResult11 ;
StkFloat interMediateResult5 = interMediateResult6 + interMediateResult10 ;
StkFloat interMediateResult4 = interMediateResult5 ;
StkFloat interMediateResult17 = oNoise1.tick();
StkFloat interMediateResult16 = interMediateResult17 ;
StkFloat interMediateResult15 = oDelay2.tick( interMediateResult16 );
StkFloat interMediateResult14 = 0.9866 *  ( interMediateResult15 );
StkFloat interMediateResult19 = oNoise1.tick();
StkFloat interMediateResult18 = interMediateResult19 ;
StkFloat interMediateResult13 = interMediateResult14 + interMediateResult18 ;
StkFloat interMediateResult12 = interMediateResult13 ;
StkFloat interMediateResult25 = oNoise1.tick();
StkFloat interMediateResult24 = interMediateResult25 ;
StkFloat interMediateResult23 = oDelay3.tick( interMediateResult24 );
StkFloat interMediateResult22 = 0.9866 *  ( interMediateResult23 );
StkFloat interMediateResult27 = oNoise1.tick();
StkFloat interMediateResult26 = interMediateResult27 ;
StkFloat interMediateResult21 = interMediateResult22 + interMediateResult26 ;
StkFloat interMediateResult20 = interMediateResult21 ;
StkFloat interMediateResult3 = interMediateResult4 + interMediateResult12 +
interMediateResult20 ;
StkFloat interMediateResult2 = 0.33 *  ( interMediateResult3 );
oWvOut1.tick( interMediateResult2 );
 }
```

Fig. 4.4 Extract from code generated by the model compiler

Step-III A skeletal interpreter for the visual language was generated using a tool called the 'Builder Object Network (BON) Component Creation Tool'. The tool generates a set of graph-traversal classes specific to the particular visual language environment meta-model for which it was invoked. The interpreter can be understood as a set of transformations[5] on the object graph that represents the model in the visual language. The interpreter skeleton generator tool

[5]The object-graph itself is generally not modified. The transformation results in new object-graphs or linear-structures (possibly commands to a synthesis virtual machine).

exposes the object-graph via an API called the 'Builder Object Network'. The interpreter is invoked via a button in the toolbar in the generated visual environment. It is realized as a dynamic link library (.dll) module that gets linked in with the visual environment. A registration process is automatically invoked to make the connection with the visual-editor module and the interpreter dll, as part of the interpreter build process. For the interpreter skeleton generated for the environment illustrated in Figure 4.3, the object graph included elements of the following types: Container, WaveOut, WaveIn, Adder, Delay, BiQuad, Gain, InputSocket, OutputSocket, ImpulseGenerator and NoiseGenerator.

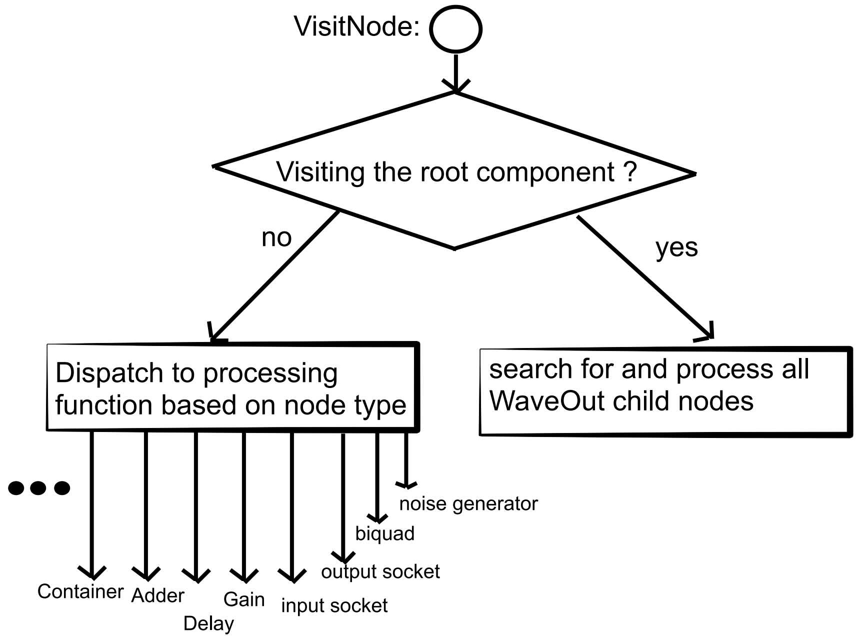

VisitNode:
Visiting the root component ?
no
yes
Dispatch to processing function based on node type
search for and process all WaveOut child nodes
noise generator
biquad
output socket
input socket
Container
Adder
Delay
Gain

Fig. 4.5 Outline of the sound-design-model compiler.

Step-IV The skeletal interpreter was then used to fill out the implementation for the actual interpreter. The semantics chosen for the language was to interpret the object-graphs as a signal-processing network realized using C++ code targeting STK. Invoking the interpreter in the generated environment produces C++ source code representing the synthesis or signal-processing component represented by the visual model being edited. This code was then compiled using the Visual C++ compiler to generate .wav files. Figure 4.4 shows an extract from the code generated for the model shown in Figure 4.3. Other usage scenarios are just as easily implemented. The interpreter may be designed to generate code or intermediate representations to achieve other tasks. The generated code may be integrated into a larger C++ application, or with code generated using other visual formalisms (for example, state-charts representing the behavior of the user-interface of the sound-synthesis or signal-processing component). The code may directly write MIDI/OSC streams to a software synthesizer[6]. Similar scenarios are explored in Section 4.2. These scenarios, however, use an architectural style different from the model-driven architecture[7] style presented in the current section. Figure 4.5 illustrates the outline of the algorithm used by the model compiler.

4.1.3 Generating a sequencing environment using GME

Step-I The sequencing environment provides a simplified representation of musical elements that is a "language" for modeling tonal structure. Figure 4.6 shows an extract from the meta-model used to describe the language and environment formally. The language includes types such as Note, Chord, 'Chunk' (a generic container for musical objects of any type), 'ChunkItem' (a super-class that represents the most generic sequence-able item, which is specialized to model other musical structures that the language defines) and so on. Types such as ChunkItem are labelled as 'FCO's. FCOs are 'First Class Objects'. They represent an abstract class of objects that cannot be instantiated. Only one of the sub-classes can be instantiated. This is similar to an abstract base class in C++. For example, there is no real meaning to creating a new object of type ChunkItem in the absence of additional information regarding what kind of ChunkItem object it is. On the other hand, a ChunkItem of Silence can be meaningfully instantiated. The MetaGME language provides many such powerful abstractions for describing modeling languages, including Models (objects with hierarchical structure), Atoms (primitives), References (aliases), Sets, As-

[6]STK itself does not support OSC. A software synthesizer that supports OSC may be used, or an OSC compatibility layer could be added for an STK based synthesizer.

[7]'Model Driven Architecture' is also a technical term used to refer to the specific brand of model-based design advocated by the OMG [Wikipedia, 2006h].

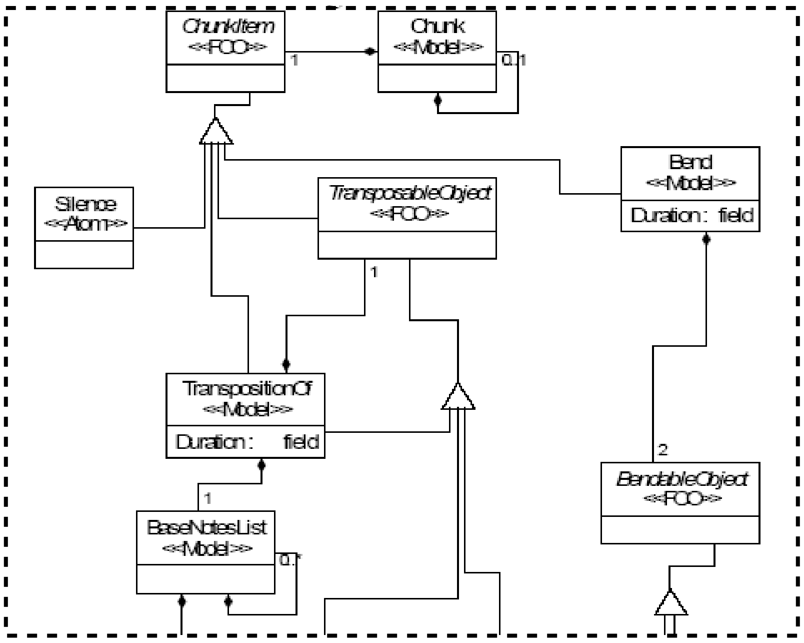


Fig. 4.6 Extract from the meta-model for the tonal structure language.

pects and Attributes.

Step-II Figure 4.7 shows the visual environment generated by running the tonality modeling language meta-model through the MetaGME interpreter. The environment utilizes hierarchy/containment as the principal abstraction[8]. The user-interface is quite similar to the sound-design environment. Complex objects are built by dragging and dropping primitives on to the design surface of the complex object. Attributes are set using the properties toolbar (lower right corner of Figure 4.7). It is noteworthy that the environment is atypical of the application of GME to music authoring environments advocated here, in the sense that it provides a general purpose, Category B language to model music. More typically, a very specific musical context would be meta-modeled to generate a Category A tool[9]. The meta-model for such a language, however, would use a few hundred elements and would build over abstractions much

[8]A full scale environment for modeling musical structures of a particular variety would utilize many of the other abstraction mechanisms provided by MetaGME.

[9]For instance, the arpeggios in a specific piece by, say, Al Di Meola.

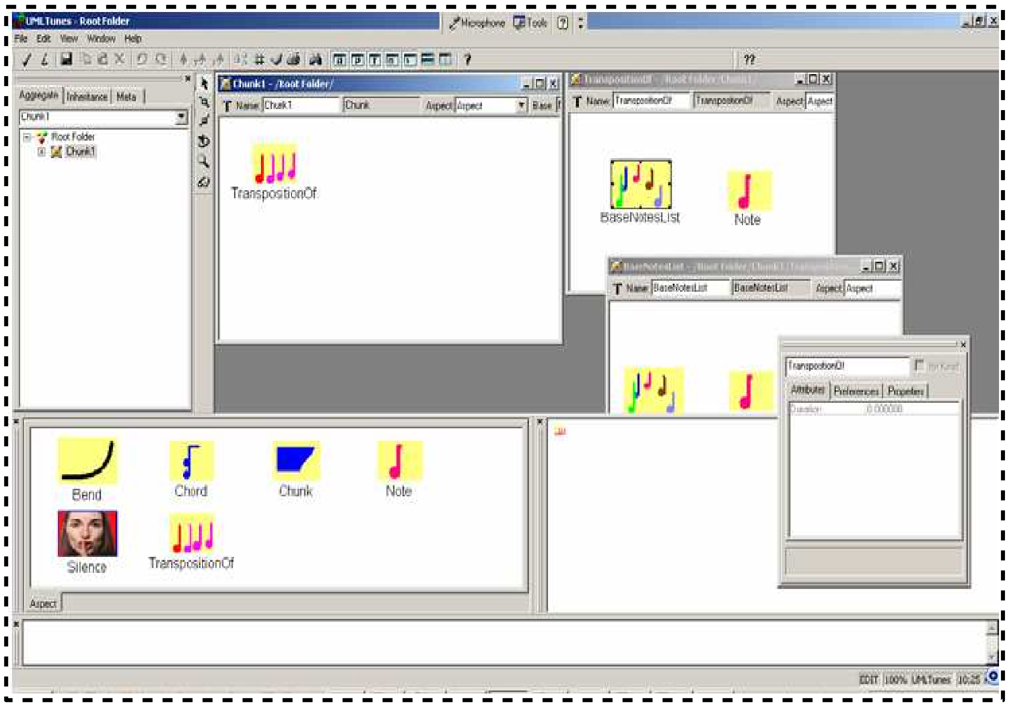


Fig. 4.7 The visual environment generated for the tonal structure language.

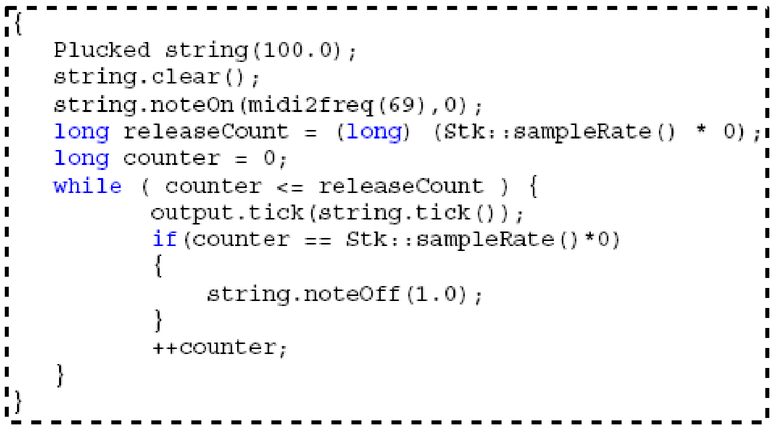
like those used in the current demonstration. Detailed user-experience and information-architecture research with musical experts, required to arrive at such a meta-model, is beyond the scope of the current work. Greenfield et al. [2004], in a recent account of meta-modeling, notes that:

> Natural languages, such as English, art, sculpture and music, are evolving organic phenomena. Given the current state of the art, they are too complex to be processed effectively by ... (meta-modeling methods).

Though the problem of arriving at a general description for 'all music' or for music of 'an entire genre' is infeasible, Category A tools for prototyping specific works may be designed using the methodology demonstrated here. Such a tool may also be generated by providing the domain-expert (musician or performer) with a Category B language and environment similar to the environment described here and writing *meta*-interpreter to generate the Category A tool described by the musician using the Category B language. This would be similar to using the MetaGME interpreter to generate the tonality language from the MetaGME (UML) meta-model. GME provides a facility called 'Metamodel Composability' that is similar. However, this would necessitate the musician to learn UML, which is rather unfriendly. More work is therefore required to make the generation of new environments by musicians, unaided by an engineering team, a reality. However, the cost and effort of producing such an environment is tremendously decreased by utilizing the methodology just outlined[10]. Tools like AtoM^3 reduce the learning curve even further by replacing the step of writing an interpreter in C++ by a declarative rule-based transformation facility.

Step-III The compiler generation process using the BON component creation tool is identical to the previous experiment.

Step-IV As in the previous experiment, the compiler for the sequencing language was designed to emit C++ source code utilizing STK. Figure 4.8 shows an extract from the C++ source emitted for the model. This results in a user-



```
{
   Plucked string(100.0);
   string.clear();
   string.noteOn(midi2freq(69),0);
   long releaseCount = (long) (Stk::sampleRate() * 0);
   long counter = 0;
   while ( counter <= releaseCount ) {
          output.tick(string.tick());
          if(counter == Stk::sampleRate()*0)
          {
              string.noteOff(1.0);
          }
          ++counter;
   }
}
```

Fig. 4.8 Extract from the C++ code generated for a sequence model.

[10]As a casual estimate, the author spent about 2 months learning the GME tool set. However, after the initial learning curve, utilizing GME to generate the environments discussed here took an average of a mere 10 days per environment. This is far lesser than approximately 6 months required to hand-code these environments. The quality of the resulting tool is also far superior. These numbers were not part of a formal quantitative study on the development activity. [Vanderbilt University, 2006] links to some studies by Vanderbilt University that may provide additional insights.

experience similar to compiling a CSound score file[11] and less like utilizing an interactive sequencer. This may be remedied by emitting MIDI/OSC messages to a real-time synthesizer directly, as opposed to generating source code. This was not explored in the experiment. Sections 4.2 and 4.3.1 explore this possibility with 'Grease' and WPF.

4.2 Design of the Scriptable Synthesizer - 'Grease'

4.2.1 Motivation

One of the challenges faced in the GME-based implementation was the mapping to the generated source code. End-users deal with high-level entities like signal-flow graphs and hierarchical representations of music. However, these are difficult to map to a low-level infrastructure framework like STK[12]. The 'Grease'[13] library was designed as a dynamic wrapper around STK that provided a simpler programming model for describing both the structure of signal-flow networks as well as event-driven and state-based behavior using a uniform, message-based programming model similar to Objective-C or SuperCollider. It also provides for a simple extensibility framework. The extensibility framework currently provides for build-time extensions for synthesis components. With a few modifications, it can also support run-time extensions. It already supports run-time extensions to the user-interface and message-based components by leveraging the dynamic programming facilities of the .NET platform.

The programming interface provided by the Grease wrappers is very similar to other dynamic, interpreted synthesis languages like Max/MSP, SuperCollider and ChucK. The choice of implementing a wrapper around STK, as opposed to using the existing libraries, was motivated by the need to integrate with the Microsoft .NET platform. The .NET platform itself was chosen for its ability to interface components written in one language with many other languages, for the introspection and reflection facilities provided by the

[11]RTCSound is an interactive version of CSound [Vercoe, 1990].

[12]Obviously, it is even harder to map the representations directly to a buffer of samples representing digital audio.

[13]The name 'Grease' was chosen to contrast with the traditional role of scripting languages, which is to act as a 'glue' between existing components. The implemented wrappers are seen as introducing dynamism between components created with STK, acting, in some sense, as a lubricant.

framework, as well as the ease of integration with the rich platform for visual interaction provided by Windows Presentation Foundation. Section 4.2.5 describes the manner in which exposing the synthesizer as a .NET assembly allows the functionality from Grease to be made available to C# programs and IronPython[14] scripts, each of which has its strengths in particular implementation scenarios.

A similar solution can be designed to integrate with the graphical and modeling facilities provided by the Swing and the Eclipse platform, by interfacing Grease to expose its functionality to Eclipse via the Java Native Interface (JNI). The Python scripting can be implemented in this scenario using Jython, a Python implementation targeting the Java Virtual Machine (JVM) [Hugunin, 1997b]. This may be implemented in future versions. The graphical features and WYSIWYG designers provided by WPF were better suited than the ones provided by the SWT and Eclipse platforms for the purpose of the demonstrations. However, both the .NET CLR and the JVM provide for reflection and introspection based dynamic loading of components at run-time, facilitating automatic extensibility. .NET based runtime extensions are explored in Section 4.3.1.

4.2.2 Grease Design

Figure 4.9 shows the key classes in the Grease class hierarchy. These classes are involved in implementing the message-passing semantics and the dynamic, garbage-collected interface provided by the library. Examples of wrapper classes, like Composite, Sequence and GuitarString are displayed in small-text.

Interface and High-Level Semantics

The Object class defines the simple interface that is required of all Grease objects. The interesting methods are tick, processMessage, and GetID. The tick method moves the Grease object to the next clock-tick in the simulation, and in the case of signal-processing objects, generates a sample of audio in the process. However, all Grease objects are assumed to be inherently 'tick'-able. Being a virtual function, it may be overridden by different classes to implement the ticking semantics of that class. 'processMessage' implements

[14]IronPython is a Python implementation targeting the .NET Common Language Runtime (CLR) [Hugunin, 1997a]. The CLR is a virtual machine functionally very similar to the Java Virtual Machine.

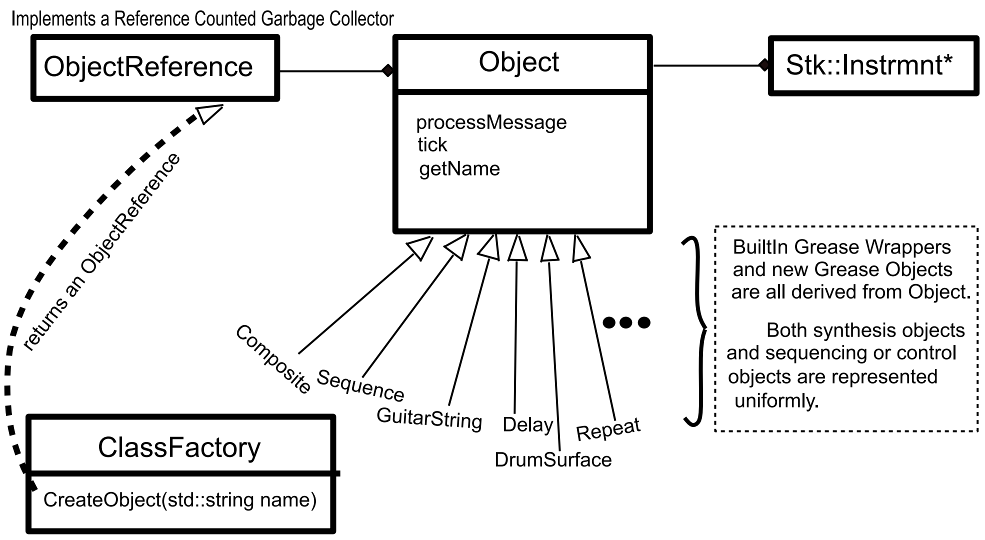

Implements a Reference Counted Garbage Collector
ObjectReference
Object
processMessage
tick
getName
Stk::Instrmnt*
returns an ObjectReference
BuiltIn Grease Wrappers and new Grease Objects are all derived from Object.
Both synthesis objects and sequencing or control objects are represented uniformly.
Composite
Sequence
GuitarString
Delay
DrumSurface
Repeat
ClassFactory
CreateObject(std::string name)

Fig. 4.9 Highlights of the Grease C++ hierarchy.

the message invocation semantics that defines the programming model of Grease. Much like SmallTalk and SuperCollider[15], all activities in Grease occur as a result of sending messages to objects. The creation of objects is no exception. Objects can be created by sending the name of the object's class to the 'new' object. The method-invocation is tied to the message by a lookup table that is defined on a per-class basis. The 'Object' base implements a set of macros that must be used by implementers of new Grease types. These macros automatically expand into code that installs a lookup table that maps messages to member-function pointers for the class. All messages are dispatched by the lookup table to the appropriate member function.

Memory Management

The creation of objects is orchestrated by 'ClassFactory' in conjunction with 'ObjectReference'. Each new type implemented in Grease must register a ClassFactory implementation that generates objects of the type. On invocation of the 'new' message, a global class-factory method looks up the class-factory registry to retrieve the appropriate class factory and instructs it to generate a new object. The library-user is not directly given access to an Object pointer though. In order for the object life-times to be automatically managed

[15]As also the JavaScript interface to Max/MSP.

by the garbage collection scheme, all Objects are wrapped by an ObjectReference handle class, which enables the garbage-collection. The ClassFactory therefore returns to users an ObjectReference to the Object that was created.

Since ObjectReferences are passed and stored by value, their life-times are automatically tied to the life-time of the program-block in which they (or the object that contains them) are defined. This completely frees the Grease library user from having to worry about allocating and deallocating synthesis and sequencing objects. ObjectReference itself implements garbage collection using a reference-counting strategy. The idea is to keep track of the number of references to an object and to delete the object when the reference count reaches zero, as described in [Wilson, 1992]. The reference-counting scheme, though not the most suitable in all cases, is popular with many language implementations. The standard implementation of Python, for example, uses reference-counting to implement its garbage collector. Other popular language implementations that use reference-counted garbage collectors include Visual Basic and Delphi [Wikipedia, 2006].

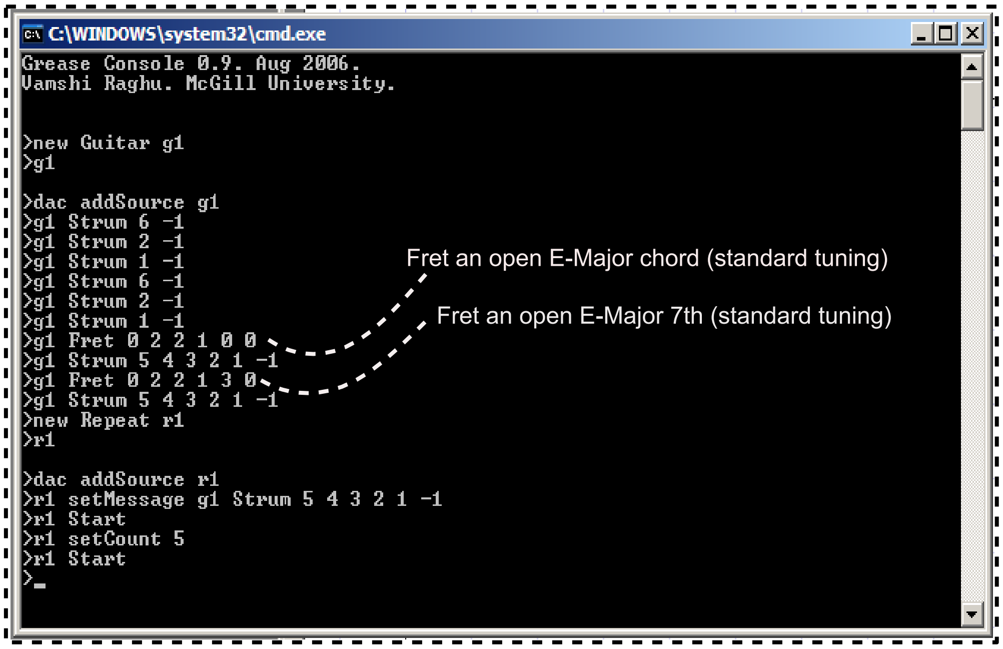
Core Extensible Language

The Grease library offers various programming-language-like features that are implemented on top of the basic message-passing scheme. Control-flow constructs in the Grease library are also implemented as first-class objects. An example is the Repeat object. It is the Grease equivalent of a while loop. Loops are executed by instantiating a new Repeat object (or modifying the parameters of an existing one) with the count, the target object and the message to be sent as arguments. The use of the message passing facility to implement all language constructs was motivated by the original architectural requirement of having an extensible language and was directly adapted from the ideas behind SmallTalk, SuperCollider and the Javascript interface to Max/MSP. This facilitates the end-user to implement new control-flow constructs that may be particularly suited to modeling a specific signal-processing or sequencing task.

Uniform Representation of Signal-Flow and Sequencing

Creation of signal-flow graphs is also done by sending messages to the new object (for unit-generator and sequencing-object creation) and sending connection messages to the appropriate objects. It is worth noting that Grease was meant to be used in conjunction with a visual editor like the ones developed in Sections 4.1.2 - 4.1.3 (using GME) and Section 4.3.1 (using WPF + IronPython). Therefore, while a message-sequence representation of signal-flow graphs may seem unnatural to do by hand, it is very amenable to being generated and manipulated by visual editors. In addition, the underlying message-based implementation enables interactive end-user scripting to be added automatically by exposing Grease variables via a command-line or GUI interface. This opens up the possibility of using innovative combinations of visual modeling and textual programming to describe musical processes.

4.2.3 Programming from the Grease Console


C:\WINDOWS\system32\cmd.exe
Fret an open E-Major chord (standard tuning)
Fret an open E-Major 7th (standard tuning)

```
Grease Console 0.9. Aug 2006.
Vamshi Raghu. McGill University.


>new Guitar g1
>g1

>dac addSource g1
>g1 Strum 6 -1
>g1 Strum 2 -1
>g1 Strum 1 -1
>g1 Strum 6 -1
>g1 Strum 2 -1
>g1 Strum 1 -1
>g1 Fret 0 2 2 1 0 0
>g1 Strum 5 4 3 2 1 -1
>g1 Fret 0 2 2 1 3 0
>g1 Strum 5 4 3 2 1 -1
>new Repeat r1
>r1

>dac addSource r1
>r1 setMessage g1 Strum 5 4 3 2 1 -1
>r1 Start
>r1 setCount 5
>r1 Start
>_
```

Fig. 4.10 Exposing the Grease synthesizer via a console.

Figure 4.10 shows a sample application exposing the Grease synthesizer via the console. The synthesizer can be scripted using the simple message-passing syntax as illustrated. New objects are given names by the user as they are created. Sending a message to an object involves naming the object and suffixing the message and its parameters after the name. Adding an object as a source to the 'dac' object causes it to be added to the audio output, as well as to invoke its tick method to synchronize with the audio output. The object may, in turn, tick its sources to produce its output, and so on. These details are hidden from the end-user of the scripting interface and are only relevant to the synthesis-application and extension/external-object developers.

In Figure 4.10 above, an instance of an extension object called Guitar is created. The Fret message causes the Guitar to update its internal state to reflect the fret-pattern passed as part of the message. It mimics the positioning of the left-hand[16]. Future 'Strum' messages strum the specified strings, taking into account the previously set fretting pattern. A sequence of periodic strumming can be scripted by using the Repeat object. More complex loops and conditionals are currently best implemented by writing an extension object as discussed in Section 4.2.4[17]. Of course, if the extension object itself is a signal-processing element (for example an extension that implements a guitar effects pedal), such an object can be modeled using a tool like GME or AtoM^3, and a model-compiler can be written to generate an entire class of Grease extensions (in this case different kinds of guitar effects pedal implementations), as well as the UI that can be used to 'script' it visually when the extension is instantiated. Internally, the UI would interact with the scripting interface to manipulate the instantiated extension at run-time.

4.2.4 Extending the Synthesizer - Mixing Grease code with Regular C++

This section illustrates the manner in which Grease, a Category B framework, can be used to implement Category A objects. The case of a specific drum loop is considered. The variations in the drum loop, which are meant to be navigated at run-time, are captured inside a Grease extension and are exposed as methods. The resulting extension can be scripted at run-time to explore all the variations in the drum-loop. It is best understood as a 'reactive-loop'. The demonstration is intended to serve as an illustration of the usage of the framework in a simplified performance/prototyping context. The drum loop consists of a bass track and a snare track.

[16]In the case of a right-handed guitarist, and vice versa.

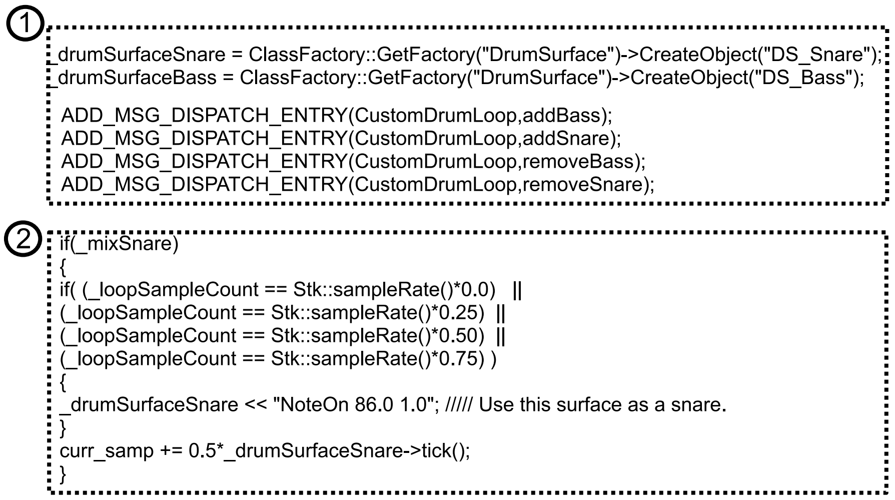
[17]It is also possible to write a simple set of conditional operators themselves as external objects. This would allow a particular set of sequencing scenarios to be scripted at run-time, instead of writing extension objects for each scenario.

```
class CustomDrumLoop : public Object
{
public:
 CustomDrumLoop (const std::string& name);
 virtual StkFloat tick();
 DECLARE_MESSAGE_MAP(CustomDrumLoop);
 // Message Map Entries
 DECLARE_MSG_HANDLER(Reset);
 DECLARE_MSG_HANDLER(Start);
 DECLARE_MSG_HANDLER(Stop);
 DECLARE_MSG_HANDLER(addBass);
 DECLARE_MSG_HANDLER(addSnare);
 DECLARE_MSG_HANDLER(removeBass);
 DECLARE_MSG_HANDLER(removeSnare);
protected:
 ObjectReference _drumSurfaceSnare;
 ObjectReference _drumSurfaceBass;
 int _loopSampleCount;
 bool _mixBass;
 bool _mixSnare;
 bool _started;
};
```

Fig. 4.11 C++ Interface for a Grease extension.

The implementation consists of a C++ class that captures the possible dynamics or interaction with the loop and exposes this interaction. Figure 4.11 shows the use of Grease macros to automatically generate the message dispatch and the Grease extension interface implementation for the class. The sequence supports the addition and deletion of the bass and snare tracks at run-time. Additionally, the loop may be reset while it's running. This simple set of operators can be used to script the loop at runtime, generating an interesting set of variations. The variations can be used to prototype a more complex sequence that can be saved as a script. The script, in turn can be loaded by another C++ based extension. This interplay of C++ code and Grease script can be leveraged to tackle a wide range of sequencing problems.

Figure 4.12 shows an extract from the C++ implementation file. Item 1 illustrates the manner in which Grease objects can be instantiated from C++ code. The code does not use 'new' and 'delete' operators, but rather delegates the object creation to the appropriate ClassFactory. The returned ObjectReference is stored by value in the CustomDrumLoop



```
_drumSurfaceSnare = ClassFactory::GetFactory("DrumSurface")->CreateObject("DS_Snare");
_drumSurfaceBass = ClassFactory::GetFactory("DrumSurface")->CreateObject("DS_Bass");

ADD_MSG_DISPATCH_ENTRY(CustomDrumLoop,addBass);
ADD_MSG_DISPATCH_ENTRY(CustomDrumLoop,addSnare);
ADD_MSG_DISPATCH_ENTRY(CustomDrumLoop,removeBass);
ADD_MSG_DISPATCH_ENTRY(CustomDrumLoop,removeSnare);
```

```
if(_mixSnare)
{
if( (_loopSampleCount == Stk::sampleRate()*0.0)   ||
(_loopSampleCount == Stk::sampleRate()*0.25)  ||
(_loopSampleCount == Stk::sampleRate()*0.50)  ||
(_loopSampleCount == Stk::sampleRate()*0.75) )
{
_drumSurfaceSnare << "NoteOn 86.0 1.0"; ///// Use this surface as a snare.
}
curr_samp += 0.5*_drumSurfaceSnare->tick();
}
```

Fig. 4.12 Extract from the C++ implementation for the extension.

object. In turn, the CustomDrumLoop object registers a ClassFactory object that creates CustomDrumLoops. This allows either the command line, or another C++ extension to delegate the creation of CustomDrumLoops to the ClassFactory. Additionally, the reference counting scheme implemented through the ObjectReference class manages the life-time of the DrumSurface objects (see Figure 4.12, Item 1). Therefore, no 'delete' operations are necessary to free the DrumSurfaces. They will be deleted when they are no longer being used, as determined by the reference-count going to zero. Grease macros are used again to map the message handlers to the appropriate messages. Item 2 illustrates the use of C++ control-flow constructs like 'if' statements to orchestrate the sequencing. The ObjectReference class overloads the << operator to mean message-dispatch. This causes the message to be looked up in the message-dispatch table corresponding to the type of object that the ObjectReference contains, and the invocation of the appropriate member function.

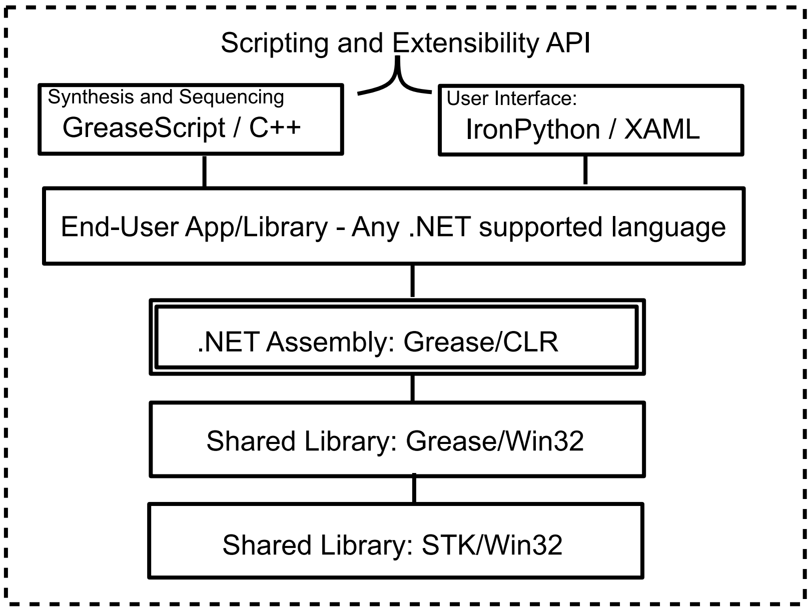

Scripting and Extensibility API
Synthesis and Sequencing
GreaseScript / C++
User Interface:
IronPython / XAML
End-User App/Library - Any .NET supported language
.NET Assembly: Grease/CLR
Shared Library: Grease/Win32
Shared Library: STK/Win32

Fig. 4.13 Grease/CLRs place in the proposed extensibility/scriptability architecture.

4.2.5 Converting Grease into a .NET accessible assembly

The need to integrate the synthesizer functionality with a rich, scriptable[18] user-interface motivated the conversion of the Grease C++ library into a .NET assembly (the .NET equivalent of a library file). Windows Presentation Foundation (WPF) was among the few graphics and windowing frameworks identified that was both rich in its feature set and supported scripting and extension at *run-time*. WPF can be programmed in any language that has a compiler that targets the .NET Common Language Runtime (CLR).

Since the Microsoft C++ compiler supports the automatic compilation of existing C++ code to target the CLR, a first attempt was made to build Grease from the ground up as a managed assembly. Though the build was successful with a few changes to Grease and STK, the performance degraded audibly. To remedy this, a set of wrapper classes was written for the synthesizer class alone that marshalled data from CLR client code to the C++ Grease/STK code. Thus, the Grease code was retained as a native/Win32 library

[18]There are two kinds of scripting involved in this discussion. One involves scripting the *synthesizer*. The other involves scripting the *user-interface*. It is currently challenging to accomplish both with a single scripting language. Grease script is used to script the synthesizer and IronPython is used to script the UI.

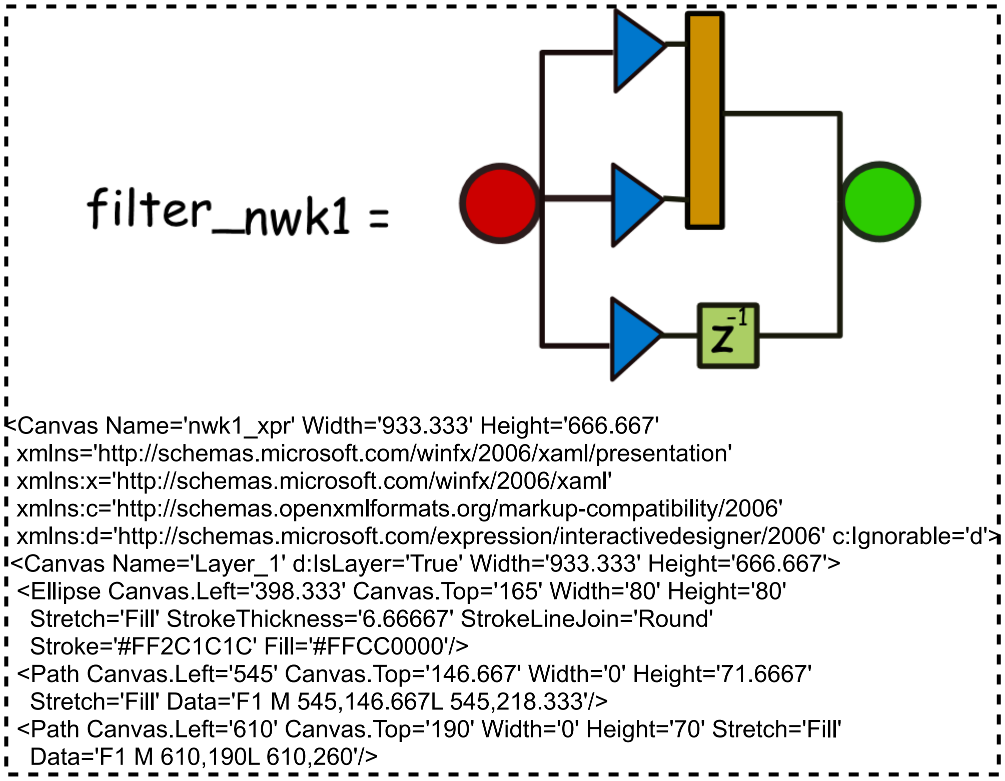
and a wrapper that transformed calls from CLR code to the C++ code was written. This resulted in absolutely no audible degradation of performance. The synthesizer component is now programmable in the entire spectrum of .NET languages, including C#, Visual Basic and IronPython. The position of the Grease/CLR component in the proposed application architecture is illustrated in Figure 4.13. This retains all synthesis and sequencing code in C++ and pushes only the user-interface and scripting functions to the 'managed' or CLR space, striking a balance between flexibility on the one hand and performance on the other.

4.3 Experiments with WPF

Windows Presentation Foundation was identified as representative of the state of the art in user-interface and graphics technology. Its rich 2D and 3D capabilities include the ability to use arbitrary graphics as user-interface elements (with full hit testing and content/composition support). Graphic design tools like Microsoft Expression Graphic Designer (for graphical element design), Microsoft Expression Interactive Designer (for UI behavior design), Maya, Zam 3D, and many other third-party[19] tools are available that support the creation of professional-grade 2D and 3D graphics that can be exported to XML Application Markup Language (XAML), Microsoft's hierarchical object serialization format.

Initial experiments with XAML-compatible tools included generating and designing various UI and visual language elements with Expression Graphic Designer. UI behaviors were implementable in a simple manner by exporting the XAML generated by Expression Graphic Designer into Expression Interactive Designer. The UI front end could simply use synthesizer elements by including a reference to the Grease managed assembly. Figure 4.14 shows a mock-up of a signal-flow graph designed in Graphic designer. The tool generates XAML that can be transformed at build time using the MS-Build tool, or at run-time using the dynamic XAML deserializer as described in Section 4.3.1. The experiments confirmed the possibility of a) Hit testing, drag/drop and other interactivity for arbitrary graphic elements; b) Composing complex visual elements by nesting elements; c) Adding interactivity at build time using the designers, and at run-time using dynamic script. This was

[19]Here by third-party is meant organizations other than Microsoft offering support for Microsoft technologies and tools.



```
<Canvas Name='nwk1_xpr' Width='933.333' Height='666.667'
xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'
xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml'
xmlns:c='http://schemas.openxmlformats.org/markup-compatibility/2006'
xmlns:d='http://schemas.microsoft.com/expression/interactivedesigner/2006' c:Ignorable='d'>
<Canvas Name='Layer_1' d:IsLayer='True' Width='933.333' Height='666.667'>
<Ellipse Canvas.Left='398.333' Canvas.Top='165' Width='80' Height='80'
  Stretch='Fill' StrokeThickness='6.66667' StrokeLineJoin='Round'
  Stroke='#FF2C1C1C' Fill='#FFCC0000'/>
<Path Canvas.Left='545' Canvas.Top='146.667' Width='0' Height='71.6667'
  Stretch='Fill' Data='F1 M 545,146.667L 545,218.333'/>
<Path Canvas.Left='610' Canvas.Top='190' Width='0' Height='70' Stretch='Fill'
  Data='F1 M 610,190L 610,260'/>
```

Fig. 4.14 A mock-up of visual elements for signal-flow prototyped using a graphical design tool (1) and an extract from the generated XAML markup (2).

followed by a complete prototype that involved building a visual element that validated the idea of boot-strapping the authoring environment using increasingly complex components with visuals and behavior. The manner in which this process could be combined with the model-based design methodology described in Section 4.1 is discussed further in Section 5.1.

4.3.1 Embedding a XAML + IronPython-based Scripting and Extensibility Engine

Figure 4.15 shows the use-case that motivated the prototype involving embedding a XAML markup + IronPython-based scripting and extension engine. New types of synthesis and sequencing objects often necessitate the construction of new interactive visual editors that can be used by end-users to a) construct objects of the type and b) manipulate the ob-

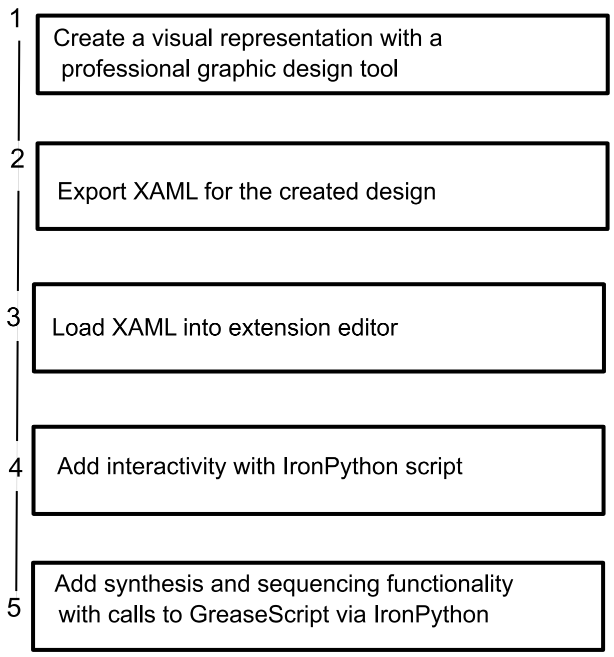

1
Create a visual representation with a professional graphic design tool
2
Export XAML for the created design
3
Load XAML into extension editor
4
Add interactivity with IronPython script
5
Add synthesis and sequencing functionality with calls to GreaseScript via IronPython

Fig. 4.15 Steps involved in writing a new visual extension

jects at run-time. Dynamic languages have been a popular choice for implementing user-interface dynamics. An example is Tcl/Tk [Ousterhout, 1994]. In particular, new code for user-interface behavior can be generated by the script itself using meta-programming techniques. The process of scripting a running application using a language like Python involves exposing all the program variables to an embedded Python interpreter. This traditionally involved running a wrapper generator to generate Python callable versions of all functions and types (Section 3.2). Much of this is automated by the .NET framework. Since the Grease synthesizer is implemented in native C++, it is manually exposed as a managed assembly (Section 4.2.5). The designer used to generate the user-interface generates C# code, which builds into a .NET assembly. The IronPython engine leverages the meta-data included in these managed assemblies to automatically bind symbols in the IronPython interpreter to variables in the .NET assembly that creates it (the engine). It uses the reflection features of the .NET framework to generate Common Intermediate Language (CLI) byte-code corresponding to methods in the IronPython script, *while the application is running*. This byte-code is then *compiled* by the Just In Time .NET compiler to machine code, also while the application is running. This results in a rather efficient[20] variety of end-user scripting and extensibility that integrates seamlessly with the originally compiled portion of the application as shown in Figure 4.16. The process can be used to boot-strap more complex environments, by using the visual designers to *generate* other visual designers.

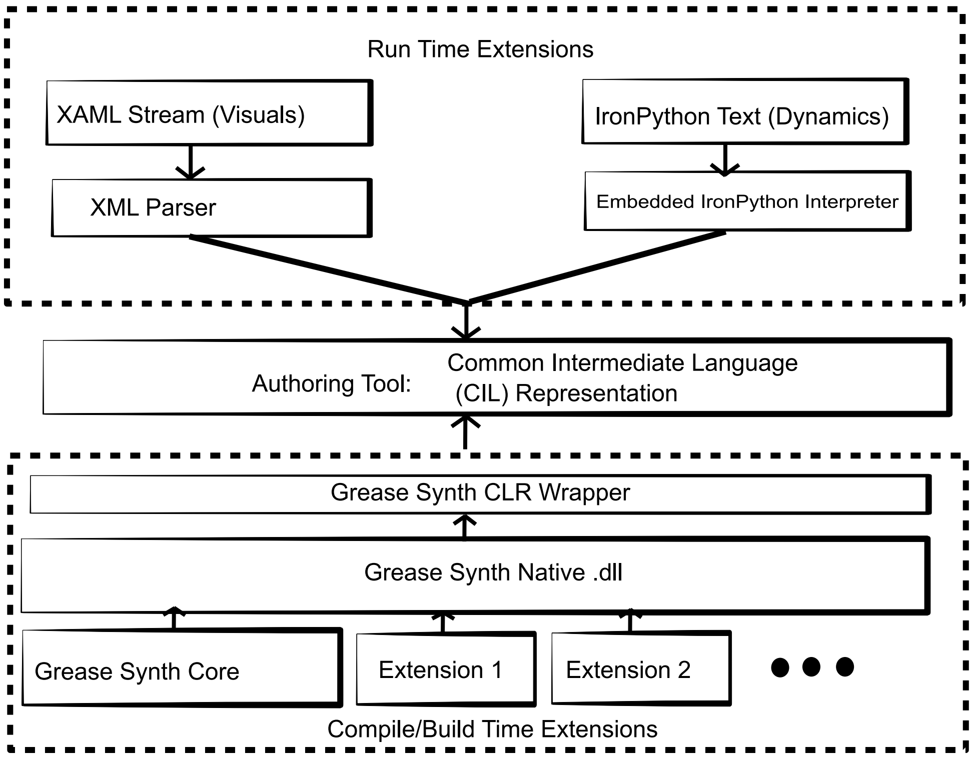


Fig. 4.16 Run Time and Build Time extensions to an Authoring Tool

4.3.2 Prototyping an interactive guitar chord editor using the scripting facility

This section explores the process of using the embedded scripting engine to prototype a visual component for manipulating a set of synthesis and sequencing objects. The component adds itself to a core authoring environment that provides a scripting window and access to the synthesizer. The example relates to representing chords meant to be played with a six stringed guitar. The only way to represent this in Grease script syntax would be as a string of six integers indicating the fret positions on the respective strings. A chord

[20][Hugunin, 1997a] includes comparison between traditional Python interpreters and IronPython.

is visualized with a guitar fret-board, with dots indicating the fretted positions. As a first step towards a language that translates the visual representation to the underlying Grease messages whenever it is encountered, it is useful to build a self-contained and 'nest-able'[21] component. The steps involved in the process are:

Step-I The graphical elements that represent the new type are designed using the WYSIWYG interface provided by Expression Graphic Designer as shown in Figure 4.17. Although the current representation doesn't involve animate-able portions in the syntax, this is allowed by WPF and supported by the designers. The grid represents the first four frets of a six stringed guitar. The horizontal lines represent the strings and the vertical lines represent the frets.

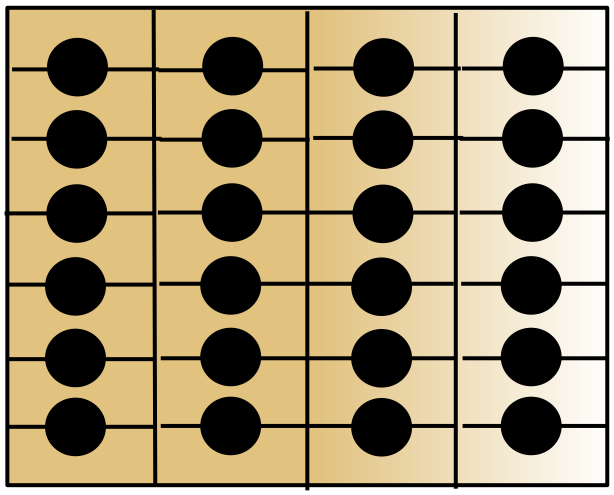


Fig. 4.17 Designing the visual elements for the chord-editor

Step-II The XAML for the visual elements is exported from the designer to the clip-board[22]. This is then pasted onto the scripting window in the authoring environment. The XAML representation is human-readable. Inspecting the representation gives hints as to the kind of meta-program that can be developed

[21]The nesting here is used in a visual sense. Any WPF element that conforms to the System.Windows.UIElement type can be nested inside another System.Windows.UIElement type according to the layout rules specified by the type of the container element.

[22]Prior to Step 2, the generated XAML is augmented with a button declaration via Expression Interactive Designer. The button is intended for articulating the chord representation.

to add dynamics to the component. Each fret on a single string is represented by a line-segment element. The line-segment elements are given identifiers in the XAML representation to reflect the corresponding string number and fret number. For example, the fifth fret on the second string is given the identifier '_52'. The dots are given similar identifiers.

Step-III An IronPython meta-program is written that exploits the structure in the identifiers to generate IronPython functions that handle MouseDown events on the line-segments and the dots. The functions generated by the IronPython meta-program assume the existence of a reference to the Grease synthesizer. The meta-program is entered into the IronPython window in the authoring environment.

```
for string_no in range(1,7):
    for fret_no in range(1,5):
        ellipse_id = repr(string_no) + repr(fret_no)
        ellipse_ref = "dot_" + ellipse_id
        line_ref = "_" + ellipse_id
        function_name1 = "on_" + ellipse_id + "_Click"
        function_decl1 = "def" + function_name1 + "(*args):\n"
        function_body1 = " " + ellipse_ref + ".Opacity = 0\n"
        exec(function_decl1 + function_body1)
        exec(ellipse_ref + ".MouseUp += " + function_name1)
        function_name2 = "on_" + line_ref + "_Click"
        function_decl2 = "def " + function_name2 + "(*args):\n"
        function_body2 = " " + ellipse_ref + ".Opacity = 1\n"
        exec(function_decl2 + function_body2)
        exec(line_ref + ".MouseUp += " + function_name2)
```

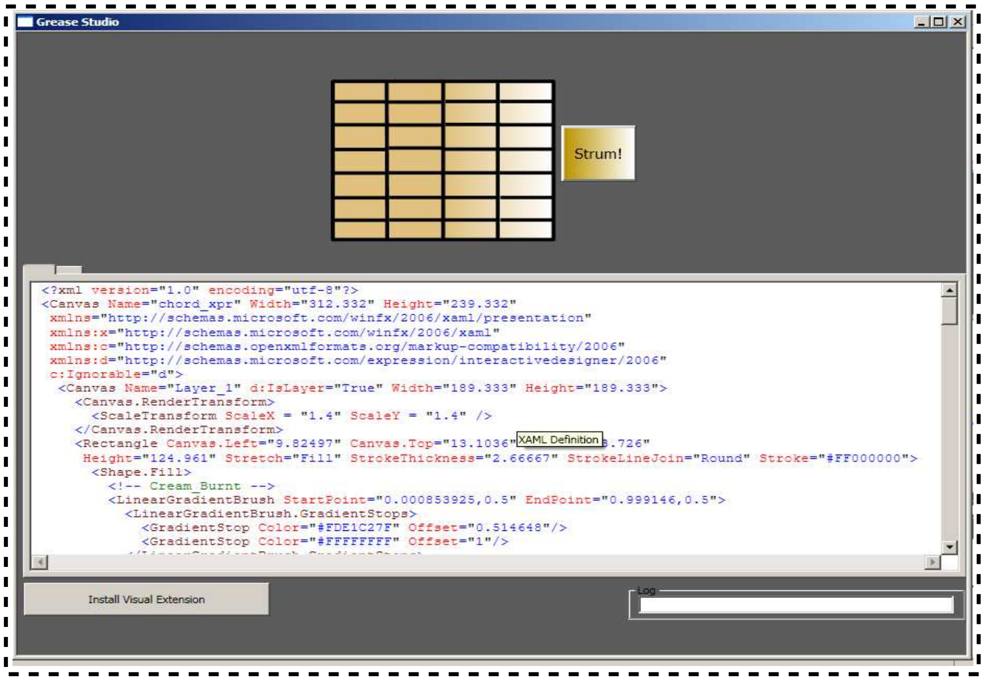
Fig. 4.18 Python meta-program for generating the dynamics.

Step-IV Clicking on 'Install new Visual Component' initiates the component installation process. The XAML is streamed through the .NET XML parser, and an in-memory representation of the entire visual-tree is recreated. The meta-program is executed via the IronPython engine, which attaches event-handlers to the visual-tree. When the process is completed, the component is rendered inside the parent window and is fully functional, as shown in Figure 4.19.

Step-V The meta-program also generates GreaseScript calls via the IronPython reference to the synthesizer object.

The behavior of the environment provides for interactive editing of tabulature[23]-like representations of guitar chords. Some basic patterns were experimented with to test the

[23]Guitar tabulature is very similar to the notation used here and is a common way of sharing chord representations among amateur musicians.



```
<?xml version="1.0" encoding="utf-8"?>
<Canvas Name="chord_xpr" Width="312.332" Height="239.332"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:c="http://schemas.openxmlformats.org/markup-compatibility/2006"
 xmlns:d="http://schemas.microsoft.com/expression/interactivedesigner/2006"
 c:Ignorable="d">
  <Canvas Name="Layer_1" d:IsLayer="True" Width="189.333" Height="189.333">
    <Canvas.RenderTransform>
      <ScaleTransform ScaleX = "1.4" ScaleY = "1.4" />
    </Canvas.RenderTransform>
    <Rectangle Canvas.Left="9.82497" Canvas.Top="13.1036"          .726"
     Height="124.961" Stretch="Fill" StrokeThickness="2.66667" StrokeLineJoin="Round" Stroke="#FF000000">
      <Shape.Fill>
        <!-- Cream_Burnt -->
        <LinearGradientBrush StartPoint="0.000853925,0.5" EndPoint="0.999146,0.5">
          <LinearGradientBrush.GradientStops>
            <GradientStop Color="#FDE1C27F" Offset="0.514648"/>
            <GradientStop Color="#FFFFFFFF" Offset="1"/>
```

Fig. 4.19 The new visual component rendered inside the parent environment.

interactivity. The creation of patterns was fairly straightforward. Clicking a string-segment between two frets turns on the dot in the segment, effectively fretting the string segment between the frets. Clicking on the dot un-frets the segment, removing the dot. These behaviors were generated by the IronPython meta-program and dynamically associated with the visuals. Clicking on the 'Strum!' button invokes the synthesizer. Figure 4.20 shows the results of creating some basic chord patterns[24]. The IronPython script generated for the visual component constructor automatically invokes GreaseScript to create a 'Guitar' object via the synthesizer reference available to the IronPython engine. The 'Guitar' component, in turn is a build-time extension to Grease that internally delegates its functionality to the GuitarString objects that in turn generate samples using STK's 'Plucked' class. The signal-flow graph representing the connection between the Guitar objects with the Mixer objects is also created by invoking GreaseScript through the IronPython synthesizer reference.

[24]The guitar strings are assumed to be in standard tuning.

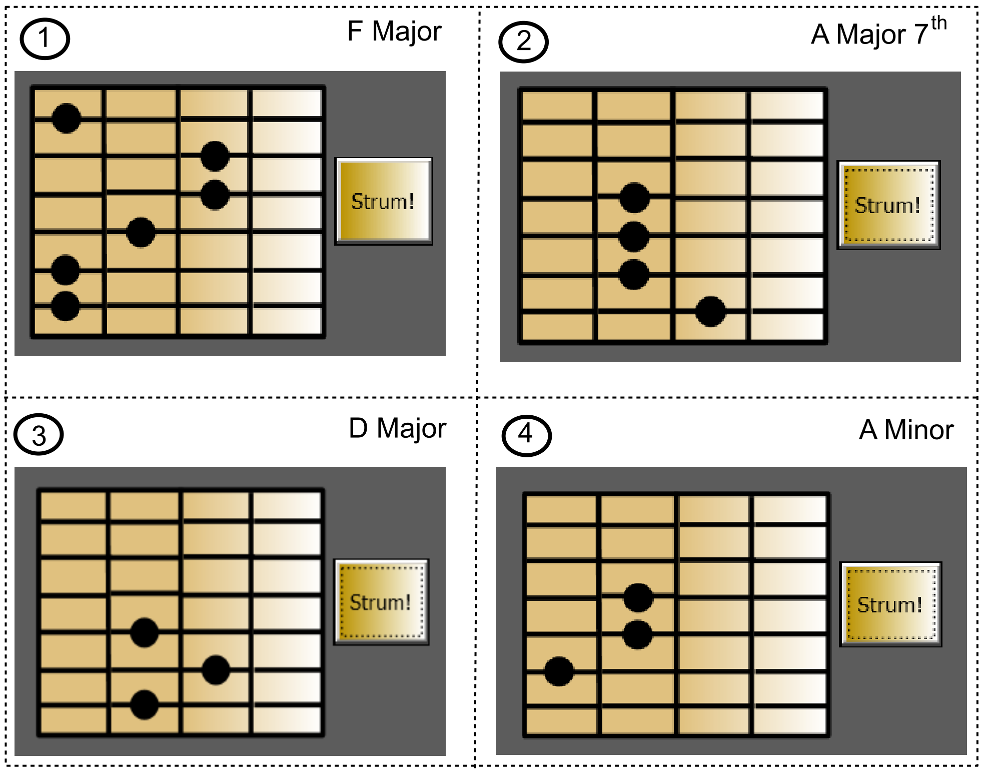


Fig. 4.20 Simple chords being edited in the installed interactive chord editor component.

4.3.3 Possible Enhancements to the Extension Prototype

It is straightforward to make the fine-grained functionality of the visual editors available to the end-user programmatically via the IronPython interface. This could be used, for example, to allow the end-user to create macros that strum the chord with a specific sequence of chords in a particular rhythm. Taking the idea of boot-strapping a more complex environment further, the same design could be applied to create a visual chord sequencer using a StackPanel element that internally delegates the visual and audio rendering, as well as interactivity of the chords to the Chord components that it holds. These are not explored in the demonstrations. The IronPython meta-program currently generates a set of functions. It could be made to generate a Python class-declaration for a meta-constructor, making the component even more modular. In all, the functionality provided by the base environment and the scripting engine opens up the possibility of creating future functionality in the authoring environment as self-contained extensions with rich visuals and interactivity. It also opens up the possibility of creating meta-editors, that allow end-users to generate other editors, much along the lines of Elody, OpenMusic, AtoM^3 and GME.

Chapter 5

Conclusions and Future Work

This chapter tries to re-situate, in retrospect, the ideas and results from previous chapters in the context of the thesis investigation and the larger context of computer-music research. The technical and artistic implications of the design experiments are summarized and possible directions for future research and development are outlined.

5.1 Conclusions

As introduced in Chapter 1, the aim of the thesis was to evaluate technical advances in various software domains, and to use this evaluation to update design methodology for the creation of music authoring tools that balanced the benefits of extensible special-purpose (Category A) tools and specializable general-purpose (Category B) frameworks. This consists, in essence, of supporting the related features of end-user extensibility, customization and personalization. While this is a considerably challenging goal, it is a goal shared by many other domains and they have evolved re-usable components for realizing these goals over the years.

5.1.1 Historical perspective

Chapter 2 surveyed the history of computer music software and identified components of authoring tools that were common to traditional synthesis languages, libraries and studio software. In particular, audio synthesis programming languages can be seen as Category B frameworks for music authoring. These languages themselves have evolved at least two distinguishable components, one specialized for the task of describing sound design and processing, and another specialized for the task of describing 'control' sequences - sequencing. Recording and studio-based software, on the other hand, has traditionally evolved as a package of Category A tools that are each designed to solve a specific task.

While Category A tools like guitar-synthesizer modules and drum sequencers tend to be better suited, in terms of ease of use, to the particular usage contexts for which they were developed, they tend to be inflexible to possible variations in usage contexts that may be required by particular groups of end-users. Category B frameworks like audio synthesis programming languages, on the other hand, are designed to express solutions to a very large class of problems. While they are very flexible, it is hard for domain-experts without significant programming experience to use them.

5.1.2 Identification of re-usable tools from other domains.

Two classes of 'meta'-frameworks surveyed in Chapter 3 were identified as a fertile starting point for design investigations. The goal was to assimilate the problem of bridging Category B frameworks and Category A tools, into well-developed, existing conceptual and implementation frameworks for solving this problem. The frameworks have evolved in response to the fact that this problem is ubiquitous across domains.

Visual Modeling Tool Generators

The first of these 'meta'-frameworks has evolved from the modeling and simulation community and is a design methodology called 'model-driven architecture'. In Chapter 4, a pragmatic approach was taken to identify possible ways of re-using the tools developed by this community to the problem of developing authoring tools. Visual environment generators that facilitated the automatic generation of context-specific, restrictive, visual modeling environments for certain aspects of music authoring were explored. These environments are generated from formal descriptions of the environments called 'meta-models'. They require a formal analysis of the usage context in advance. Results of experiments that involved generating simplified prototypes of sound-design and sequencing environments using the GME tool were documented.

Scriptability, Scripting Language Engines, and Informal Notations

The second of these 'meta'-frameworks are re-usable 'scripting' language engines. These are Category B frameworks that facilitate the process of exposing, to the end-user, the underlying Category B components used to build a Category A tool. The motivation for including such a feature is that Category A tools like drum sequencers are often (but not always) themselves built with more general purpose components like audio synthesis programming languages. By giving the end-user optional access to this facility, usage-scenarios that were not conceived by the tool designer, but realize-able with the underlying components used to build the tool, may be constructed by the end-user herself. Since the intention is to support un-anticipated usage-scenarios, the dynamic nature of scripting languages is actually well suited to the task. Chapter 4 explored the process of embedding the IronPython engine to script the user-interface of an authoring tool. It explored the process of making a scriptable audio synthesizer. It also explored the utility of both the synthesizer and UI scripting facilities by demonstrating their use to build extensions to the synthesizer and the user-interface, respectively.

GUI and Graphics Toolkits

This thesis also investigated the trade-off between visual and textual representations. Formalized notations like those used by hierarchical signal-flow, state-charts and petri-nets have a well defined syntax and semantics. These are useful to describe well-defined sound synthesis and sequencing processes. On the other hand, context-specific or artist-specific processes cannot be anticipated by tool designers, and these are best served by visual versions of Category B frameworks like dynamic scripting languages. Visual scripting is popular in many other craftsmanship-intensive domains like visual design, narrative design and game design. Even though *many aspects* of these domains are very much amenable to formal design and analysis, a large part of the *creative* process involves using the authoring tool in unanticipated contexts. Visual scripting facilities provide for end-users to tailor the authoring tool to function as required in these undefined contexts[1], using a friendlier version of text-based Category B scripting frameworks. Chapter 3 explored some visual scripting solutions developed in the video game industry. It also explored Category B graphics and

[1]The context is defined in an operational sense as 'the context enabled by the modification made by User X', but this does not qualify as a formal definition of the context.

GUI frameworks that could be used to build visual scripting solutions. These were explored further in Chapter 4 in the context of Windows Presentation Foundation related demonstrations, which, along with the reflection facilties, compiler front-ends, automated interpreter engine bindings and integration with professional graphic design tools available for the .NET framework, constitutes an excellent framework for the design of hybrid, textual/visual scripting languages. Chapter 4 explored the manner in which dynamic visual components may be designed and installed at run-time and hierarchically composed into an interactive visual scripting facility. The 'content-composition' feature of Windows Presentation Foundation, along with the dynamic code-generation features of IronPython on the .NET platform made this possible. The tool support provided by the platform to mix these components with efficient C++ implementations of a synthesizer further increased the value of the platform[2].

5.2 Future Work

5.2.1 Integration of Bottom-Up and Top-Down Approaches

Chapters 3 and 4 presented both pre-designed, formally modeled approaches, also known as 'top-down' approaches to building context-specfic tools and on-the-fly, informal, interactive and prototype-driven approaches, also known as 'bottom-up' approaches. They recognized the importance of both these approaches, which stems from the fact that better tool support can be built using pre-designed, top-down approaches, when applicable. On the other hand, the creative process inherently demands the application of the authoring tool in unanticipated contexts and bottom-up approaches involving dynamic languages and scripting are especially relevant here. It is possible to combine these approaches by using a scripting-based solution itself to implement a top-down modeling tool. This would allow for partial solutions expressed with informal descriptions to be stream-lined, in the long run, into well-defined context-specific design tools.

[2]It is important to note that these features may be available in upcoming versions of many other platforms.

5.2.2 Leveraging Dynamics and the 3D Medium

The visual language design explorations were concerned with 2D demonstrations with limited interactivity. These ideas translate to the 3D medium with interactive animations. Such a visual language mapping dynamic elements to other dynamic[3] elements may blur the boundaries between computer-based prototyping/design and computer-based performance. Thus, the act of a musician in the role of a composer exploring musical possibilities is concretized by the visual interface to the prototyping tool and made available to an audience, transforming the compositional/prototyping activity into an improvisational/performance activity. The WPF/IronPython approach scales to many 3D scenarios by using XAML exported from 3D design tools like Maya and Zam 3D. For richer experiences, the environment can be rebuilt to use a gaming engine like Unreal Engine 3.

5.2.3 Using alternative input devices for language interaction

Along the same lines, it is possible to extend this notion of an interactive language and environment to use non-traditional input devices better suited to the interaction. A chord editor, for example, could be manipulated with a midi-guitar or keyboard. A hierarchical *space* of related chords, on the other hand, could be navigated by a combination of arm and palm gestures, captured using devices like the Polhemus Liberty and the Vicon. A calculus for combining gestures in a manner that allows the expression of sequencing and sound processing could be built by the hierarchical composition of visual editors and input interaction behaviors.

5.2.4 Extending the architecture to enable web deployment

The most practical enhancement to the entire architecture would be to facilitate the construction and deployment of new funtionality over the web. This would require extension-representations to be stored in a structured storage like a traditional database and to provide for versioning and search features for the database of extensions. A well implemented versioning scheme would allow for the collective/community-driven development of genre-specific and style-specific extensions to the authoring tool core, which could be downloaded or executed directly over the web. Searchability would ensure the discovery

[3]Here ‘dynamic’ does not mean ‘dynamic language’ in a programming-language sense. Rather, it means dynamic visual elements (animation) in the syntax of the language.

of useful community authored artifacts by other users in search for similar funtionality. Such functionality could possibly be realized by re-using existing architecture for Massively Multiplayer Online Role Playing Games (MMORPGs).

5.3 Artistic and Philosophical Significance

5.3.1 Personalize-able Infrastructure - The Creation of a New Artistic Medium

Artistic creation, in general, defies credible analysis from an engineering perspective and, more so, from a scientific or even rational perspective. It is often regarded as the proverbial black-hole of scientific and rational enquiry into human thought and existence. Turning the tables around, [Becker and Eckel, 2003] refer to the epistemological and metaphysical function once attibuted to works of art, "in so far as truth[4] could appear in a given work (of art)". However, they also refer to the shift in modern theories of art towards approaches oriented to affirm the utilitarian, explanatory stance of the empirical sciences.

Even from such a purely utilitarian stance, the direct relationship between the value of a work of art and the technique or effort required to produce it, has paradoxical interpretations in relation to the applicability of technology to the arts. Theodore Adorno famously objected to the commoditization and mechanization of musical art in his 1932 essay, 'On The Social Situation of Music' [Adorno, 1932]. A review of Adorno's book, 'Essay's on Music' [Adorno, 2002; Schroeder, 2003] quotes him as saying about his reading of Wagner that he

> ... realized without reservation that the binding, truly general character of musical works of art is to be found, if at all, only through the medium of their particularity and concretion, and not by recourse to any kind of general types.

Also referring to Adorno, [Becker and Eckel, 2003] note that

> He pointed out that [the][5] epistemological objective of art always was diametrically opposed to that of technique: While technique aimed at a general ordering and a global control, art - in contrast - revealed individual and particular aspects of the

[4]The epistemological funtion being related to the central nature of the notion of truth to human enquiry.
[5]Parenthetical remarks are additions by the current author.

> human existence. In artistic expression, new views of the world which contrast with general paradigms of technology (science) could be experienced and articulated. In its lack of purpose, art aimed not at making something available, but rather, in its very distance from the attitudes proper to technique, it referred to alternative modes of individual and cultural ways of living.

The empowerment of artists themselves with the ability to create tools that address the needs of new artistic contexts defined by them was the motivating vision behind the investigations in the current work. While the investigations did not quite get to the point of demonstrating a clear path to the realization of this long-term goal, the technical review and design experiments point strongly towards the readiness of the state-of-the-art for realizing such transformative, malleable and personalize-able media. They add to existing technical evidence in the music-technology literature for a possibly paradoxical resolution of the conflict between technological control and individual expression, with the empowerment of individual expression by technology itself.

References

Adorno, T. L. W. (1932). Zur gesellschaftlichen Lage der Musik (On the Social Situation of Music). *Zeitschrift fr Sozialwissenschaft (Journal for Sociology)*.

Adorno, T. L. W. (2002). *Essays on Music*. University of California Press.

Agha, G. and Hewitt, C. (1987). Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. *Research Directions in Object-Oriented Programming*, pages 49–74.

Agon, C., Assayag, G., Delerue, O., and Rueda, C. (1998). Objects, Time and Constraints in OpenMusic. In *Proceedings of the International Computer Music Conference (ICMC)*, Ann Arbor, Michigan.

Agrawal, A. (2003). Graph Rewriting And Transformation (GReAT): A Solution For The Model Integrated Computing (MIC) Bottleneck. *Automated Software Engineering.*, 00:364.

Amatriain, X., Ammi, P., and Ramirez, M. (2002). CLAM, Yet Another Library for Audio and Music Processing. In *Proceedings of 17th Annual ACM Conference on Object Oriented Programming, Systems, Languages and Applications.*, Seattle, WA, USA.

Amatrian, X. (2005). *An Object-Oriented Metamodel for Digital Signal Processing with a focus on Audio and Music*. PhD thesis, Universitat Pompeu Fabra.

Arum, P. and Amatriain, X. (2005). CLAM, An Object-Oriented Framework for Audio and Music. In *Proceedings of 3rd International Linux Audio Conference*, Karlsruhe, Germany.

Assayag, G., Agon, C., Fineberg, J., and Hanappe, P. (1997). An Object Oriented Visual Environment For Musical Composition. In *Proceedings of the International Computer Music Conference (ICMC)*, Thessaloniki, Greece.

Assayag, G., Agon, C., and Stroppa, M. (2000a). High Level Musical Control of Sound Synthesis in OpenMusic. In *Proceedings of the International Computer Music Conference (ICMC)*, Berlin.

Assayag, G., Agon, C., and Stroppa, M. (2000b). High Level Musical Control of Sound Synthesis in OpenMusic. In *Proc. ICMC*, Berlin.

Assayag, G., Rueda, C., Laurson, M., Agon, C., and Delerue, O. (1999a). Computer Assisted Composition at Ircam: PatchWork & OpenMusic. *Computer Music Journal*, 23(3).

Assayag, G., Rueda, C., Laurson, M., Agon, C., and Delerue, O. (1999b). Computer Assisted Composition at Ircam: PatchWork & OpenMusic. *Computer Music Journal*, 23(3).

Beazley, D. M. (2003). Automated scientific software scripting with SWIG. *Future Gener. Comput. Syst.*, 19(5):599–609.

Becker, B. and Eckel, G. (June 2003). On the Relationship between Art and Technology in Contemporary Music. *Electronic Journal For Philosophy*.

Borchers, J. and Max, M. (1998). Design Patterns for Interactive Musical Systems. *IEEE MultiMedia*, 5(3):36–46.

Boswell, D., King, B., Oeschger, I., Collins, P., and Murphy, E. (2006). *Mozilla as Platform*. The Mozilla Foundation.

Bresson, J., Agon, C., and Assayag, G. (2005). Visual and adaptive constraint programming in music. In *Proc. of the 10th Brazilian Symposium on Computer Music*, Belo Horizonte.

Burke, P. (1998). A Real-time Synthesis API for Java. In *Proceedings of the International Music Conference*, pages 252–255, Ann Arbor, Michigan.

Chaudhary, A., Freed, A., and Wright, M. (1999). An Open Architecture for Real-Time Audio Processing Software.

Cook, P. and Scavone, G. (1999). The Synthesis ToolKit (STK). In *Proc. of the International Computer Music Conference*, Beijing.

Costagliola, G., Delucia, A., Orefice, S., and Polese, G. (December 2002). A Classification Framework to Support the Design of Visual Languages. *Journal of Visual Languages and Computing*, 13(6):573 – 600.

Curry, B. (2006). BikeCAD. [Online; accessed 6-July-2006].

Dannenberg, R. (1992). The Implementation of Nyquist, a Sound-Synthesis Language. *Computer Music Journal*, 21(3):71–82.

Dannenberg, R. (2002). A Language for Interactive Audio Applications. In *Proc. of the Int. Computer Music Conf.*

Dannenberg, R. and Brandt, E. (1996). A flexible realtime software synthesis system.

Danvy, O. and Filinski, A. (1990). Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 151–160, New York, NY. ACM.

Didkovsky, N. (2004). Java Music Specification Language, v103 update. In *Proc. of the Int. Computer Music Conf.*

Eclipse Foundation (2001). Eclipse Platform Technical Overview. [Online; accessed 29-July-2006].

Ehrig, K., Ermel, C., H, S., and Taentzer, G. (2005). Generation of visual editors as eclipse plug-ins. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 134–143, New York, NY, USA. ACM Press.

Epic Games (2006). Unreal Technology Information Page. [http://www.unrealtechnology .com/html/technology/ue30.shtml Online; accessed 4-August-2006].

Farbood, M. M., Pasztor, E., and Jennings, K. (2004). Hyperscore: A Graphical Sketchpad for Novice Composers. *IEEE Comput. Graph. Appl.*, 24(1):50–54.

Fober, D. (1994). Real-time Midi data flow on Ethernet and the software architecture of MidiShare. In *Proc. of the Int. Computer Music Conf.*

Fober, D., Letz, S., and Orlarey, Y. (1997). L'environnement de composition musicale Elody.

Freed, A. (1995). Improving Graphical User Interfaces for Computer Music Application. *Computer Music Journal*, 19(1):4 – 5.

Furse, R. (2006a). LADSPA Home Page. [Online; http://www.ladspa.org/; accessed 27-Aug-2006].

Furse, R. (2006b). LADSPA XML GUI DTD. [Online; http://www.ladspa.org/ ladspaxml-gui.dtd; accessed 27-Aug-2006].

Garrett, J. J. (2005). Ajax: A New Approach to Web Applications. [Online <http://www.adaptivepath.com/publications/essays/archives/000385.php>; accessed 30-July-2006].

Grame (2000). *MidiShare Developer Documentation v.1.80 Release Notes.*

Greenfield, J. and Short, K. (2003). *Developing Domain Specific Languages.* Wiley Publishing, Inc.

Greenfield, J., Short, K., Cook, S., Kent, S., and Crupi, J. (2004). *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, page 281.

Hugunin, J. (Mar. 24, 1997a). IronPython: A fast Python implementation for .NET and Mono. In *Proceedings of PyCON 2004*, Washington, DC. Python Software Foundation.

Hugunin, J. (Oct. 14-17, 1997b). Python and Java - The Best of Both Worlds. In *Proceedings of the 6th International Python Conference*, San Jose, California. Corporation for National Research Initiatives (CNRI).

Ierusalimschy, R., de Figueiredo, L. H., and Filho, W. C. (1996). Lua - an extensible extension language. *Softw. Pract. Exper.*, 26(6):635–652.

Institute for Software Integrated Systems (2006). *GME 6 User's Manual. Version 6.0.*

McCartney, J. (2006). SuperCollider Screen Shot. [Online; accessed 9-July-2006].

Minsky, M. (1986). *The Society of Mind.* Simon & Schuster, Inc., New York, NY, USA.

Object Management Group (2006). Object Management Group Home Page. [Online; http://www.omg.org/; accessed 26-Aug-2006].

Ousterhout, J. K. (1994). *Tcl and the Tk Toolkit.* Addison Wesley.

Porcaro, N., Jaffe, D., Scandalis, P., Smith, J., Stilson, T., and Duyne, S. V. (1998). SynthBuilder:A Graphical Rapid-Prototyping Tool for the Development of Music Synthesis and Effects Patches on Multiple Platforms. *Computer Music Journal*, 22(2):35–43.

Puckette, M. (1991). Combining Event and Signal Processing in the MAX Graphical Programming Environment.

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

Rosenboom, D. and Polansky, L. (1985). HMSL: A real-time environment for formal, perceptual and compositional experimentation. In *Proc. of the Int. Computer Music Conf.*, Burnaby, B.C.

Rowe, R. (1991). *Machine Listening and Composing: Making Sense of Music with Cooperating Real-Time Agents.* PhD thesis, Massachusetts Institute of Technology.

Rowe, R. (2001). *Machine Musicianship.* MIT Press, Cambridge, MA, cloth edition.

Rowe, R. (2005). Personal Effects: Weaning Interactive Systems from MIDI. In *Proceedings of the Spark Festival.*

Scaletti, C. A. (1989). Kyma: an interactive graphic environment for object-oriented music composition and real-time software sound synthesis written in Smalltalk-80.

Schmerl, B. and Garlan, D. (2004). Acmestudio: Supporting style-centered architecture development. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 704–705, Washington, DC, USA. IEEE Computer Society.

Schroeder, S. (June 2003). Book Review, Essays on Music. *Essays in Philosophy*, 4(2).

Smart, J. (2006). About Julian Smart. [Online; http://www.anthemion.co.uk/julian.htm; accessed 29-July-2006].

Smith, J. O. (1996). Physical Modeling Synthesis update. *Computer Music Journal*, 20(2).

Stoy, J. (1977). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.*

Sun, X. (2005). Parallel DEVS modelling of Traffic in AToM^3. [Online http://msdl.cs.mcgill.ca/MSDL/people/simon/presentation/; accessed 1-Aug-2006].

Sussman, G. J. and McDermott, D. V. (1972). From PLANNER to CONNIVER–A genetic approach. In *Proceedings of the Fall Joint Computer Conference*, pages 1171 – 1179.

Tanaka, A. (2003). Composing as a Function of Infrastructure. *Surface Tension: Problematics of Site.*

Taube, H. (1991). Common Music: A music composition language in Common Lisp and CLOS. *Computer Music Journal*, 15(2).

TrollTech (2006). QT 4.1 Whitepaper. [Online; accessed 28-July-2006].

Truchet, C., Assayag, G., and Codognet, P. (2001). Visual and adaptive constraint programming in music. In *Proc. of the Int. Computer Music Conf.*, Havana, Cuba.

Vanderbilt University (2006). Institute for Software Integrated Systems - Welcome Page. [Online: http://www.isis.vanderbilt.edu/; accessed 13-August-2006].

Vangheluwe, H. and de Lara, J. (2002). XML-based modeling and simulation: meta-models are models too. In *WSC '02: Proceedings of the 34th conference on Winter simulation*, pages 597–605. Winter Simulation Conference.

Vangheluwe, H. and de Lara, J. (2003). Foundations of multi-paradigm modeling and simulation: computer automated multi-paradigm modelling: meta-modelling and graph transformation. In *WSC '03: Proceedings of the 35th conference on Winter simulation*, pages 595–603. Winter Simulation Conference.

Vercoe, B. (1990). Real-Time CSOUND: Software Synthesis with Sensing and Control. In *Proceedings of the 1990 ICMC*, pages 209–211, Glasgow. International Computer Music Association.

Virtools Inc (2006). Overview of Virtools Dev. [Online; http://www.virtools.com/ solutions/products/virtools_dev.asp; accessed 26-Aug-2006].

Wang, G. and Cook, P. (2004). The Audicle: A Context-Sensitive, On-the-fly Audio Progamming Environ/mentality. In *ICMC '04: Proceedings of the 2004 International Computer Music Conference*, Miami, FL, USA.

Wang, G., Misra, A., Kapur, A., and Cook, P. R. (2004). Yeah, ChucK it! => Dynamic, Controllable Interface Mapping. In *NIME '05: Proceedings of the 2005 conference on New interfaces for musical expression*, pages 196–199, Singapore. National University of Singapore.

Wikipedia (2006a). Compiler-compiler — Wikipedia, The Free Encyclopedia. [Online; accessed 11-August-2006].

Wikipedia (2006b). Csound — Wikipedia, The Free Encyclopedia. [Online; accessed 6-July-2006].

Wikipedia (2006c). Digital Audio Workstation — Wikipedia, The Free Encyclopedia. [Online; accessed 24-July-2006].

Wikipedia (2006d). FutureSplash Animator — Wikipedia, The Free Encyclopedia. [Online; accessed 30-July-2006].

Wikipedia (2006e). Game designer — Wikipedia, The Free Encyclopedia. [Online; accessed 4-August-2006].

Wikipedia (2006f). Game Engine — Wikipedia, The Free Encyclopedia. [Online; accessed 24-July-2006].

Wikipedia (2006g). Logic Pro — Wikipedia, The Free Encyclopedia. [Online; accessed 24-July-2006].

Wikipedia (2006h). Model-Driven Architecture — Wikipedia, The Free Encyclopedia. [Online; accessed 12-August-2006].

Wikipedia (2006i). MUSIC-N — Wikipedia, The Free Encyclopedia. [Online; accessed 6-July-2006].

Wikipedia (2006j). Philippe Manoury — Wikipedia, The Free Encyclopedia. [Online; accessed 9-July-2006].

Wikipedia (2006k). Qt (toolkit) — Wikipedia, The Free Encyclopedia. [Online; accessed 29-July-2006].

Wikipedia (2006). Reference counting — Wikipedia, The Free Encyclopedia. [Online; accessed 14-August-2006].

Wikipedia (2006a). Standard Widget Toolkit — Wikipedia, The Free Encyclopedia. [Online; accessed 30-July-2006].

Wikipedia (2006b). Unified Modeling Language — Wikipedia, The Free Encyclopedia. [Online; accessed 11-August-2006].

Wilson, P. R. (1992). Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France). Springer-Verlag.

Zadel, M. (2006). A Software System for Laptop Performance and Improvisation. Master's thesis, McGill University.

Zadel, M. and Scavone, G. (2006). Different Strokes: a Prototype Software System for Laptop Performance and Improvisation. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 168–171, Paris, France.

Zbyszynski, M. and Freed, A. (September 2005). Control of VST Plug-ins using OSC. In *ICMC '05: Proceedings of the 2005 International Computer Music Conference*. International Computer Music Association (ICMA).