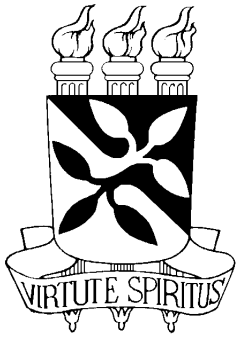

UNIVERSIDADE FEDERAL DA BAHIA

Desenvolvendo uma meta-linguagem para síntese sonora

Tese submetida ao

Programa de Pós-Graduação em Música

como pré-requesito para a obtenção do grau de

Doutor em Música

Pedro Ribeiro Kröger Junior

Salvador, Bahia

Janeiro, 2004

Omnia aliena sunt, tempus tantum nostrum est.

Seneca, *Epistulae morales ad Lucilium 1*

Sumário

Lista de Figuras

Lista de Exemplos

Agradecimentos

Ao meu orientador, Prof. Jamary Oliveira pela incansável ajuda, incentivo, e amizade. Igualmente ao Prof. Russell Pinkston com quem tive a oportunidade de trabalhar durante o período de doutorado-sanduíche na University of Texas at Austin.

A Alda Oliveira, pelo incentivo e carinho ao longo dos anos.

Aos amigos Ken e Bea Fincher pela inestimável ajuda quando morei em Austin, TX.

Aos amigos Ricardo Bordini, Pablo Sotuyo, Pedro Augusto, Wellington Gomes, Pete Moss, John Latto, e Young-Hwan Yeo pela amizade e disscussões intrigantes.

À CAPES que tornou esse trabalho financeiramente possível, tanto no Brasil quanto no exterior.

À minha família, especialmente minha mãe Dedy e minha irmã Laura pelas constantes ajudas e incentivos.

Um agradecimento especial não poderia deixar de ir para minha querida Mara, por coisas demais para listar aqui.

Aos amigos da Orquestra de Câmera da Emus-UFBA, Ângelo Rafael, e principalmente, os colegas de naipe Davi Cerqueira, Hugo Leonardo, e Renata D'Urso. Aos amigos que fizeram ou fazem música de câmera comigo, especialmente, Laura Jordão, Neemias e Arlene Santos, Ático Razera, e Dennis Leoni.

Lista de abreviaturas e definições de termos

CLM. *Common Lisp Music*, linguagem de síntese sonora desenvolvida por Bill Schottstaedt.

CXL. Abreviatura de *Csound* XML *Library*.

DSP. Abreviação em inglês para *Digital Signal Processing*.

DTD. Abreviação de *Document Type Definition*, definição de tipo de documento.

EMNML. Abreviação de *Extensible Music Notation Markup Language*, linguagem extensível de marcação para notação musical.

Front-Ends. Um programa, em geral gráfico, que utiliza a funcionalidade básica de outro programa.

Hack. No contexto usado nessa tese, um conserto rápido para algum problema, sem ser necessariamente a melhor solução. Para mais definições do termo, ver `http://www.catb.org/~esr/jargon/html/H/hack.html`.

MDL. Abreviação de *Music Description Language*, linguagem de descrição musical.

Meta-linguagem. Uma linguagem usada para definir ou descrever outra linguagem.

Opcode. Abreviatura em inglês de *operation code*, código de operação. Geralmente utiliza-se opcode em português, ainda que codop (código de operação) seja possível.

Parser. Um programa que lê código fonte e converte para código de objeto. Mesmo em português utiliza-se "parser" e "parsear".

SDML. Abreviação para *Standard Music Description Language*, linguagem padrão de descrição musical.

SVG. Abreviação de *Scalable Vector Graphics*, gráficos vetoriais escaláveis.

SWSS. Abreviatura de *SoftWare Sound Synthesis*, síntese sonora em *software*.

UG. Abreviação de Unidade Geradora.

VML. Abreviação de *Vector Markup Language*, linguagem de marcação vetorial.

Widget. Nas interfaces gráficas é um símbolo gráfico completo, como uma barra de rolagem.

Wrapper. Dado ou programa secundário que precede o principal de modo que ele possa rodar com sucesso.

XML. Abreviação de *eXtensible Markup Language*, linguagem de marcação extensível.

Resumo

A síntese sonora em *software* está intimamente ligada aos programas da família Music N iniciados pelo Music I em 1957. Apesar de seus méritos, como as unidades geradoras e a flexibilidade de uma linguagem de partitura, o Music N apresenta alguns problemas como limitações na reutilização de instrumentos, inflexibilidade de parâmetros, falta de linguagem gráfica, e normalmente apenas um paradigma para partituras.

Algumas soluções concentram-se em novas implementações da Music N, enquanto outras concentram-se na criação de ferramentas auxiliares como pré-processadores, e utilitários gráficos. Contudo as novas implementações em geral concentram-se em grupos de problemas específicos, sem resolver outros; e as ferramentas auxiliares resolvem um único problema sem ligação com os demais.

Neste trabalho nós investigamos o problema da criação de uma meta-linguagem para síntese sonora capaz de utilizar diferentes paradigmas tais como unidades geradoras e execução de programas externos. A criação de uma meta-linguagem para síntese sonora permite uma solução elegante para os problemas colocados, sem a necessidade de implementar um novo compilador acústico, e permite uma integração difícil de ser alcançada com o uso dos utilitários atuais.

Abstract

The software sound synthesis is closely related to the Music N programs started with Music I in 1957. Although the Music N has many advantages such as the unit generators and a flexible score language, it presents a few problems like limitations on instrument reuse, inflexibility of parameters, lack of a built-in graphical interface, and usually only one paradigm for scores.

Some solutions concentrate in new from-scratch Music N implementations, while other focus in building user tools like pre-processors and graphical utilities. Nevertheless, the new implementations in general focus in specific groups of problems leaving other unsolved. The user tools only solve one unique problem without connection with others.

In this work we investigate the problem of creating a meta-language for sound synthesis capable of using different paradigms like unit generators and running external programs. The creation of a meta-language for sound synthesis constitutes an elegant solution for the above cited problems, without the need of a yet new acoustic compiler implementation, and allows a tight integration which is difficult to obtain to have with the present user tools.

CAPÍTULO 1

Introdução

Desde quando os computadores começaram a ser usados para tarefas musicais como síntese sonora, tem-se procurado maneiras de melhorar a interação entre o compositor/músico e o computador mantendo-se flexibilidade e potencialidade. Contudo, não é raro o caso onde programas de síntese requeiram que o compositor aprenda uma linguagem de programação completa, tarefa pouco razoável considerando a formação acadêmica do compositor. Esta tese identifica e discute alguns problemas relacionados com programas para síntese e apresenta algumas possíveis soluções.

1.1 Problemas

A história da síntese sonora por *software* (SWSS) está intimamente ligada à série de programas escritos por Max Mathews nos anos 50 e 60. A série, cujo primeiro programa (Music I) foi escrito em 1957 culmina com o Music V, desenvolvido em 1969. Segundo Roads, "for many musicians, including the author of this book, it [Music V] served as an

introduction to the art of digital sound synthesis"[1] (Roads 1996, p. 90).

Outros programas para SWSS, como o Music 4BF, Music 360, Music 11, Csound, Cmusic, Common Lisp Music, dentre outros, foram desenvolvidos tendo o Music V como modelo. Em geral esses programas são referidos como programas do tipo ou família Music N.

Apesar de seus méritos, como as unidades geradoras, o uso de uma linguagem de partitura flexível, poder e velocidade de processamento, a Music N apresenta alguns problemas que podem ser divididos em: criação de instrumentos, criação de partituras, e interação entre ambos.

Com relação à criação de instrumentos, o primeiro problema é a dificuldade de um mesmo instrumento ser usado em diversas composições com alterações mínimas. Isso se deve ao fato de que em algumas implementações do Music N os instrumentos e funções são numerados ao invés de nomeados. Poucas implementações permitem grande flexibilidade de comunicação e troca de dados entre instrumentos, e nenhuma permite a definição de saídas sonoras dependente de contexto. O segundo problema é que originalmente os parâmetros de uma unidade geradora são fornecidos como uma lista ordenada. Isso dificulta não apenas a utilização pelo usuário (é difícil lembrar a ordem e função de todos os parâmetros, principalmente quando uma unidade geradora usa dezenas deles) quanto por programas que necessitem extrair dados do instrumento. O terceiro problema é a falta de escalabilidade dos utilitários criados para descrever instrumentos graficamente. Esses utilitários têm que conhecer a fundo a sintaxe da linguagem, às vezes implementando um *parser* completo. Alguns programas como Supercollider e Csound implementam opcodes para *widgets* gráficos, mas essa solução implica em ter os elementos gráficos codificados no mesmo nível dos opcodes de síntese sonora. Essa solução não é escalável porque os dados de síntese estão misturados com dados gráficos; se a maneira que o instrumento é representado tem que ser mudada, o instrumento tem que ser modificado.

A criação de partituras representa um problema completamente diferente porque é ne-

[1] "para muitos musicistas, incluindo o autor desse livro, ele [Music V] serviu como uma introdução à arte da síntese sonora digital".

las que a música é descrita. E compositores diferentes compõem de maneiras diferentes e necessitam de ferramentas diferentes. Algumas soluções como pré-processadores e uso de linguagens genéricas de programação são limitadas. Os pré-processadores são limitados a uma única sintaxe enquanto que no uso de linguagens de programação exige-se que o compositor aprenda uma linguagem completa antes de começar a compor, o que não é razoável. Outro problema é que o uso de linguagens de programação tende a criar "nichos" e duplicação de trabalho.

O último problema é a falta de integração entre a orquestra e a partitura, e principalmente, a falta de integração entre soluções para a partitura (como pré-processadores) e a orquestra. Em geral as ferramentas para partitura transferem a representação em um nível mais alto que o da lista de notas. Porém, isso faz com que não haja comunicação entre a pré-partitura—o arquivo que será processado e convertido na partitura—e a orquestra (fig. 1.1). Uma comunicação entre o arquivo do pré-processador e a orquestra, ou melhor ainda, entre a pré-partitura e uma pré-orquestra seria altamente desejável (fig. 1.2).

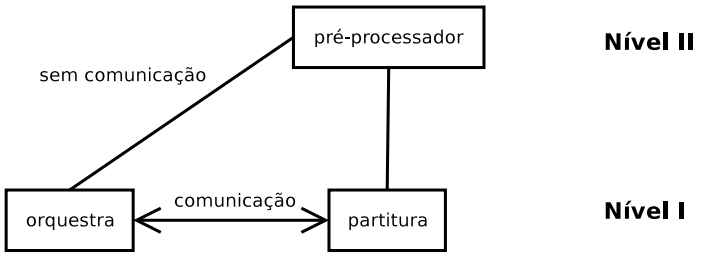

Figura 1.1: Relação entre partitura, orquestra, e pré-processador

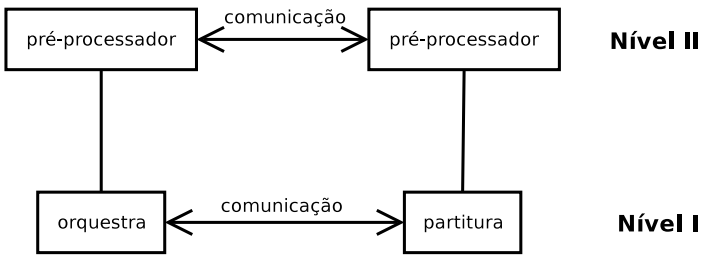

Figura 1.2: Relação entre partitura, orquestra, e pré-processador

1.2 Objetivos

1.2.1 Objetivos Gerais

Investigar a criação de uma meta-linguagem para síntese sonora capaz de utilizar diferentes paradigmas tais como unidades geradoras e execução de programas externos. A criação de meta-linguagens para síntese sonora constitui um problema interessante de pesquisa, porque permite soluções sem a necessidade de implementar um novo compilador acústico. A meta-linguagem funciona como um estágio intermediário entre a descrição de instrumentos e a síntese *per se*.

1.2.2 Objetivos específicos

A criação de uma biblioteca de interface entre a meta-linguagem e uma linguagem real de síntese (e.g. Csound).

A criação de mega-instrumentos, uma maneira de descrever instrumentos maiores compostos de instrumentos definidos na meta-linguagem.

A criação de uma descrição hierárquica e modular de eventos.

1.3 Organização da tese

Esta tese está organizada da seguinte maneira:

- o capítulo 1 identifica os problemas que serão abordados e os objetivos deste trabalho
- o capítulo 2 examina trabalhos relacionados ao tópico desta tese
- o capítulo 3 descreve a csoundXML, uma meta-linguagem para síntese sonora proposta para ajudar a resolver os problemas aqui apresentados
- o capítulo 4 descreve a CXL, uma biblioteca em XML para Csound cujo intento é ligar o csoundXML ao Csound

- o capítulo 5 introduz o conceito de mega-instrumentos, que permitem a descrição em alto nível dos componentes de síntese sonora. Com os mega-instrumentos é possível criar instrumentos do csoundXML que contenham outros instrumentos, permitindo grande modularidade

- o capítulo 6 descreve as soluções para uma descrição hierárquica de eventos e "renderização" distribuída

- o capítulo 7 demonstra alguns exemplos de aplicação das tecnologias apresentadas, como o programa Monochordum, cujo código fonte se encontra no apêndice B.

CAPÍTULO 2

Revisão de Literatura

Neste trabalho algumas considerações sobre sistema para síntese são postas em relevo. Em relação aos programas da família Music N, procurou-se rever diferentes implementações, mas com ênfase no Csound (Vercoe 2001; Boulanger 2000). Alguns problemas específicos do Csound podem ser vistos em (Dahan 2001; Gogins 2001; Kröger 2000; Pope 1993).

Uma visão geral do processo de compor com computadores e alguns aspectos históricos podem ser vistos em (Loy 1989; Pennycook 1985; Lyon 2002; Pope 1995; Smith 1991).

2.1 Linguagens para síntese sonora

2.1.1 Introdução

Em geral um sistema para síntese pode ser comparado com uma orquestra no mundo real (Deyer 1984), como pode ser visto nas figuras 2.1 e 2.2. No sistema de música por computador (fig. 2.2) a partitura contém os dados a serem processados, o regente e os musicistas

são o processo, e o instrumento é o meio de síntese (Deyer 1984).

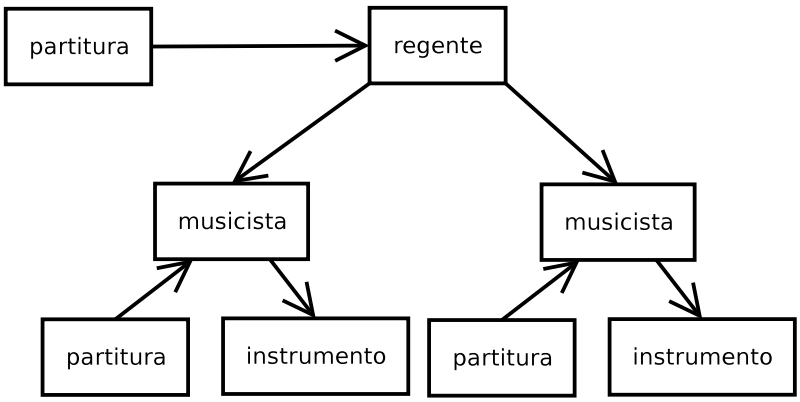

Figura 2.1: Uma orquestra (baseado em (Deyer 1984, p. 252))

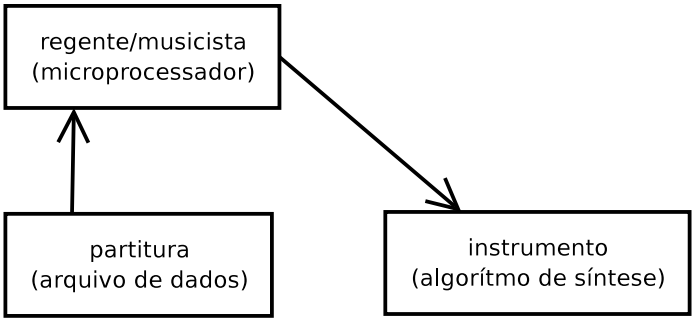

Figura 2.2: Um sistema de música por computador (baseado em (Deyer 1984, p. 252))

Na Music N a partitura é colocada em um arquivo separado e tem geralmente uma lista de notas e eventos. Os instrumentos são reunidos em um arquivo chamado de "orquestra". O programa de síntese, ou *compilador acústico* lê ambos arquivos e gera som como resultado.

2.1.2 Design de instrumentos

A Music N apresenta uma solução bastante boa como linguagem de síntese sonora. As unidades geradoras permitem bastante flexibilidade já que elas podem ser conectadas a praticamente qualquer outra unidade geradora ou modificadores de sinais (Roads 1996, p. 787) permitindo a construção de instrumentos complexos de uma maneira relativamente fácil. Além da orquestra e da partitura, muitas vezes se usa um pré-processador (ver seção 2.1.3.3) para lidar com representações composicionais em um nível mais alto e musical que a lista de notas. A divisão em três linguagens, uma para composição, outra para a parti-

tura, e outra para o instrumento "turns out to be a convenient one. It is often used implicitly. ... These languages can vary between a general programming language and a simple data representation"[1] (Desain e Honing 1988, p. 31).

Por outro lado,

> it is a great advantage to have a close link between the compositional language and the synthesis language. With Nyquist[2], one language serves both composition and signal processing, and blurs the distinction between these tasks[3] (Dannenberg, Desain, e Honing 1997, p. 291).

A vantagem de se ter uma única linguagem para composição e síntese é a possibilidade de tratar problemas composicionais e de síntese de forma unificada. Outra vantagem é que em geral esses sistemas incluem a possibilidade de funções criadas pelo usuário, resultando na criação de um sistema pessoal. A vantagem de ter diferentes linguagens é que se pode escolher sintaxes diferentes para problemas diferentes. Pode-se até mesmo usar linguagens de programação diferentes para cada uma delas aproveitando melhor o potencial de cada uma. Por exemplo, *C++* para criar os instrumentos de síntese, devido seu poder e velocidade, e alguma linguagem de *scripting* (como *TCL* ou Python) ou simbólica (como Lisp) para a linguagem composicional. Outra vantagem é que é mais fácil reunir coleções de instrumentos já que eles ficam necessariamente em arquivos separados.

Apesar das características básicas do Music N não terem mudado significativamente (Roads 1996, p. 788–789), diferentes implementações do Music N usam diferente paradigmas. Por exemplo, Cmix e CLM são ambas implementações do Music N, mas enquanto o CLM usa o paradigma de listas (*Lisp*) o Cmix usa o paradigma de programação estruturada (*C*).

O Csound "uses a mix of assembly language (e.g., `goto` as only control structure) and Fortran (e.g., variable type being determined by the first character of a name) as its programming language model"[4] (Pope 1993, p. 36). Pope acredita que a linguagem de

[1] "acaba sendo algo conveniente. ... Essas linguagens podem variar entre uma linguagem geral de programação e uma simples representação de dados".

[2] Linguagem de síntese sonora, ver (Dannenberg 1997; Dannenberg 1993a)

[3] "é uma grande vantagem ter uma ligação próxima entre a linguagem composicional e a linguagem de síntese. Com Nyquist uma linguagem serve tanto para o processamento composicional quanto para o processamento de sinal e atenua a distinção entre essas tarefas".

[4] "usa uma mistura da linguagem assembly (e.g., `goto` como o único controle de estrutura) e Fortran (e.g.,

orquestra do Csound é boa para não-programadores e iniciantes, mas "feia" e menos flexível em comparação com linguagens como Cmix e Cmusic (Pope 1993, p. 50).

2.1.2.1 Reutilização de instrumentos

Um dos principais problemas na utilização de instrumentos em programas de síntese é a dificuldade de reutilização, ou seja, a possibilidade de um mesmo instrumento ser usado em diversas composições com alterações mínimas. O principal fator é o uso de números para instrumentos e funções, ao invés de nomes.

Por exemplo, caso se esteja trabalhando em uma composição que contenha dezenas de instrumentos e deseje-se usar um instrumento que tenha sido utilizado em uma outra composição, provavelmente ter-se-à que mudar o número do instrumento (e reler todo o arquivo de `orquestra` da composição atual para ver qual número é possível de ser usado), e pior, mudar o número atribuído às funções no instrumento (e reler todas as funções definidas na `partitura` para ver que números estão disponíveis), e provavelmente mudar novamente o instrumento para refletir as recentes mudanças com as funções. E se esse instrumento usar sons externos não será surpresa se eles tiverem que ser renomeados—de `soundin.1` para `soundin.111`, por exemplo—caso essa nova composição já esteja usando um arquivo com aquele nome. E novamente modificar a orquestra e partitura para que reflitam essa mudança.

Instrumentos nomeados. Uma solução comum é definir instrumentos usando nomes (variáveis) ao invés de números. Dentre as linguagens que implementam esse recurso estão Fugue (Dannenberg, Fraley, e Velikonja 1991), Nyquist (Dannenberg 1997), CLM (Schottstaedt 1994), e M (Puckette 1984)[5]. A vantagem dessa abordagem é que instrumentos podem ter nomes mais significativos como `reverb`. Dessa maneira é muito mais fácil manter uma biblioteca (ou banco de dados) de instrumentos para poder reutilizá-los. "The 'M' code is much more symbolic: the table is named 't-sine' instead of '1'; the parameters

tipo de variável sendo determinado pelo primeiro caractere de um nome) como modelo de linguagem de programação".

[5] A partir da versão 4.23 esse recurso foi implementado em algumas versões do Csound mais ainda não está completamente estável e testado.

are named ‘cps’ and ‘amp’ intead of ‘p4’ and the instrument itself is named ‘bang’ instead of ‘1’”[6] (Puckette 1984, p. 18). A solução definitiva seria eliminar o arquivo de partitura (que pode conter vários instrumentos) e usar arquivos separados para cada instrumento. Quando o sistema ler a partitura verá quais instrumentos precisam ser carregados e irá procurá-los no caminho (*path*) designado. Até onde sabemos, essa solução não existe na literatura para programas de síntese sonora, mas é usada para carregar pacotes para o sistema de preparação de documentos LaTeX (Lamport 1994) e plugins do sistema de plugins para Linux, LADSPA (Phillips 2001).

Entrada/saída flexíveis. Na Music N o conceito das unidades geradoras (UG) é bastante poderoso e flexível porque qualquer UG pode servir como entrada ou saída para outra UG. Da mesma maneira é interessante que um instrumento possa servir como entrada ou saída para outro instrumento, provendo um nível a mais de hierarquia e modularidade. No Csound isso só é possível de maneira inadequada com variáveis globais ou com o sistema `zak`. Whittle escreveu

> the Zak system some years ago as a quick hack to get me by until the hoped-for arrays were introduced to Csound. While Csound has many strengths, I think it sucks as a language because there are no arrays, no user-definable data types and no functions. Perhaps some of this has changed in recent years, but I can’t imagine the changes are elegant[7] (Whittle 2003).

E de fato as soluções nesse sentido para Csound tem sido mais “hacks” rápidos que soluções definitivas (Maldonado 2003).

Múltiplas saídas. Um outro recurso que praticamente não existe em nenhum programa de síntese é o de se ter múltiplas definições de saída. Por exemplo, no Csound existem diferentes comandos para diferentes tipos de saída, como mono ou estéreo (`out` e `outs`, respectivamente). Como o tipo de saída é definido em cada instrumento, para modificar o

[6] “O código ‘M’ é muito mais simbólico: a tabela se chama ‘t-sine’ ao invés de ‘1’; os parâmetros são chamados ‘cps’ e ‘amp’ ao invés de ‘p4’ e ‘p5’ e o instrumento é chamado ‘bang’ ao invés de ‘1’”.

[7] “o sistema Zak alguns anos atrás como um hack rápido até que os tão esperados arrays fossem introduzidos no Csound. Ainda que o Csound tenha muitos pontos fortes, eu acho que ele é ruim como linguagem porque não tem arrays, tipos de dados definidos pelo usuário, e funções. Talvez algo disso tenha mudado nos últimos anos, mas eu não posso imaginar que as mudanças sejam elegantes”.

tipo de saída global tem que se modificar o código de cada instrumento. Naturalmente isso é pouco prático, especialmente quando se está experimentando diferentes tipos de configuração. Uma solução é ter uma lista com diferentes possibilidades de saídas e o programa escolherá qual a saída mais apropriada.

2.1.2.2 Parâmetros auto-explicativos

Um paradigma comum, principalmente entre programas baseados em *Lisp*, é o uso de parâmetros auto-explicativos (Schottstaedt 2002; Schottstaedt 1994; Dannenberg, Fraley, e Velikonja 1992). O Music N, sendo primariamente baseado em listas simples (lista de notas, listas de parâmetros), não utiliza esse recurso, bem como implementações modernas como Csound, Cmusic (Loy 2002; Moore 1998), e Cmix (Pope 1993). Contudo esse recurso traria diversas vantagens para as linguagens baseadas em Music N.

A primeira vantagem de se ter parâmetros auto-explicativos é poder escrever os parâmetros em qualquer ordem. Por exemplo, para o opcode `oscil` o Csound define 3 parâmetros:

```
oscil amplitude,freqüência,função
```

que devem ser substituídos pelos equivalentes numéricos:

```
oscil 10000,440,1.
```

A ordem dos parâmetros deve ser estritamente seguida. O código

```
oscil 440,10000,1
```

traria um resultado diferente que o exemplo anterior. Por outro lado, tendo parâmetros auto-explicativos como em

```
oscil amp:  10000, freq:  440, func:  1,
```

a ordem pode ser mudada sem prejuízo para o compilador, como em:

```
oscil freq:  440, func:  1, amp:  10000.
```

Outra vantagem é a possibilidade de saber de imediato quais parâmetros foram usados e quantas vezes. Um uso óbvio disso é para utilitários gráficos, como geradores de funções. O programa pode ler o arquivo de entrada e procurar pela palavra-chave `func`. Dessa

maneira é possível saber quais são as funções e como editá-las.

Porém, o uso de parâmetros auto-explicativos é mais comum em partituras do que em instrumentos, mesmo em linguagens que trabalham com esse paradigma. Uma das prováveis razões é porque o uso de parâmetros auto-explicativos torna a definição do instrumento muito mais prolixa. Schottstaedt deixa isso claro no manual do CLM:

> when make-oscil is called, it scans its arguments; if a keyword is seen, that argument and all following arguments are passed unchanged, but if a value is seen, the corresponding keyword is prepended in the argument list. So, for example,
> ```
> (make-oscil :frequency 440.0)
> (make-oscil :frequency 440.0 :initial-phase 0.0)
> (make-oscil 440.0)
> (make-oscil)
> (make-oscil 440.0 :initial-phase 0.0)
> (make-oscil 440.0 0.0)
> ```
> are all equivalent, but
> ```
> (make-oscil :frequency 440.0 0.0)
> (make-oscil :initial-phase 0.0 440.0)
> ```
> are in error, because once we see any keyword, all the rest of the arguments have to use keywords too[8] (Schottstaedt 2002).

Ele implementou essa

> unusual argument interpretation because in many cases it is silly to insist on the keyword; for example, in make-env, the envelope argument is obvious and can't be confused with any other argument, so it's an annoyance to have to say ':envelope' over and over[9] (Schottstaedt 2002).

Algumas linguagens que implementam esse tipo de recurso na partitura, como Fugue, simplesmente ignoram esse recurso no instrumento. O nosso exemplo anterior seria definido

[8] "quando make-oscil é chamado ele procura seus argumentos; se uma palavra chave é vista, esse argumento e todos os argumentos seguintes são passados sem modificação, mas se um valor é visto, a palavra chave correspondente é acrescentada no início da lista de argumentos. Por exemplo,
```
(make-oscil :frequency 440.0)
(make-oscil :frequency 440.0 :initial-phase 0.0)
(make-oscil 440.0)
(make-oscil)
(make-oscil 440.0 :initial-phase 0.0)
(make-oscil 440.0 0.0)
```
são todos equivalentes, mas
```
(make-oscil :frequency 440.0 0.0)
(make-oscil :initial-phase 0.0 440.0)
```
retornam um erro, porque quando uma palavra chave é vista, todo o resto do argumento tem que usar palavras chaves também".

[9] "interpretação não usual porque em muitos casos é bobo insistir na palavra chave; por exemplo, no make-env, o argumento do envelope é óbvio e não pode ser confundido com nenhum outro argumento, então é irritante ter que dizer ':envelope' o tempo inteiro".

em Fugue algo como: `(oscil 10000 440 1)` (Dannenberg, Fraley, e Velikonja 1992), ou seja, com uma lista de parâmetros como no Music N. Apesar de possuir grande flexibilidade, o CLM não suporta a troca da ordem, uma das principais vantagens desse recurso. O Musickit possui (Jaffe 1989) suporte completo para parâmetros auto-explicativos, contudo utiliza o esquema pouco estruturado de `noteOn` e `noteOff` como o MIDI, além de não possuir estruturação em eventos.

Resumindo, apesar de acrescentar prolixidade à definição da partitura, o uso de parâmetros auto-explicativos pode possibilitar a automação de tarefas ou a criação de ferramentas para lidar com parâmetros. Um bom efeito colateral é que as ferramentas não precisam conhecer a fundo a sintaxe da linguagem.

2.1.2.3 Unidades gráficas

É necessário fazer a distinção entre *editor de parâmetros* (EP) e *editor de instrumentos* (EI).

O editor de parâmetros "or voice editor lets musicians adjust the parameters of a synthesis instrument, preferably while listening to the sound"[10] (Roads 1996, 749). Escolhemos a nomenclatura "editor de parâmetros" por ser mais clara em português, ainda que o termo original seja algo como "editor de conexões" (*patch editor*). "The term 'patch' originates from the modular analog synthesizers of the 1960s and 1970s, where a patch was a configuration of modules interconnected with patch cords"[11] (Roads 1996, 749).

O editor de instrumento permite que o usuário crie instrumentos de síntese sonora conectando módulos de processamento de sinal. A diferença básica entre o editor de instrumentos e o editor de parâmetros é que enquanto esse permite variações de um *preset patch* aquele permite a construção de novos *patches* a partir de uma coleção de módulos (Roads 1996, 749).

Originalmente os EP foram designados para trabalhar com DSPs de arquitetura fixa,

[10] "ou editor de vozes permite que o musicista ajuste os parâmetros de um instrumento de síntese, preferencialmente enquanto escutando o som".

[11] "O termo 'patch' tem origem dos sintetizadores analógicos dos anos 1960 e 1970, quando um patch era a configuração dos módulos interconectados com cabos de conexão".

enquanto os editores de instrumentos para trabalhar com software de síntese (programas de computador). Os EP inicialmente eram encontrados em sintetizadores como o Moog III, Arp 2500, ou Yamaha DX7 e se tornaram comuns em teclados MIDI (Roads 1996, 750-753). Dentre as vantagens do editor de parâmetros estão:

- oculta os detalhes não importantes. Em geral um instrumento de complexidade moderada tem dezenas de parâmetros e comandos. Nem todos os parâmetros são passíveis de modificação ou de interesse imediato. O editor de parâmetros permite que se concentre em um seleto grupo de parâmetros.

- fácil acesso aos parâmetros. Normalmente um EP permite a modificação de dados usando dispositivos gráficos como *slides* ou botões.

- pode permitir E/S flexível. Dada sua condição de "caixa preta" (i.e., alguns dados são escondidos do usuário) o EP pode ser usado para se comunicar com outros EP criando estruturas mais complexas.

Editores de parâmetros não são comuns para Csound, provavelmente devido ao fato dele não ser estritamente um programa para rodar em tempo-real. Apesar de unidades gráficas terem sido implementadas no Csound, e alguns instrumentos que funcionam como EP terem sido criados (Comajuncosas 2002a; Comajuncosas 2002b; Comajuncosas 2002c; Comajuncosas 2002d), nenhum verdadeiro EP foi proposto ou criado até então. O Supercollider (McCartney 2002; McCartney 1996) tem recursos gráficas que facilitam a construção de *slides* para controlar parâmetros.

Os editores de instrumentos gráficos remontam aos anos 70 como o MINTSYN, Oedit, e Reved. Nos anos 90 os editores de instrumentos se tornaram relativamente comuns (Roads 1996, 753). "The starting point for these editors is the modular patching found in the Music N model .... That is, musicians patch together modules to make a synthesis instrument"[12] (Roads 1996, 753). Eles podem ser *front-ends* ou *self-contained*. O *front-end* é uma

[12] "O ponto de partida para esses editores é a conexão modular encontrada no modelo do Music N .... Isso é, musicistas conectam módulos para fazer um instrumento de síntese".

interface gráfica para algum programa tradicional de síntese do tipo Music N onde o usuário pode criar o instrumento visualmente, e o programa converte para o código do Music N. O *self-contained* faz parte do núcleo do programa (e.g. Reaktor[13]). Dentre as vantagens dos editores de instrumento estão, rápida construção de instrumentos, visualização do fluxo do som/dados, e descrição genérica (podem gerar diferentes saídas).

Os editores de instrumentos para Csound são relativamente comuns, como o Csgraph (Bianchini 2002), Visual orchestra (Perry 2002), e Patchwork (Pinkston 1995; Lent, Pinkston, e Silsbee 1989), apenas para citar alguns. Eles servem como auxílio visual à criação de instrumentos mas possuem algumas deficiências que podem ser sumarizadas como:

1. falta de integração com a partitura. Em geral essas ferramentas geram o código relativo ao instrumento *per se* onde a partitura tem que ser adequada ao instrumento.

2. formato binário. O uso de formatos binários dificulta a manipulação de dados por outras ferramentas e a criação de conversores por terceiros. O uso de formato de texto puro seria uma solução mais apropriada.

3. GUI-específico, só funcionam em determinados sistemas. É importante separar o núcleo do programa da unidade gráfica. Dessa maneira é mais fácil poder criar um núcleo portável e adequar a interface gráfica as características de cada sistema e/ou arquitetura.

4. podem ser difíceis de entender e modificar. Instrumentos mais complexos terminam sendo definidos em uma verdadeira "teia" de unidades geradoras interligadas. Uma solução para isso seria ter diferentes níveis de definição.

5. conversão em um único sentido. Os arquivos de partitura são gerados a partir do gráfico, mas não o contrário.

6. não possuem entradas e saídas flexíveis, ao contrário, tornam isso mais difícil pela falta de diferentes níveis de definição.

[13] Programa proprietário para síntese sonora em `http://www.native-instruments.com/`.

O Patchwork foi escrito como um *front-end* básico para programas da família Music N, mas até a presente data ele só gera código para o Csound.

Programas na tradição do Music N são mais fáceis de serem representados graficamente que linguagens declarativas como Nyquist devido ao esquema de conexão. Essa é provavelmente uma das razões porque praticamente não existem editores de instrumentos para programas como Nyquist e CLM.

Algumas linguagens como o Max (Oppenheim 1991b; Puckette 2002) e PD (Puckette e Apel 1998; Puckette 1997; Puckette 1996) usam um paradigma diferente do Music N, o de *patches*. Por isso elas são habitualmente chamadas de linguagem de fluxo de dados, linguagem de conexão, ou linguagem de programação visual (Desain e Honing 1993a). Enquanto no Music N as unidades geradoras podem ser conectadas, nesse tipo de linguagem *tudo* pode ser conectado, inclusive operações numéricas. Contudo a representação gráfica desse tipo de programa tende a ser menos clara com o aumento de complexidade, e

> instead of the neat, old-fashioned block diagrams [e.g., of Music V-style instruments][14] that we used to see in articles ..., now awkward-looking Max patches are often presented—no different symbols for modules, no different line types for different signal types, and a mess of wires[15] (Desain e Honing 1993a, p. 93).

Até agora supôs-se que as unidades gráficas anteriormente descritas gerariam código a partir da disposição gráfica. Infelizmente o contrário não acontece, instrumentos do Csound não podem ser "importados" nesses programas. Esse é um recurso da qual praticamente não se fala nas publicações formais, contudo é um assunto recorrente entre usuários e desenvolvedores do Csound. Um recurso

> potentially useful, and often asked for ... in VisOrc is an Import function for ORC and SCO files so you could graphically edit some of the thousands of CSound instruments available on the web[16] (Perry 2000).

[14] O comentário entre colchetes é do editor do periódico onde o artigo foi publicado.

[15] "ao invés dos diagramas arrumados e antigos [e.g., dos instrumentos no estilo do Music V] que acostumamos ver em artigos ..., agora patches visualmente estranhos do Max são freqüentemente apresentados—sem símbolos diferentes para módulos, sem linhas diferentes para diferentes tipos de sinais, e uma confusão de fios".

[16] "potencialmente útil e freqüentemente pedido no VisOrc é uma função para importar arquivos ORC e SCO de modo que você possa editar graficamente alguns dos milhares de instrumentos do CSound disponíveis na internet".

Nesse email enviado à lista de discussão do Csound, Perry contempla a possibilidade de implementar o recurso de importação no seu VisOrc (Perry 2002), esse recurso, contudo, nunca foi implementado. Ele mesmo sugere a solução, "if visorc saved its projects and instruments in a clearly structured text format it would be possible ... to write some code that did just this"[17] (Perry 2000).

Nós acreditamos que ter o instrumento descrito em um "formato de texto claramente estruturado" possibilita a solução dos problemas relacionados aos instrumentos. Não apenas os problemas de descrição gráfica, mas de reutilização e flexibilidade. No capítulo 3 introduziremos o csoundXML, uma meta-linguagem por nós desenvolvida para síntese em texto estruturado.

2.1.3 *Design* de partituras

A linguagem de partitura serve para especificar a lista de notas que contém o nome dos instrumentos, durações, tempo de início, e parâmetros dos eventos sintetizados pela orquestra. "The score may also include composition procedures ..., but in the simplest case it is simply a list of note events"[18] (Roads 1996, p. 790).

O modelo do Music N sugere que se tenha um arquivo que descreva a orquestra, onde vão as definições de síntese, e outro para a partitura, onde é descrita a composição *per se*. A vantagem desse modelo é que ele permite uma grande flexibilidade tanto para a síntese quanto para a composição. Outros sistemas (derivados ou não do Music N) podem incorporar ambos arquivos ou suprimir um deles, a bem da simplicidade. Por exemplo, alguns sistemas de síntese sonora descartam uma linguagem de partitura em favor de arquivos e comandos MIDI, enquanto sistemas de sampleamento descartam a possibilidade de síntese sonora.

A linguagem de partitura "can be used for expressing intermediate composition re-

[17] "se visorc salvasse seus projetos e instrumentos em uma estrutura de texto claramente estruturada, seria possível ... escrever algum código que fizesse exatamente isso [representar graficamente instrumentos escritos em texto puro]".

[18] "A partitura pode também incluir procedimentos composicionais ..., mas no caso mais simples ela é simplesmente uma lista de notas".

sults, and can function as the main data representation in the compositional part of the system. It should indeed be a general representation language for musical objects"[19] (Desain e Honing 1988, p. 31).

2.1.3.1 Representação musical

Não incluiremos aqui uma revisão completa dos sistemas de representação musical porque isso fugiria do propósito geral deste texto. Uma introdução pode ser vista em (Dannenberg 1993b; Selfridge-Field 1997), enquanto (Byrd 1994; Cahill e Ó Maidín 2001; Hoos, Renz, e Görg 2001; Balaban 1996; Cahill 1998; Diener 1990; Droettboom ; Haken e Blostein 1993; Pope 1989; Hoos, Hamel, e K. Renz 1998; Huron 2002; D. e Fujinaga 2001; Droettboom et al. 2001; Renz e Hoos 1998; Brinkman 1984; Pope 1992) provê mais detalhes em profundidade. Uma extensa lista de códigos musicais é provida em (Castan 2002; Mounce 2002).

Muitas representações são impróprias para intercâmbio porque se concentram em um único paradigma. Elas podem ser úteis somente como códigos de entrada. Por outro lado representações como o SMDL (Newcomb 1991; ISO/IEC 1995) tentam representar demais. O SMDL não atraiu usuários e desenvolvedores porque é difícil aplicar uma representação tão genérica para uma ferramenta particular (Castan, Good, e Roland 2001).

Assim como Dannenberg, também achamos que "music invites formal description"[20] (Dannenberg, Desain, e Honing 1997, p. 271). O desenvolvimento de linguagens formais para música é dado pela necessidade de compositores de expressar suas composições no computador em um formato fácil, sucinto, e amigável ao musicista; e pela pesquisa de como sinais musicais e estruturas de evento discretas podem ser formalizadas de uma maneira útil, expressiva, compacta, e manipulativa (Pope 1997). Um formalismo musical

> implemented as a computer program must be completely unambigous, and implementing ideas on a computer often leads to greater understanding and new insights into the

[19] "pode ser usada para expressar resultados composicionais intermediários, e pode funcionar como a principal representação de dados na parte composicional do sistema. Ela deve ser uma linguagem geral de representação para objetos musicais".

[20] "a música encoraja uma descrição formal".

underlying domain[21] (Dannenberg, Desain, e Honing 1997, p. 271).

O problema da representação musical é que a música tem muitos conceitos complexos cujo significado depende do contexto, sendo portanto difícil capturar esse significado com uma linguagem formal e manter a riqueza das construções e interações. Quando as representações não reduzem a música a construções simplistas e rígidas, e quando podem ser definidas claramente e formalmente, seu uso será natural (Dannenberg, Desain, e Honing 1997).

Um dos elementos que Dannenberg coloca como necessários para a obtenção de isomorfismo (correspondência escrita) entre representação e o som é o do "polimorfismo sensível ao contexto" ou "abstração comportamental" (Dannenberg, Desain, e Honing 1997). Um outro conceito que é interessante é o da "adequação representacional" onde conceitos musicais simples devem ser representados de maneira simples e somente notações complexas devem requerer representação complexa (Hoos et al. 2001).

Nos anos 80 Dannenberg iniciou o desenvolvimento de um programa (que nunca foi concluído) para notação que, além de produzir partituras e partes, "can serve as an interface to various forms of *software* and hardware synthesis. It can provide a standardized representation for music so that output from one program can serve as input to another"[22] (Dannenberg 1986, p. 153).

Uma representação padrão daria a possibilidade de comunicação entre diferentes programas e intercâmbio entre sistemas, ou seja, "a standard representation for computers will facilitate the transportability of scores, and allow several different synthesis devices as well as a variety of applications to be driven from one workstation"[23] (Deyer 1984, p. 251).

Infelizmente a representação padronizada não se concretizou. Atualmente o único protocolo largamente usado para intercâmbio de dados entre programas e hardware é o MIDI,

[21] "implementado como um programa de computador deve ser completamente desprovido de ambiguidade, e implementar idéias no computador freqüentemente conduz a um maior entendimento e novos insights no domínio implícito".

[22] "pode servir como uma interface para várias formas de programas e hardware de síntese. Ele pode prover uma representação padrão para música de modo que a saída de um programa pode servir de entrada a outro".

[23] "uma representação padrão para computadores irá facilitar a transportabilidade de partituras, e permitir que tipos diferentes de dispositivos de síntese e uma variedade de aplicativos possam ser comandados de uma estação de trabalho".

cujas limitações são bem conhecidas (Selfridge-Field 1997; Selfridge-Field 1994) como a impossibilidade de notação enarmônica (i.e. bemóis não podem ser distinguidos dos sustenidos e vice-versa); a não existência de durações estritas (o MIDI usa eventos como `noteon` para indicar que uma nota foi iniciada e `noteoff` para indicar seu término), o que dificulta transcrições rítmicas; a falta de articulações; e mudança interna que algumas articulações causam, como *staccato* (muda a duração da nota) e tremolo (muda o instrumento).

O MusicXML (Good 2001) procura ser a nova linguagem de intercâmbio. "MusicXML is intended to support interchange between musical notation, performance, analysis, and retrieval applications"[24] (Good 2001, p. 113). Ele tem sido cada vez mais utilizado em programas comerciais como o Finale[25] e Sibelius[26]. Infelizmente no MusicXML não há nenhum suporte para partituras que precisam de diversos parâmetros adaptáveis a diferentes contextos (em oposição a um conjunto fixo de parâmetros). E, devido ao seu *design*, não é possível aumentar a linguagem com definições para síntese. Além disso ele tem más decisões de *design* como o uso de excesso de informação na codificação. Ele codifica elementos como números de compasso, direção da haste da nota, linha onde fica a clave de sol, quando eles podem ser deduzidos pelo programa. Outro problema é que alguns dados não são descritos de maneira "musical". O ex. 2.1 mostra o código usado para definir a armadura de mi♭ maior. Definir a armadura pelo nome como em `<key type='eb' mode='major'>` seria mais apropriado que codificar diretamente o número de sustenidos ou bemóis na armadura.

Exemplo 2.1 Definindo armadura no MusicXML

```
<key>
  <fifths>-3</fifths>
  <mode>major</mode>
</key>
```

[24] "A intenção do MusicXML é permitir o intercâmbio entre aplicativos de notação musical, performance, analise, e extração".

[25] `http://www.finalemusic.com`.

[26] `http://www.sibelius.com`.

2.1.3.2 Listas de notas

Nas linguagens da família da Music N a partitura é representada como uma lista de notas sem hierarquia codificada por números. A maior vantagem das listas de notas é o controle preciso de cada parâmetro dos sons sintetizados, enquanto os maiores problemas são "the numerical orientation, rigid syntax, and a lack of higher level structures (such as phrases and voices)"[27] (Roads 1996, p. 795).

A falta de estrutura é provavelmente um dos aspectos mais problemáticos com as listas de notas. É impossível representar estruturas como frases e interação entre vozes com esse tipo de sintaxe. A lista de notas é considerada como estando no nível mais elementar da estrutura musical (Roads 1996). Para lidar com diferentes aspectos musicais, alguns instrumentos acabam tendo dezenas de parâmetros:

> we often work with instruments which have as many as 99 parameters. These added parametres will refer to things such as quadriphonic position, envelopes, vibrato, reverberation, glissando limits, various ways of creating different timbres, etc[28] (Smith 1981, p. 226).

Essa grande quantidade de parâmetros é um problema, porque

> in a complicated instrument with many parameters, the note list is unreadable as music. Since the position of each number in the list determines the parameter to which the number applies, all parameters must be supplied for every note, or else a special character must be inserted into that position to indicate a repeating or null value[29] (Roads 1996, 795–796).

Portanto "to creat and edit note lists, we obviously need a general parameter-naming facility"[30] (Schottstaedt 1983, p. 13). O programa básico para se criar e editar listas de notas é um *pré-processador*. "The primary reason for using a note-list preprocessor is

[27] "a orientação numérica, a sintaxe rígida, e a falta de estruturas em um nível mais alto (como frases e vozes)".

[28] "nós trabalhamos freqüentemente com instrumentos que tem até 99 parâmetros. Esses parâmetros acrescentados referem-se a coisas como posição quadrafônica, envelopes, vibrato, reverberação, limites de glissando, diferentes formas de criar diferentes timbres".

[29] "em um instrumento complicado com muitos parâmetros, a lista de notas é ilegível como música. Já que a posição de cada número na lista determina o parâmetro na qual o número se aplica, todos os parâmetros devem ser fornecidos para cada nota, ou um caractere especial deve ser inserido naquela posição para indicar um valor repetido ou vazio".

[30] "para criar e editar listas de notas nós obviamente necessitamos de um recurso geral de parâmetros nomeados".

that it facilitates coding musical ideas for a music-synthesis program"[31] (Brinkman 1981, p. 178).

Algumas vezes programas de síntese oferecem outros tipos de sintaxe no lugar das listas de notas, mas "the limitations of the note-list syntax become much less painful if the note-list generator provides all the control and flexibility the composer needs"[32] (Schottstaedt 1983, p. 13).

2.1.3.3 Pré-processadores

Linguagens como Formes (Rodet e Cointe 1984) e Pla (Schottstaedt 1983) foram criadas para lidar e processar lista de notas, suprimindo algumas de suas limitações como a falta de hierarquia de frases e notas. Esses programas podem ser considerados como *front ends* para um programa Music N já que eles podem gerar listas de notas a partir da sua descrição de alto nível (Roads 1996, p. 796).

O uso de bloco de parâmetros foi o primeiro tipo de pré-processador para lista de notas e foi originalmente implementado no Score (Smith 1981). O Score "was one of the first attempts to develop a text-based musical data representation that derives its metaphores from CPN [Common Practice Notation], as distinct from computational models"[33] (Loy e Abbott 1985, p. 258). Outros pré-processadores como o score 11 (Brinkman 2000; Brinkman 1981), e recentemente o nGen (Kuehn 2001) foram implementados posteriormente. Nesses pré-processadores os habituais campos de parâmetros do Music N (ex. 2.2) são substituídos por *blocos* de parâmetros (ex. 2.3).

Exemplo 2.2 Campos de parâmetros no Music N

```
i p1 p2 p3 p4 p5
i p1 p2 p3 p4 p5
i p1 p2 p3 p4 p5
```

[31] "A razão primária para usar um pré-processador de lista de notas é que ele facilita a codificação de idéias musicais para um programa de síntese musical".

[32] "as limitações da sintaxe de lista de notas se tornam muito menos dolorosas se um gerador de lista de notas provê todo o controle e flexibilidade que o compositor necessita".

[33] "foi uma das primeiras tentativas de desenvolver uma representação de dados baseada em texto que deriva suas metáforas da prática musical comum, diferentemente de modelos computacionais".

Além da óbvia mudança do sentido horizontal para o vertical, essa abordagem traz diversas vantagens, como uso de terminologia musical tradicional, facilidades para obter boa articulação e fraseado, e a possibilidade de gerar automaticamente dados redundantes (Smith 1981).

Exemplo 2.3 Blocos de parâmetros

```
p1 { ... }
p2 { ... }
p3 { ... }
p4 { ... }
p5 { ... }
```

O Pla implementa a idéia de *message passing* (Krasner 1980; Weinreb e Moon 1981). Cada instrumento tem um valor padrão. Se, em uma lista de parâmetros, um parâmetro é modificado, o valor padrão é substituído. Se um nome de uma mensagem é encontrado, o valor padrão é mantido e o código associado com a mensagem é executado (Schottstaedt 1983). Parâmetros descritivos (ou *mensagens,* como Schottstaedt os chama) como `Pizzicato` substituem listas numéricas de parâmetros (em geral, um único parâmetro descritivo substitui mais de um parâmetro numérico). O ex. 2.4 mostra uma lista de notas típica de uma linguagem do tipo Music N. A mesma lista pode ser escrita de maneira muito mais concisa no Pla, como pode visto no ex. 2.5. Essa lista é muito mais legível e flexível que a outra, especialmente porque a ordem exata das mensagens não importa e "if all instruments are able to recognize a core of messages, reorchestration of a score becomes easy, even when the parameters of various instruments do not fall in the same order"[34] (Schottstaedt 1983, p. 13).

Exemplo 2.4 Lista de notas típica (Schottstaedt 1983, p. 13)

```
Violin .000, 1.210, A/2, .005, Amp, .000, .000, .065, Ind, .000, .000,
2.501, F23, .000, .000, 5.000, .015, Amp, 70.135, 1.000, .100;
```

Exemplo 2.5 Lista de notas no Pla (Schottstaedt 1983, p. 13)

```
Violin .000 1.210 A/2 Soft Pizzicato Locate: 69.23 1 Molto Vibrato
```

[34] "se todos os instrumentos são capazes de reconhecer o núcleo das mensagens, a reorquestração de uma partitura se torna fácil, mesmo quando os parâmetros de vários instrumentos não estejam na mesma ordem".

Contudo essa abordagem tem um problema básico, ela mistura o uso de listas de mensagens com listas de propriedades. Ainda no ex. 2.5 a mensagem `Locate` tem dois parâmetros de entrada, `69.23` e `1`, sendo na verdade uma lista de propriedades.

Em geral os parâmetros de uma lista de propriedades são representados em pares como nome: valor, e.g. `Nota: do, Duração: semínima, Dinâmica: forte` (Dannenberg 1993b). A vantagem das listas de propriedades sobre uma estrutura de dados fixas é que elas permitem que novas informações sejam facilmente acrescentadas. Por exemplo, um compositor poderia editar uma música em notação tradicional mas acrescentar dados de timbre a cada nota. Esses dados seriam passados ao programa de síntese e ignorados pelo programa de notação (Dannenberg 1986). Outra vantagem é que apenas os parâmetros diferentes precisam ser indicados, não sendo necessário reescrever *todos* os parâmetros como nas listas de notas comuns.

Os pré-processadores[35] cumprem adequadamente a função para a qual são designados. Todavia, a inexistência de um pré-processador para orquestras faz com que exista um buraco entre a pré-partitura—o arquivo que será processado e convertido na partitura—e a orquestra. Uma comunicação entre o arquivo do pré-processador e a orquestra, ou melhor ainda, entre a pré-partitura e uma pré-orquestra seria altamente desejável, já que variáveis poderiam ser trocadas entre ambos arquivos, dentre outros recursos.

2.1.3.4 Pequenas linguagens

A elaboração de uma linguagem de representação musical em geral é um processo complexo e às vezes mesmo desnecessário. No livro *Beyond Midi* Selfridge-field pergunta:

> do you really need to invent a new code? The answer is: probably not. As the preceding chapters demonstrate, codes which have already been developed answer, collectively, to a wide range of demands, from simple statistical counts to complex cognitive procedures[36] (Selfridge-Field 1997, p. 572).

[35] Alguns dos pré-processadores disponíveis para Csound podem ser vistos em (Puxeddu 2000; Bartetzki 1997a; Bartetzki 1997b; Blasser 1999; Cooke 2001; Hanna 1999; Miranda 1997; Winkler 2000b).

[36] "você realmente precisa inventar uma nova codificação? A resposta é: provavelmente não. Como os capítulos anteriores demonstraram, codificações que já foram desenvolvidas anteriormente respondem, coletivamente, a um vasto âmbito de demandas, da simples contagem estatística a procedimentos cognitivos complexos".

Por outro lado, em geral as linguagens de representação musical são implementadas segundo algum paradigma específico. Por exemplo, linguagens para notação precisam de instruções específicas como posicionar, alinhar, e desenhar as notas. Programas de síntese por sua vez não precisam dessa informação, mas sim de dados sobre parâmetros de síntese, e controle sobre a estrutura e tempo. Contudo, alguma conexão entre diferentes paradigmas é por vezes necessário. Acrescentar a possibilidade de notação a uma linguagem de síntese é particularmente interessante.

Como "no single musical input language or user interface can adequately accommodate a large range of compositional styles and intentions"[37] (Decker e Kendall 1984, p. 243), poder-se-iam criar "pequenas linguagens" como mini-pré-processadores, para atender a um problema específico ou refletir um determinado estilo de sintaxe. Com isso seria possível ter mais de uma metáfora para entrada de dados pelo usuário, deficiência criticada por Oppenheim (1992).

Dessa maneira criar-se-iam ferramentas para construir pequenas linguagens, e cada usuário poderia definir a sintaxe que melhor lhe sirva, afinal

> the attempt to satisfy this wide range of musical demands with a singe general-purpose synthesis language fails. ... The need for new strategies becomes obvious when one considers that our notions about synthesis and compositional interfaces keep changing year by year and that a great deal of software is constantly discarded[38] (Decker e Kendall 1984, p. 243).

Por definição uma "pequena linguagem" deve ser pequena, ter um propósito e seguir um paradigma específico, e ser de fácil utilização (Kaplan 1994, p. 3). No mundo da ciência da computação as pequenas linguagens são muito comuns e alguns exemplos incluem arquivos de configuração, edição de linha (Sed), recompilação incremental (Make), desenho de figuras (Pic), dentre muitos outros.

Isso indica uma mudança fundamental de conceito. Ao invés de tentar prover as funcionalidades em uma única linguagem de partitura, "geradores de linguagem de partitura"

[37] "nenhuma linguagem musical ou interface de usuário única pode acomodar adequadamente um âmbito grande de estilos e intenções composicionais".

[38] "a tentativa de satisfazer esse vasto âmbito de demandas musicais com uma única linguagem de síntese de propósito geral falha. ... A necessidade por novas estratégias se torna óbvia quando se considera que nossas noções sobre interfaces para síntese e composição continua mudando a cada ano e que uma grande quantidade de software é constantemente deixada de lado".

poderiam ser criados para atender as necessidades específicas de cada usuário.

2.1.3.5 Módulos

Algumas linguagens de partitura como Pla (Schottstaedt 1983), Canon (Dannenberg 1989), Nyquist (Dannenberg 1997; Dannenberg 1993a), Formes (Rodet e Cointe 1984), e Supercollider (McCartney 2002; McCartney 1996) encontram-se entre a descrição musical e uma linguagem de programação onde a principal idéia é criar partituras simples que gerarão partiras mais complexas (Dannenberg 1989). Em geral essas linguagens são utilizadas em sistemas específicos e não interagem com outros sistemas.

Recentemente, com a popularização das linguagens de *scripting* como *Python* e *Tcl*, inúmeros módulos para se processar listas de notas (sobretudo do Csound) têm surgido. Sua principal característica é que uma composição é de fato um programa na linguagem original desse módulo. Ou seja, esse módulo é composto, na verdade, de um conjunto de funções e procedimentos pré-definidos, prontos para trabalhar com dados musicais. Geralmente eles funcionam também como pré-processadores, definindo maneiras de lidar com listagens de notas de maneira distinta à do Csound. Porém o compositor pode usar todos os recursos da linguagem original no processo da composição, não ficando limitado àqueles definidos pelo módulo. Essa constitui uma das maiores vantagens desse tipo de abordagem já que se pode usar todo o poder de uma linguagem de programação moderna.

Os módulos mais conhecidos são escritos para as linguagens de *scripting* mais usadas: Pysco (Winkler 2000a) para *Pyton*, JCself (Kay e Heeren 2000) para *Java*, Scheme Score (Ramsdell 2001) para *Scheme*, Perlscore (Shepard 1999) para *Perl*, e Cybil (Burton e Piché 1998b) para *Tcl*.

Os módulos de programação são, de certa maneira, um avanço em relação aos pré-processadores no sentido em que são extensíveis, ou seja, podem facilmente ser ampliados pelo usuário/compositor. Outra característica marcante a seu favor é que todos os recursos de uma linguagem de programação ficam automaticamente ao dispor do compositor. Por outro lado, como essas partituras são de fato programas, o usuário tem que aprender (com um

certo grau de profundidade) a linguagem para a qual o módulo foi escrito, além de dominar a própria linguagem do Csound. Ainda que essa idéia não seja de todo desinteressante, ela parece servir mais ao compositor que já domina determinada linguagem que aquele que está buscando uma ferramenta composicional. E novamente, aqui não temos qualquer comunicação entre a pré-partitura e a orquestra, já que todos os módulos cuidam apenas da geração de partituras.

2.1.3.6 Notação hierárquica

Concordamos com Desain que "even in the most uncoventional music, organizational and structural aspects are essential"[39] (Desain e Honing 1993a, p. 5). E as linguagens de partitura deveriam possuir algum tipo de relacionamento estrutural, afinal "defining structural relationships is perhaps the most important goal, and the most difficult task of music representation"[40] (Roland 2001, p. 132). Quando a linguagem de partitura não contém estrutura nativamente (como no Music N) é comum usar outra linguagem (ver seção 2.1.3.2) para gerar a partitura (Dannenberg, Desain, e Honing 1997). Uma vantagem das

> hierarchically structured descriptions of music is that transformations such as tempo or pitch can be applied to aggregates of musical objects. In general, hierarchy is a way of representing structure, and it should be no surprise that many composition languages support the notion of hierarchy[41] (Dannenberg 1993b, p. 21).

Ao invés de usar uma linguagem adicional (i.e. um pré-processador), a própria partitura pode ter uma estrutura adicional. Tipicamente, funções, macros, ou procedimentos representam comportamentos ou coleções de eventos musicais. Invocando as funções em diferentes lugares, varias instâncias de comportamento podem ser obtidas. Uma estrutura hierárquica pode ser obtida pelo uso de funções aninhadas (Dannenberg, Desain, e Honing 1997).

Castan enumera alguns exemplos "óbvios" de hierarquia (Castan, Good, e Roland 2001):

[39] "mesmo na música mais não-convencional, aspectos organizacionais e estruturais são essenciais".

[40] "definir relações estruturais é talvez o objetivo mais importante, e a tarefa mais difícil da representação musical".

[41] "descrições de música hierarquicamente estruturadas é que transformações como andamento ou nota podem ser aplicadas a agregados de objetos musicais. Geralmente, hierarquia é uma maneira de representar estrutura, e não deveria ser surpresa que muitas linguagens de composição suportam a noção de hierarquia".

1. sistemas dentro da página
2. pautas dentro de um sistema
3. compassos dentro de uma pauta
4. acordes dentro de um compasso
5. notas dentro de um acorde

Contudo ele não fala de vozes e principalmente de frases, períodos, seções, que constituem elementos hierárquicos muito mais úteis para o compositor. Além do que os ítens 1, 2, e 3 são deduzíveis e modificáveis, além de serem de pouco interesse para um sistema de composição já que dizem respeito à posição de elementos na partitura convencional.

Algumas soluções envolvem o uso de "vozes" e "seções" (Smith 1981; Schottstaedt 1983) ou de *chunks*, onde eventos são organizados em árvores (Buxton et al. 1978). De qualquer forma a idéia básica é que estruturas possam ser aninhadas (por exemplo, vozes dentro de seções) e que os grupos maiores possam ser tratados como uma única nota (para transformações e re-agrupamentos). Acreditamos que a solução definitiva seja o uso de eventos (Kröger 2003b). Dessa maneira pode-se obter os objetivos básicos e possuir um controle temporal sobre cada evento, além de poder representar o tempo de cada evento em função de outros eventos.

2.1.3.7 Representação do tempo

Uma linguagem de partitura deveria ter bons recursos para a representação rítmica. A lista de notas do Music N representa segundos (no caso do Music V) ou *beats* (no caso do Csound). Pré-Processadores como o Score (Smith 1981) usam símbolos ou números que representam as figuras rítmicas da notação tradicional.

O Score representa as figuras rítmicas como números correspondentes ao denominador (e.g. 2 para mínima, 4 para semínima, e assim por diante). Grupos rítmicos complexos são obtidos pela formula $R = n * 4/T$ onde n é o número de unidades durante o tempo de

T semínimas. Assim uma colcheia em uma tercina tem valor igual a 12 ($R = 3 * 4/1$) e cada grupo de uma quiáltera de sete notas no lugar de uma semínima pontuada tem o valor de 18.6667 ($R = 7 * 4/1.5$) (Smith 1981). O primeiro problema é que isso não é exatamente uma codificação mas a representação numérica de cada valor, tornando difícil associar o valor final com a representação musical inicial. O segundo problema é que todas as figuras rítmicas são representadas em um único nível, mesmo em quiálteras aninhadas. O terceiro problema é que, como essa representação usa valores decimais, é fácil prever problemas com dízimas, onde a soma das partes é menor que o todo[42]. Finalmente, ela não facilita operações com o objeto representado[43].

O Score11 (Brinkman 2000; Brinkman 1981) resolve parcialmente os três primeiros problemas anteriormente citados. O ritmo visto na fig. 2.3 pode ser codificado no Score11 como `(4.=(4.=16/16/16/16/16) 4.)`. As quiálteras são agrupadas por parênteses e o sinal de igual indica a figura sendo substituída (e.g. `2=4*3` significa 3 semínimas (`4*3`) no lugar de uma mínima (`2=`)). Essa é de fato uma codificação, e as figuras rítmicas são representadas separadamente da indicação de quiáltera. A hierarquia das quiálteras aninhadas é mantida e o valor real de cada figura só é computado quando necessário, internamente. Contudo essa codificação não facilita operações com o objeto representado.

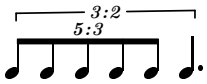

Figura 2.3: Ritmo complexo

O MusicXML e Musedata codificam o denominador da fração em um elemento separado do numerador (Good 2001, p. 121). Para se obter o ritmo real o programa tem que procurar pelos elementos[44] `<note>` e `<divisions>`, ou seja, um trabalho duplo.

Oliveira (Oliveira 1994a; Oliveira 1994b) propõe uma solução muito mais simples e elegante para a representação de grupos rítmicos complexos através de frações. O grupo

[42] Se uma semínima tem a duração de 1000 milisegundos, uma colcheia em uma quiáltera terá a duração de 333 milisegundos (arredondando). Uma série de 30 quiálteras contra 10 semínimas serão distorcidas em 10 milisegundos. (Dannenberg 1993b, p. 22).

[43] Da mesma maneira que operações como transposição, retrogradação, e multiplicação podem ser efetuadas com notas codificadas no sistemas de módulo 12.

[44] Para uma introdução a terminologia do XML ver o capítulo 3.1 na página 38.

rítmico da fig. 2.3 seria representado como $\frac{2}{3}(\frac{3}{5}(\frac{1}{8},\frac{1}{8},\frac{1}{8},\frac{1}{8},\frac{1}{8}),\frac{3}{8})$. Assim como o Score11, os grupos são separados por parênteses[45]; a diferença é que os números fora dos parênteses representam a modificação da quiáltera, enquanto os números dentro dos parênteses representam o valor das figuras. A grande vantagem dessa codificação é que diversas operações podem ser efetuadas, inclusive para verificar a validade da notação proposta. Para obter o valor total de cada figura multiplica-se o valor da figura pelo valor das quiálteras que ela faz parte. Por exemplo, o valor da primeira nota é igual a $\frac{1}{20}$. O valor total de cada figura individual é $\frac{1}{20},\frac{1}{20},\frac{1}{20},\frac{1}{20},\frac{1}{20},\frac{1}{4}$. Se agruparmos as frações com o mesmo denominador e reescrevemos as frações cujo denominador não representam figuras básicas (semínimas, colcheias, etc.) teremos $\frac{4}{5}(\frac{1}{16},\frac{1}{16},\frac{1}{16},\frac{1}{16},\frac{1}{16}),\frac{1}{4}$, cuja representação musical pode ser vista na fig. 2.4. Ou seja, partindo-se da codificação pode-se representar o ritmo original de diferentes maneiras, inclusive corrigindo erros de lógica.

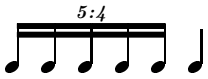

Figura 2.4: Ritmo complexo simplificado

Uma outra possibilidade é o uso de um denominador comum em toda uma seção e representar as figuras como os numeradores; dessa maneira evita-se a conversão para decimais, fonte de imprecisão. Uma aplicação disso pode ser vista nas composições “Mutação I” e “Mutação II” (Oliveira 2001b; Oliveira 2001c).

Um outro tipo de representação consiste no uso de “curvas de andamento”, que é

> a function from beat number to tempo, where tempo is the instantaneous ratio of beats per second, or equivalently, the first derivative of the function from time to beats. . . . The tempo curve is a nice abstraction for mathematical specification. For example, logarithmic tempo curves have been found to produce smooth tempo changes that are musically plausible (at least better than linear)[46] (Dannenberg 1993b, p. 22).

Nas funções de tempo generalizadas—onde cada função de controle é uma função de mais parâmetros, refletindo diferentes aspectos de tempo, como “tempo inicial”, “duração

[45] Colchetes também são usados no lugar de parênteses.

[46] “uma função de número de batidas para andamento, onde o andamento é a razão instantânea de batidas por segundos, ou de maneira equivalente, a primeira derivada da função de tempo para batidas. . . . A curva de andamento é uma abstração interessante para especificação matemática. Por exemplo, descobriu-se que curvas de andamento logarítmicas produzem mudanças de andamento uniformes que são musicalmente plausíveis”.

absoluta", e "progresso relativo"—os valores são passados automaticamente pelo sistema. O usuário pode usar alguns parâmetros e ignorar outros para fazer funções de tempo adequadas a diferentes objetos musicais (Desain e Honing 1992).

Problemas do aspecto rítmico e sua representação podem ser vistos com maior profundidade em (Anderson e Kuivila 1986; Bilmes 1992; Brandt 2002; Brandt 2001; Brandt 2000; Cemgil et al. 2001; Cemgil, Desain, e Kappen 2000; Dannenberg 1994; Dannenberg 1991; Desain e Honing 1993b; Desain, Jansen, e Honing 2000; Desain et al. 2000; Shmulevich e Povel 2000a; Windsor et al. 2000; H. 1993; Honing 2001; Rogers, Rockstroh, e Batstone 1980; Windsor et al. 2001; Honing 1995; Shmulevich e Povel 2000b; Timmers e Desain 2000).

2.2 Linguagens de programação

Um outro fator a ser considerado é a ligação da linguagem de síntese com linguagens de programação. Existem três tipos básicos: as linguagens "independentes" como o Csound, as criadas no topo de linguagens gerais de programação como CLM, e as linguagens para música que tem uma poderosa linguagem de programação embutida como o Supercollider.

Em síntese sonora muitos parâmetros são necessários, sendo muito difícil antecipar as necessidades do compositor. Alguns acham que

> a programming language is ideal for describing customized instruments that have only the parameters of interest and that behave according to the composer's requeriments. In other words, the composer can create his or her own language that is specific to the compositional task at hand[47] (Dannenberg, Desain, e Honing 1997, p. 291).

Por outro lado não é razoável querer que compositores se tornem programadores (Desain e Honing 1988) e "many of the criticisms leveled against general-purpose language are not so valid when data-preparation systems like Leland Smith's SCORE program

[47] "uma linguagem de programação é ideal para descrever instrumentos customizados que tem somente os parâmetros de interesse e que se comporta de acordo com os requerimentos do compositor. Em outras palavras, o compositor pode criar sua própria linguagem que é específica à tarefa composicional em mãos".

are available"[48] (Haynes 1980, p. 23). Mas como não existem programas para preparar partituras para todas as linguagens "it is therefore often necessary for the composer to write his or her own score-preparation routines in a high-level programming language, and this is beyond the capability of many musicians"[49] (Haynes 1980, p. 23).

Schottstaedt acha que "our experience demonstrates that composers find a programming language far more congenial than a data-entry system when trying to write computer music"[50] (Schottstaedt 1983, p. 20).

Contudo, construir linguagens para síntese no topo de linguagens genéricas não tem se mostrado eficaz historicamente. O número de linguagens desse tipo que foram criadas e desapareceram é grande (Loy e Abbott 1985), talvez porque o compositor tem que aprender a linguagem "base" antes de poder fazer qualquer coisa. Isso também ajuda na criação de "grupos" especializados (e.g. aqueles que trabalham com *Lisp*, ou os que trabalham com *Java*) ao invés de uma linguagem "neutra" como o Music N. Outro problema é a tendência a duplicação de trabalho. Um programa como o Supercollider usa uma linguagem assemelhada ao *Smalltalk*. Programadores de outras linguagens podem-se sentir compelidos a escrever um programa semelhante mas com uma sintaxe baseada na sintaxe da linguagem na qual o programa foi escolhido.

2.3 Sistemas para composição e síntese sonora

A criação de instrumentos e partituras geralmente é feita como parte de algum sistema maior de composição e síntese sonora. E "a fundamental consideration in evaluating any music synthesis system is the flexibility offered to the composer in controlling the evolution of a musical structure"[51] (Haynes 1980, p. 23). Deyer propôs um sistema integrado de

[48] "muitas das críticas contra as linguagens de propósito geral não são tão válidas quando sistemas de preparação de dados como o SCORE de Leland Smith estão disponíveis".

[49] "é muitas vezes necessário que o compositor escreva suas próprias rotinas de preparação de partitura em uma linguagem de programação de alto nível, e isso está além da capacidade de muitos musicistas".

[50] "nossa experiência demonstra que compositores acham uma linguagem de programação muito mais amigável que um sistema de entrada de dados quando tentam compor música computacional".

[51] "uma consideração fundamental ao avaliar-se qualquer sistema para síntese musical é a flexibilidade oferecida ao compositor para o controle da evolução de uma estrutura musical".

composição e síntese centrado no arquivo de partitura, como pode ser visto na fig. 2.5 (Deyer 1984).

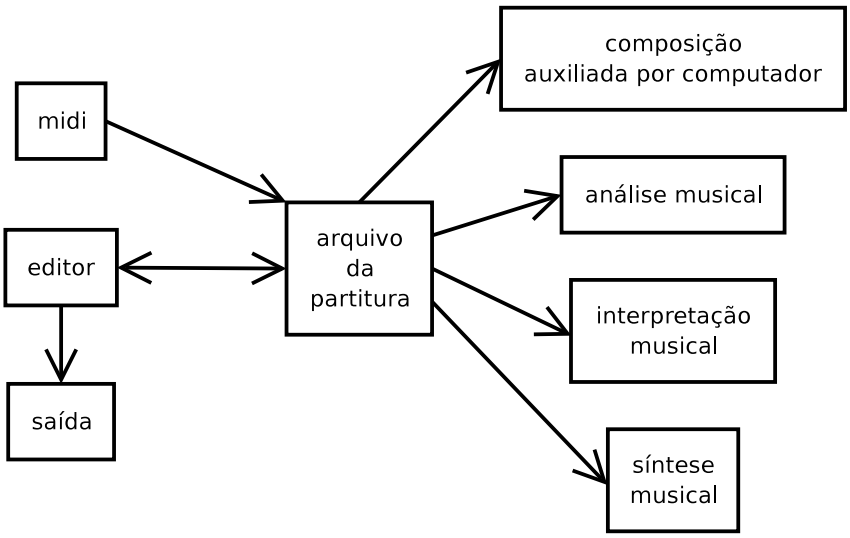

Figura 2.5: Sistema integrado (baseado em (Deyer 1984, p. 251))

Rodet pergunta

> why is MCS [Music Composition and Synthesis] one of the most interesting fields for testing advanced technology for computing? In MCS, perhaps even more than in artificial intelligence, programs have to be tested, modified, and rewritten very often[52] (Rodet e Cointe 1984, p. 32).

Isso talvez mostre porque, assim como em inteligência artificial, o *Lisp* é tão usado (Desain e Honing 1996a; Desain e Honing 1996b). Além de ser uma linguagem de nível muito alto, e boa para processamento simbólico, ele é um ambiente incremental e interpretado de programação, ideal para programação experimental e exploratória (Dannenberg, Desain, e Honing 1997).

Oppenheim tem algumas idéias mais gerais de como deveria ser um ambiente de composição e síntese (Oppenheim 1986):

1. facilidade para o compositor pensar em termos musicais

2. ambiente de tempo real que estimule a experimentação de idéias musicais

[52] "porque CSM [Composição e Síntese Musical] é um dos campos mais interessantes para o teste de tecnologia de computação avançada? No CSM, talvez ainda mais que em inteligência artificial, programas têm que ser testados, modificados, e reescritos muito freqüentemente".

3. boa correlação entre as ações do compositor e a música resultante
4. liberdade para o compositor determinar seus próprios procedimentos composicionais
5. possibilidade de trabalhar dentro de um contexto musical completo
6. facilidade para retrabalhar facilmente qualquer elemento de uma composição
7. a possibilidade de controlar diversos parâmetros simultaneamente

Rodet também enumera algumas características desejáveis de sistemas para composição e síntese (Rodet e Cointe 1984):

1. generalidade. A representação de um processo o mais geral possível (e.g. o modelo de um crescendo deve ser aplicado a diferentes sons)
2. universalidade. Um modelo deve tentar ser independente de técnicas de síntese
3. compatibilidade. Modelos devem ser aplicáveis em qualquer contexto
4. simplicidade. Modelos devem seguir convenções de comunicação e pressuposição (e.g. valores padrões)
5. facilidade de uso.
6. modularidade e construção hierárquica.

O objetivo deveria ser um sistema de composição e não um sistema de produção musical. O sistema para composição deve ser extensível pelo usuário; modular—funções e objetos são isolados e protegidos entre si e podem ser usados e estudados separadamente; e ortogonal—funções e objetos devem ser representados de uma maneira que qualquer extensão irá se encaixar no sistema e usar todos os recursos. Além disso ele pode ter bom mecanismo para nomes—codificação de parâmetros o mais próximo possível do uso normal (e.g. mf ao invés de 67); bom mecanismo para abstração—permitir que famílias de objetos semelhantes sejam definidos (Desain e Honing 1988). Para Desain esses objetivos lembram

o de uma linguagem de programação geral, contudo querer que compositores se tornem programadores não é razoável. Ele sugere que a idéia é ter ferramentas no topo de uma linguagem de programação na forma de "geradores de programas" (Desain e Honing 1988, p. 30).

Outro fator que deve ser levado em consideração é a modularidade do sistema. Dentre as vantagens de um sistema modular estão (Deyer 1984):

1. ferramentas para síntese e composição são programas independentes, de modo que possam ser mais eficientes e apropriados

2. devido ao item 1 o número de programas pode crescer sem problemas, provendo mais funcionalidade

3. pode-se obter uma abordagem unificada de tempo-real e tempo-não-real, e de programas e hardware

Um protótipo de um sistema modular baseado no UNIX, Sched, foi desenvolvido por Decker (Gancarz 1995). O sistema tinha diversas características notáveis, como diversidade de programas de síntese e geração de partituras, comunicação e troca de dados entre os programas e, principalmente, a estrutura de dados não sabe nada sobre o formato dos dados, e seu formato é auto-descritivo (Decker e Kendall 1984).

O problema com os programas monolíticos é que é necessário recompilar grandes seções de código para implementar novos algoritmos de síntese, além de haver a necessidade de se conhecer a estrutura do programa. Quanto maior o programa fica, mais difícil é para consertar bugs (Decker e Kendall 1984). Esse é um problema que só agora está sendo resolvido em programas como o Csound.

Um recurso muito importante é a possibilidade de se executar programas externos que interagem com o programa "principal" (Decker e Kendall 1984). A vantagem dessa abordagem é que os dados gerados por um programa podem ser lidos, filtrados, e transformados por outros programas, obtendo-se assim um ambiente de trabalho mais produtivo e flexível. A idéia é ter diversas ferramentas operando conjuntamente, ao invés de um programa único

e monolítico. Dessa maneira não é necessário ter que reinventar a roda e mais tempo pode ser gasto desenvolvendo programas para resolver problemas específicos.

Alguns programas tentam expandir os conceitos do Music N criando um sistema no topo de um compilador acústico já existente como o Csound (Barton-Davis 2001; Loureiro 1996; Debril e Lemoine 2000; Gogins 2000b; Gogins 1998; Piche e Burton 1998; Burton e Piché 1998a; Laurson 1999; Puxeddu 2001). A vantagem dessa abordagem é que não há necessidade de implementar mais um novo compilador acústico, podendo-se concentrar nos recursos a serem implementados. Contudo, em geral, esses programas não provêm uma plataforma escalável e acessível para interligar o programa ao compilador acústico (normalmente a comunicação com o compilador acústico é feita internamente); são ligados a um único sistema operacional; e em alguns casos não estão mais acessíveis. Outros programas procuram criar um ambiente de síntese completamente novo, implementando o compilador acústico como parte do sistema (Hanappe 1999; Oppenheim 1992; Oppenheim 1993; Oppenheim 1991a; Oppenheim 1991b; Oppenheim 1990; Oppenheim 1989; Assayag et al. 1997; Oppenheim 1987; Scaletti 1989; Wrighta et al. 1997; Brandon e Smith 2000; Jaffe 1991; Jaffe e Boynton 1989; Buxton et al. 1979). A vantagem dessa abordagem é que se pode conceber o compilador acústico desde o início para as tarefas desejadas do sistema. Contudo, como há mais recursos para implementar, esses sistemas acabam ficando incompletos.

CAPÍTULO 3

csoundXML: meta-linguagem para síntese sonora[1]

O csoundXML é uma meta-linguagem para síntese sonora escrita em XML, desenvolvida pelo autor deste trabalho com o propósito de descrever instrumentos de síntese em formato de texto estrutrado, conforme visto na seção 2.1.2.3 na página 17. Uma meta-linguagem é geralmente usada para definir ou descrever outra linguagem. O csoundXML descreve a linguagem de orquestra do Csound em XML com alguns acréscimos.

Seria ideal e altamente desejável que existisse uma única meta-linguagem de síntese que fosse capaz de descrever todos os algoritmos de síntese conhecidos, já que não existe um padrão genérico que independa de linguagens (Schiettecatte 2000b). Contudo essa linguagem é muito difícil de ser criada, se não impossível. Eric Scheirer, um dos principais criadores do *MPEG-4 Structured Audio* (Koenen 1999), comenta a que o objetivo inicial do *MPEG-4 Structured Audio* era funcionar como um tipo de formato intermediário entre qualquer programa de síntese,

> but it rapidly became clear that this idea is untenable. The different software synthesizers—Csound, SAOL, SuperCollider, Nyquist, and the commericial graphical

[1] Uma versão preliminar e reduzida deste capítulo foi originalmente publicada em (Kröger 2003a).

> ones—all have different underlying conceptions of events, signals, opcodes, and functions that makes it impossible to have a single format that captures anything but the very simplest aspects of behavior[2] (Scheirer 2000).

Já que uma linguagem universal para síntese não é viável, uma solução é criar um padrão e esperar que os programas de síntese o adotem ou suportem (Scheirer 2000), ou definir uma linguagem genérica que seja extensível e com algumas linguagems de "alvo" especificadas (Gogins 2000a). O SAOL (ISO/IEC 1999) é um exemplo da primeira solução enquanto o csoundXML um exemplo da segunda.

A idéia básica é que o csoundXML seja um subconjunto de um sistema completo para descrição de síntese, como será visto nos capítulos 4 e 5. Esse sistema utilizará o Csound para renderizar sons através do csoundXML.

Poder-se-ia utilizar um código genérico em XML como uma espécie de "pacote" (*wrapper*) para chamar funções e/ou instrumentos do Csound, mas a descrição de cada elemento de síntese em XML tem inúmeras vantagens, que serão vistas na seção 3.2.

3.1 Uma introdução ao XML

Aplicações XML têm sido usadas em diversas áreas como matemática (mathML), gráfica (VML e SVG), e programação (OOPML)[3]. Ainda que o número de aplicações XML para música tenha crescido consideravelmente com soluções para descrição e notação musical como o musicXML (Good 2001), MDL (Roland 2001), e EMNML (Mosterd 1999) os esforços para a síntese sonora são poucos e estão em estágio inicial de desenvolvimento como o Javasynth (Makela 2003) e FlowML (Schiettecatte 2000a). Com o XML é possível criar uma meta-linguagem para síntese sonora que pode funcionar como um formato de intercâmbio entre as linguagens já existentes. Não obstante, é importante que essa meta-linguagem seja baseada em princípios concretos e em programas já existentes evitando que o formato seja apenas algo abstrato e sem utilização prática.

[2] "mas rapidamente ficou claro que essa idéia era inatingível. Os diferentes programas sintetizadores—Csound, SAOL, SuperCollider, Nyquist, e os comerciais e gráficos—todos tem diferentes concepções de eventos, sinais, opcodes, e funções que tornam impossível que se tenha um único formato que captura tudo exceto os mais simples aspectos de comportamento".

[3] Uma lista completa pode ser vista em `http://www.oasis-open.org/cover/xml.html`.

O XML é uma linguagem de marcação para documentos contendo informação estruturada onde a marcação ajuda a identificar a estrutura do documento. O XML é um padrão definido pelo *World Wide Web Consortium* baseado no SGML. Similarmente ao HTML, o XML define elementos entre marcadores como `<exemplo>` e `</exemplo>`. A maior diferença é que no HTML os marcadores são sempre semânticos e fixos, enquanto no XML não há um conjunto de marcadores fixos e eles não são semânticos.

Dentre as vantagens do XML estão a possibilidade de criação de linguagens de marcação específicas, dados auto-explicativos, troca de dados entre diferentes aplicativos, dados estruturados e integrados (Harold 1999, pp. 6–8), além de haver inúmeros *parsers* disponíveis gratuitamente.

3.1.1 Sintaxe do XML

O XML, assim como o HTML, utiliza marcadores para delimitar certos aspectos de texto. No HTML, por exemplo, os marcadores `<b>` e `</b>` delimitam o texto que deve estar em negrito (ex. 3.1). O exemplo 3.2 mostra o resultado gráfico do ex. 3.1 como seria mostrado por navegadores de internet.

Exemplo 3.1 Exemplo de marcação em HTML

```
O negrito pode ser usado para dar <b>ênfase</b> ao texto.
```

Exemplo 3.2 Resultado do ex. 3.1

O negrito pode ser usado para dar **ênfase** ao texto.

O HTML possui um conjunto de marcadores pré-definidos que devem ser seguidos de acordo com sua finalidade original. O uso de um marcador como `<meu-marcador>` traria uma mensagem de erro ou seria ignorado.

O XML não possui nenhum marcador pré-definido, todos os marcadores são definidos pelo usuário. Outra diferença é que em HTML geralmente se descreve a *formatação* do texto com marcadores como `<b>` para negrito, `<i>` para itálico, etc, enquanto no XML procura-se descrever a *estrutura* do documento. O ex. 3.3 mostra o ex. 3.1 reescrito para usar outro

marcador, `enfase`. O marcador `enfase` não é definido no HTML, mas pode ser definido com XML para indicar que uma porção de texto deve ser enfatizada, com itálico, negrito, ou sublinhada. Essa é uma das vantagens de se descrever um documento com marcadores de estrutura ao invés de formatação; caso seja necessário alguma mudança de formatação, muda-se a maneira como a estrutura particular é mostrada. No nosso exemplo, modifica-se o marcador de ênfase para gerar texto sublinhado, e não em itálico.

Exemplo 3.3 Exemplo de marcação em XML

```
O negrito pode ser usado para dar <enfase>ênfase</enfase> ao texto .
```

Um uso típico de XML pode ser visto no ex. 3.4. O marcador `<livros>` contém informações sobre livros. A informação específica de cada livro é contida no marcador `<livro>`. Dentro desse marcador entram outras informações específicas como autor (marcador `<autor>`) e título (marcador `<titulo>`).

Exemplo 3.4 Arquivo típico de XML

```
<?xml version="1.0"?>
<livros>
  <livro codigo="1001">
    <autor>Machado de Assis</autor>
    <titulo>Helena</titulo>
  </livro>
  <livro codigo="1002">
    <autor>Machado de Assis</autor>
    <titulo>Dom Casmurro</titulo>
  </livro>
</livros>
```

O ex. 3.4 mostra algumas das características básicas de qualquer documento XML:

- a informação fica contida entre marcadores, ou *elementos.* O marcador `<marcador>` inicia um bloco enquanto o marcador do mesmo nome precedido por `/` encerra um bloco, como `</marcador>`.
- marcadores podem ser aninhados. No ex. 3.4 temos os blocos que definem os livros (marcador `<livro>`) dentro do marcador `<livros>`, e a informação de cada livro dentro do bloco que define um único livro (marcador `<livro>`). É através desse procedimento que a estrutura é criada em XML.

- a declaração XML, contida na primeira linha, identifica o documento como XML.
- os elementos podem ter informações anexadas em *atributos*, como nas linhas 3 e 7 do ex. 3.4. Os atributos têm um nome e valor associado, como `nome="valor"`. Mais sobre atributos será visto na seção 3.1.3.
- Os documentos XML devem ter um e apenas um elemento no nível mais alto. No ex. 3.4 o elemento de nível mais alto é `<livros>`.

3.1.2 Esquemas XML

A *Document Type Definition* (DTD) define a estrutura de um documento XML a partir de uma lista de elemento válidos. No ex. 3.5, por exemplo, tem-se uma estrutura organizada hierarquicamente em coral, acordes, e notas. Seria impróprio ter um acorde dentro de notas, por exemplo, como mostra o ex. 3.6, mas não há nada pré-definido na sintaxe do XML que impeça alguém de fazê-lo. Em geral os DTD são usados para *validar* um documento, ou seja, garantir que ele se mantenha dentro de certos limites válidos. No ex. 3.6 um DTD poderia ser criado para impedir que acordes sejam aninhados dentro de notas.

Exemplo 3.5 Acordes codificados em XML

```
<choral>
 <acorde>
    <nota nome="dó" />
    <nota nome="mi" />
    <nota nome="sol" />
 </acorde>
  <acorde>
    <nota nome="ré" />
    <nota nome="fá" />
    <nota nome="lá" />
  </acorde>
<choral>
```

3.1.3 Elementos e atributos

A sintaxe do XML é baseada em elementos e atributos. Um elemento do XML é uma unidade básica de informação como `<autor>Machado de Assis</autor>` e

Exemplo 3.6 Aninhamento de acordes dentro de notas

```
<acorde>
  <nota nome="dó">
    <acorde>
      <nota nome="dó"/>
    </acorde>
  </nota>
</acorde>
```

`<desligue/>`. Um elemento pode incluir informação textual como o "Machado de Assis" ou ser vazio como `<desligue/>`. Finalmente, os elementos podem incluir atributos como `<nota nome="dó"/>`. Esse último exemplo demonstra o princípio de compreensibilidade. O objeto codificado pode ser compreendido mesmo por alguém com pouco ou nenhum conhecimento de XML. O mesmo ocorre no ex. 3.7.

Exemplo 3.7 Acorde codificado em XML

```
<acorde>
  <nota nome="dó" oitava="1" />
  <nota nome="ré" oitava="1" />
  <nota nome="mi" oitava="1" />
</acorde>
```

Observe que o elemento `nota` é vazio; o valor de cada nota está codificado no atribuito `nome`. Mas nada impede que uma nota seja codificada como visto no ex. 3.8.

Exemplo 3.8 Outra codificação para nota em XML

```
<nota>
  <nome>dó</nome>
  <oitava>1</oitava>
</nota>
```

Não existe um consenso se dados devem ser armazenados em atributos ou elementos. Alguns argumentam que o DTD fornece mais recursos para ler atributos e portanto os atributos devem ser mais utilizados. Por outro lado, outros insistem que elementos são mais fáceis de editar e exibir em documentos (Marchal 2000, p. 111). Concordamos com Harold que "the data itself should be stored in elements. Information about data (meta-data) should be stored in attributes"[4] (Harold 1999, p. 101). A única excessão é quando os meta-dados

[4] "os dados deve ser armazenados em elementos. Informação sobre os dados (meta-dados) devem ser armazenados nos atributos".

devem ser estruturados. É muito mais adequado armazená-los em elementos devido a sua característica hierárquica (Harold 1999, p. 102).

3.2 Vantagens

Como foi visto na seção 3.1, dentre as vantagens de se usar o XML estão descrição estruturada, tags auto-explicativas, e facilidade de "parseamento". Algumas das vantagens de se descrever a linguagem de orquestra em XML são conversão para outras linguagens, banco de dados, *pretty-print* (ver seção 3.2.3, p.44), e ferramentas gráficas.

3.2.1 Conversão para outras linguagens

O XML tem sido usado com sucesso para criar meta-linguagens cujo principal propósito é a conversão para diferentes linguagens (Lemos 2001; Chicha, Defaix, e Watt 1999; Sarkar e Cleaveland 2001; Meyyappan 2000). O csoundXML funciona como um ponto de partida para a criação de instrumentos que podem ser convertidos para diversas linguagens de síntese, como Csound, cmix, etc. Ainda que o csoundXML não se assemelhe a uma "linguagem universal", ele foi pensado para ser compatível com a linguagem de orquestra do Csound e consequentemente outros programas da família do Music V.

3.2.2 Banco de dados

A existência de uma vasta coleção de instrumentos é uma das principais fontes de aprendizado do Csound. Agora que o número desses instrumentos passa dos 2000, faz-se necessária a criação de uma base de dados mais formal. O csoundXML permite a criação de tags de meta-informação como autor, descrição, localização, dentre outras. Essa informação pode facilmente ser extraída e manipulada, ao contrário da meta-informação inserida com comentários (ver seção 3.3.6).

3.2.3 *Pretty-print*

O *pretty-print* é muito mais do que um recurso "eye candy". A possibilidade de imprimir código do Csound com qualidade gráfica é uma necessidade de autores de artigos e livros. Tendo o código descrito em XML pode-se converter para Csound de inúmeras maneiras. Um simples exemplo é o uso de comentários. Pode-se escolher se os comentários serão ou não impressos, ou *como* serão impressos, se acima, abaixo, ou do lado do código, tudo isso sem intervenção manual (ver exemplos 3.9, 3.10, e 3.11).

Exemplo 3.9 Comentários acima

```
; some comment here
a1 oscil 10000, 440, 1
```

Exemplo 3.10 Comentários abaixo

```
a1 oscil 10000, 440, 1
; some comment here
```

Exemplo 3.11 Sem comentários

```
a1 oscil 10000, 440, 1
```

3.2.4 Ferramentas gráficas

Devido a sua descrição altamente estruturada e formal é possível descrever o instrumento em csoundXML graficamente de forma automática. Na verdade dois problemas básicos estão relacionados:

1. decisões de design para definir como certos elementos serão desenhados. Opcodes geradores de som como `oscil` são fáceis de lidar, bastando seguir paradigmas como do Patchwork (Pinkston 1995; Lent, Pinkston, e Silsbee 1989). Opcodes de conversão de dados e controle de fluxo são difíceis de serem representados eficientemente de forma gráfica.

2. algoritmos para distribuir os elementos (*patches* ou figuras) de síntese na tela sem colisão. Tendo resolvido o item anterior é necessário ter algoritmos "inteligentes" para permitir diferentes tipos de visualização e complexidade.

   Um exemplo prático pode ser visto na seção 7.2.

3.3 Sintaxe

3.3.1 Opcodes

O coração dos instrumentos do Csound são as *unidades geradoras*, implementadas como opcodes. O exemplo 3.12 mostra um uso típico para o opcode `oscil`, onde `afoo` é a variável do tipo `a` onde será armazenada a saída do oscil; `10000` é a amplitude; `440` é a freqüência, e `1` é o número da função que será acessada. O resto da linha depois do `;` é um comentário e será ignorado pelo Csound.

Exemplo 3.12 Opcode típico do Csound

```
afoo  oscil  10000, 440, 1  ; some comment here
```

No csoundXML os opcodes são descritos pelo elemento `opcode`, enquanto os seus parâmetros pelo elemento `par`. O nome dos opcodes e parâmetros é definido pelo atributo `name`. O atributo `id` define um nome único para um elemento. Ele também serve como ligação entre elementos, como variáveis (ver seção 3.3.2). O ex. 3.13 mostra como ficaria o código do ex. 3.12 escrito em csoundXML.

Informações sobre opcodes (como quantos e quais parâmetros) e parâmetros (como os tipos de valores possíveis) é definida na biblioteca de XML para Csound, CXL (ver capítulo 4). Uma espécie de referência cruzada entre o csoundXML e o CXL é feita através do atributo `name`.

O atributo `type` indica o tipo da variável (e.g. `k`, `i`, ou `a`). Dessa maneira as variáveis podem ter qualquer nome e o csoundXML se encarrega de iniciar o nome da variável com

Exemplo 3.13 Instrumento do Csound descrito em XML

```
<opcode name="oscil" id="foo"  type="a">
  <out id="foo_out"/>
  <par name="amplitude">
     <number>10000</number>
  </par>
  <par name="frequency">
     <number>440</number>
  </par>
  <par name="function">
     <number>1</number>
  </par>
  <comment>some comment here</comment>
</opcode>
```

a letra certa (linha 1 do ex. 3.13), como de praxe no Csound. Esse recurso é útil para a conversão automática entre variáveis.

Cada parâmetro pode ter três tipos de entrada, um valor númerico simples (e.g. “1”), uma variável (e.g. “iamp”), ou uma expressão (e.g. “iamp+1/idur”), que pode combinar os dois tipos anteriores e expressões matemáticas simples como adição, subtração, multiplicação, e divisão. Se a entrada for um valor númerico, o elemento `number` é usado para codificá-la (linha 4 do ex. 3.13). Caso a entrada seja uma expressão, o elemento `expr` é usado. Finalmente, se a entrada for uma variável o elemento `par` será vazio e a variável será definida pelo atributo `vvalue` (abreviação de *variable value*). O nome em `vvalue` deve ser o mesmo do atributo `id` do parâmetro ou variável definido por `defpar`.

Uma característica que salta aos olhos é a extrema prolixidade da versão “xmllificada”; nosso exemplo original (ex. 3.12) tem apenas 1 linha, enquanto a versão em csoundXML (ex. 3.13) tem 13! Uma vantagem dessa verbosidade extra é a possibilidade de fazer buscas mais completas. Ainda no ex. 3.13, um programa de desenhar funções poderia rapidamente ver quantas e quais funções um instrumento está usando procurando pelo atributo “function” na tag `<par>`. É importante ter em mente que, ao contrário do que parece a primeira vista, ter informação descrita de maneira estruturada no XML facilita o trabalho do programador e/ou usuário. Todo o processo de ler o documento XML, determinar a estrutura e propriedades dos dados, dividir esses dados em partes e passá-los para outros componentes é feito pelo

parser de XML. Existem inúmeros *parsers* disponíveis, tanto comercialmente (e.g. Xmlspy[5]) quanto gratuitamente (e.g. Expat[6] e XP[7]).

3.3.2 Parâmetros e variáveis

No Csound geralmente usam-se variáveis para definir os parâmetros dos opcodes. No csoundXML eles são definidos com o elemento `defpar`, e assim como os opcodes, têm os atributos `id` e `type` (ex. 3.14).

Exemplo 3.14 Definindo um parâmetro

```
<defpar id="gain" type="i">
  <default>20</default>
</defpar>
```

Um exemplo mais complexo pode ser visto no ex. 3.15 onde o parâmetro `gain` é definido. Uma breve descrição é inserida no elemento `description`, o valor padrão no elemento `<default>`, e o âmbito em `range`. Um exemplo de utilização é um aplicativo gráfico que pode extrair essa informação para automaticamente criar *sliders* para cada parâmetro.

Exemplo 3.15 Parâmetro com diversos dados

```
<defpar id="gain">
  <description>
    gain factor , usually between 0 − 1
  </description>
  <default>1</default>
  <range steps="float ">
    <from>0</from>
    <to>1</to>
  </range>
</defpar>
```

Um detalhe importante é que se o atributo `auto` for igual a “yes”, seu valor será o de um campo-p. Isso é, `<defpar id="notas"auto="yes"/>` no csoundXML é equivalente à `inota = p4` no Csound, com a diferença que o campo-p exato não é determinado pelo designer do instrumento e sim internamente pelo programa que implementar o

[5] Disponível em `http://www.xmlspy.com/`.
[6] Disponível em `http://www.jclark.com/xml/expat.html`.
[7] Disponível em `http://www.jclark.com/xml/xp/`.

csoundXML. Ganha-se flexibilidade com essa abordagem, já que na partitura os parâmetros serão referenciados pelo nome da variável e não pelo campo-p.

3.3.3 Controle de fluxo

O XML funciona surpreendentemente bem para descrever estruturas de controle de fluxo. Infelizmente a linguagem de orquestra do Csound é arcaica e largamente influenciada por linguagens como *Assembler* e *FORTRAN* (Pope 1993). Uma das principais consequências disso é a falta de construções estruturadas como em *C* e *Pascal*, geralmente usadas em controle de fluxo como `if then`. O Csound utiliza etiquetas (*labels*) para delimitar blocos de código (`highnote`, `lownote`, e `playit` no ex. 3.16)

Exemplo 3.16 Controle de fluxo no Csound (Vercoe 2001)

```
 if (iparam == 1) igoto highnote
2   igoto lownote

4 highnote:
    ifreq = 880
6   goto playit

8 lownote:
    ifreq = 440
10  goto playit

12 playit:
    print iparam
14  print ifreq
```

Contudo o csoundXML engloba as definições em blocos estruturados (ex. 3.17), encarregando-se de criar as etiquetas automaticamente.

3.3.4 Tipos de saída

O csoundXML define o elemento genérico `output` que engloba os opcodes de saída do Csound como `out`, `outs`, e `outq`. Dessa maneira o compositor pode, de antemão, definir como serão geradas as saídas em diferentes contextos, como mono ou estéreo. Isso é particularmente útil para fazer diferentes versões da mesma música, ou quando se quer gerar

Exemplo 3.17 Controle de fluxo no csoundXML

```
<if expression="iparam == 1">
  <then>
    <defpar id="ifreq">880</defpar>
  </then>
  <else>
    <defpar id="ifreq">440</defpar>
  </else>
</if>
<opcode id="print">
  <par name="value1">iparam</par>
</opcode>
<opcode id="print">
  <par name="value1">ifreq</par>
</opcode>
```

uma versão estéreo de uma música quadrafônica, por exemplo. É só indicar o tipo de saída e o programa se encarrega de escolher a opção certa. Até onde sabemos, esse conceito é único em linguagens de síntese. O elemento `output` contém o elemento `outtype`, que define o tipo de saída sonora. O `output` pode conter um ou mais `outtype`s. O atributo `name` determina o tipo de saída o `outtype`, como mono ou estéreo. Finalmente, o elemento `in` e o atributo `vvalue` codifica os sinais que estão sendo enviados para a saída (ex. 3.18).

Exemplo 3.18 Diferentes saidas

```
<output>
  <outtype name="mono">
    <in id="mono" vvalue="basic_out"/>
  </outtype>
  <outtype name="stereo">
    <in id="left" vvalue="basic_out"/>
    <in id="right" vvalue="basic_out"/>
  </outtype>
</output>
```

3.3.5 Funções

O elemento `deffunc` serve para descrever funções, um dos requisitos básicos para se trabalhar com o Csound. O atributo `name` indica o GEN a ser usado, enquanto o atributo `order` define um nome para a função (ex 3.19). Novamente, o programa se encarrega de converter o código para os opcodes `ftgen` ou `f`, de acordo com a necessidade. Assim como

opcode, deffunc usa o elemento par para definir seus parâmetros. O atributo name indica o tipo de parâmetro e funciona de maneira similar ao atributo de mesmo nome em opcode. Diferentemente do Csound, as funções no csoundXML são definidas em um arquivo à parte, de modo que possam ser reutilizadas quando necessário.

Exemplo 3.19 Definindo uma função

```
<deffunc name="10" id="clarinete">
  <time>0</time>
  <size>1024</size>
  <par name="partialStrength" order="1">1</par>
  <par name="partialStrength" order="3">.5</par>
  <par name="partialStrength" order="5">.3</par>
</deffunc>
```

3.3.6 Meta informação

Como vimos, o csoundXML permite a criação de meta-informação entre a informação principal. O parser se encarrega de extrair o que é necessário a cada aplicação. Elementos como author, description, e url ajudam a identificar o instrumento e facilitam na criação de bancos de dados de instrumentos.

Exemplo 3.20 Incluindo meta-informação

```
<author>Russell Pinkston</author>
<description>
  Basic vowel formant generation instrument . This uses 5
  fof units to simulate vowel formants . The formant data
  is taken from the Dodge book, pp. 230-231
</description>
<url>wwww.utexas.edu/cofa/music/ems/</url>
```

3.3.7 Expressões matemáticas

Como foi visto anteriormente, cada parâmetro pode ter três tipos de entrada, um valor númerico simples, uma variável, ou uma expressão. As expressões são definidas pelo elemento expr e as operações de soma, subtração, multiplicação e divisão pelos elementos plus, minus, times, div, respectivamente. O equivalente da expressão $2 + 2$ em csoundXML pode ser visto no ex. 3.21. Cada elemento da expressão é determinado por el.

Exemplo 3.21 Soma simples no csoundXML

```
<plus>
  <el>2</el>
  <el>2</el>
</plus>
```

Naturalmente expressões também podem ter variáveis. No csoundXML as variáveis em expressões são definidas da mesma maneira que em opcodes, com o atributo `vvalue` (ex. 3.22). Dessa maneira o *parser* do csoundXML pode rapidamente determinar quantas variáveis um instrumento possui, apenas procurando pelos elementos que possuem o atributo `vvalue`.

Exemplo 3.22 Expressão com variável

```
<expr>
  <div>
    <el>1</el>
    <el  vvalue="dur"/>
  </div>
</expr>
```

3.4 Solução mista

Uma solução intermediária para diminuir a prolixidade do XML seria entrar os parâmetros de um opcode como atributos, como visto no ex. 3.23. Nessa solução a estrutura básica do instrumento—a ligação entre os opcodes—é mantida em XML, mas os dados de configuração de cada opcode, i.e. os paramêtros, são mantidos em listas.

Exemplo 3.23 Definindo parâmetros como listas

```
<opcode name="oscil" parameters="1000,440,1">
```

Aparentemente essa solução é uma boa maneira de se chegar a um meio-termo entre as facilidades do XML e contornar sua prolixidade. Contudo é justamente a prolixidade do XML, aliada a uma sintaxe rígida, que permite que parsers genéricos sejam utilizados (Roland 2001; Marx 2002). No exemplo 3.23 seriam necessários dois parsers, um para a

estrutura do XML e outro para as informações dos parâmetros, codificados em lista. Sendo que esse último precisaria ter um registro de todos os opcodes, seus respectivos parâmetros, e a ordem em que eles devem aparecer. Já no ex. 3.13, onde apenas XML é utilizado, o parser não precisa saber nada sobre os parâmetros, porque qualquer dado necessário já está implícito no atributo `name`.

3.5 Exemplo de instrumento

Um instrumento completo escrito em csoundXML pode ser visto no ex. 3.24 e sua representação em fluxograma na fig. 3.1. Observe que apenas uma váriavel (`dur`) é definida e cinco campos-p são designados automaticamente (através do atributo `auto`). Esses recursos de automação permitem que instrumentos sejam descritos de maneira mais genérica e sem depender tanto de recursos específicos de cada linguagem, no caso, o Csound. O elemento `instr` define um instrumento.

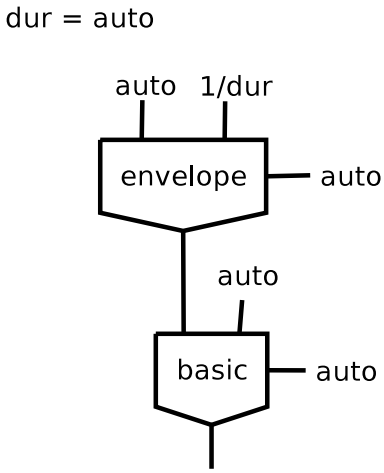

Figura 3.1: Instrumento simples em csoundXML

3.6 Exemplo de DTD

O exemplo 3.26 mostra o DTD básico usado para validar instrumentos do csoundXML. O comando `ELEMENT` define que elementos um elemento pode conter. Por exemplo, o

Exemplo 3.24 Instrumento simples em csoundXML

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE instr SYSTEM "instr.dtd">
<instr name="01_01_1">
  <defpar id="dur" name="duration" reserved="yes"/>
  <opcode id="envelope" name="oscili" type="a">
    <out id="envelope_out"/>
    <par name="amplitude" reserved="yes">
      <description>Amplitude value</description>
    </par>
    <par name="frequency">
      <expr>
        <div>
          <el>1</el>
          <el vvalue="dur"/>
        </div>
      </expr>
    </par>
    <par name="function" auto="yes">
      <description>Function with envelope shape</description>
    </par>
  </opcode>
  <opcode id="basic" name="oscili" type="a">
    <out id="basic_out"/>
    <par name="amplitude" vvalue="envelope_out">
      <description>Amplitude value</description>
    </par>
    <par name="frequency" auto="yes"/>
    <par name="function" auto="yes">
      <description>Function with waveshape</description>
    </par>
  </opcode>
  <output>
    <outtype name="mono">
      <in id="mono" vvalue="basic_out"/>
    </outtype>
    <outtype name="stereo">
      <in id="left" vvalue="basic_out"/>
      <in id="right" vvalue="basic_out"/>
    </outtype>
  </output>
</instr>
```

elemento `instr` pode conter os elementos `opcode`, `defpar`, `description`, `output`, e `author` (ex. 3.25).

Exemplo 3.25 O elemento `instr`

```
<!ELEMENT instr (opcode | defpar | description | output | author | url)*>
```

O comando `ATTLIST` define os atributos de um elemento. Em `<!ATTLIST`

`defpar id ID #REQUIRED>` o atributo `id` do elemento `defpar` é do tipo `ID` e é obrigatório.

Exemplo 3.26 DTD para instrumentos do csoundXML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT instr (opcode | defpar | description | output | author | url)*>
<!ELEMENT opcode (out | par | description)*>
<!ELEMENT range (from,to)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT defpar (description | default | range)*>
<!-- description sempre tem que vir depois de expr ou number -->
<!ELEMENT par ((expr | number)?, description?)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT default (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT div (el | div | times | plus | minus | expr)*>
<!ELEMENT times (el | div | times | plus | minus | expr)*>
<!ELEMENT plus (el | div | times | plus | minus | expr)*>
<!ELEMENT minus (el | div | times | plus | minus | expr)*>
<!ELEMENT expr (el | div | times | plus | minus | expr)*>
<!ELEMENT el (#PCDATA)>
<!ELEMENT out EMPTY>
<!ELEMENT in (expr)?>
<!ELEMENT outtype (in)*>
<!ELEMENT comment (#PCDATA)>
<!ELEMENT if (if | then | else)*>
<!ELEMENT then (if | then | else)*>
<!ELEMENT else (if | then | else)*>
<!ELEMENT author (#PCDATA)>
<!ELEMENT output (outtype)*>
<!ATTLIST defpar id ID #REQUIRED>
<!ATTLIST defpar type CDATA #IMPLIED>
<!ATTLIST el vvalue IDREF #IMPLIED>
<!ATTLIST in id ID #REQUIRED>
<!ATTLIST in vvalue IDREF #IMPLIED>
<!ATTLIST instr name CDATA #REQUIRED>
<!ATTLIST opcode id ID #REQUIRED>
<!ATTLIST opcode name CDATA #REQUIRED>
<!ATTLIST opcode type CDATA #IMPLIED>
<!ATTLIST out id ID #REQUIRED>
<!ATTLIST outtype name CDATA #REQUIRED>
<!ATTLIST par auto CDATA #IMPLIED>
<!ATTLIST par name CDATA #REQUIRED>
<!ATTLIST par vvalue IDREF #IMPLIED>
<!ATTLIST par reserved CDATA #IMPLIED>
<!ATTLIST defpar reserved CDATA #IMPLIED>
<!ATTLIST defpar name CDATA #IMPLIED>
<!ATTLIST range steps CDATA #IMPLIED>
```

CAPÍTULO 4

CXL: biblioteca de XML para csound

A biblioteca de XML para Csound (CXL) é uma biblioteca desenvolvida pelo autor deste trabalho com descrição em alto-nível dos opcodes e parâmetros do Csound. Essa biblioteca descreve os opcodes e parâmetros não apenas dizendo como eles devem ser parseados mas descrevendo como cada opcode se comporta, e que tipo de entrada é esperada, i.e. se em graus ou em decibéis. Esse recurso pode ajudar na criação de programas mais "inteligentes", como assistentes para ajudar na criação de instrumentos do Csound. O assistente poderia, por exemplo, interpretar um valor e determinar pelo contexto sua validade, além de poder sugerir valores válidos.

A CXL tem uma sintaxe similar à do csoundXML, exceto que enquanto este é usado para descrever instrumentos, aquela é usada para descrever opcodes e parâmetros do Csound.

4.1 CXL e descrição gramatical

Tradicionalmente os programas que definem uma linguagem (como linguagem de programação e linguagens para música como csound) utilizam um analizador (*parser*) gramatical que analisa a linguagem descrita como uma *gramática livre de contexto.* O sistema formal mais comum para representar as regras de descrição é a forma Backus-Naur (BNF). O *parser* lê a entrada em sequências usando uma regra gramatical; se a entrada é válida ele executa uma função associada àquela regra, por exemplo, uma função de soma na sequência `2 + 2`; se a entrada for inválida o *parser* retorna uma mensagem de erro.

Uma linguagem de programação para música como Csound necessita de um *parser* para verificar se a entrada está sintaticamente correta. O problema é que cada *parser* é definido de acordo com o programa e a linguagem de programação usada. Pode-se fazer um *parser* manualmente, usando sequências de `if then` ou usar um gerador de *parser* como o Bison. No mundo XML os DTD descrevem a sintaxe dos arquivos XML.

O CXL, similarmente aos geradores de *parser* e a DTD, descreve a sintaxe dos opcodes do Csound, contudo em um nível mais alto. Uma das vantagens dessa abordagem é que a informação da CXL pode ser convertida e usada por diferentes programas já que ela independe de linguagens de programação específicas. Outra vantagem é que a CXL pode conter diferentes dipos de meta-data, inclusive a propria documentação do opcode.

4.2 Opcodes

Como pode ser visto no ex. 4.1 o marcador para definir opcodes é `<defOpcode>`. O atributo `name` indica qual opcode está sendo definido. Os parâmetros são definidos com o elemento `<par>`. Observe que cada parâmetro tem um atributo `order` que identifica a ordem em que os parâmetros devem ocorrer no Csound. A CXL é específica para o Csound e segue a maneira como seus opcodes são declarados. Na criação de uma biblioteca para outra linguagem a definição de alguns opcodes pode ser a mesma do Csound mas com ordem diferente.

4.3 Sintaxe básica

O atributo `type` indica o tipo da variável, se `k`, `a`, ou `i`. Se um parâmetro tiver mais de um tipo de variável, pode-se juntá-las como `type="ak"` na linha 6 do ex. 4.1. A ordem das variáveis não importa, poderia ser `type="ka"`. O atributo `optional` indica se o parâmetro é opcional. Seu valor padrão é `optional="no"`, ou seja, se um parâmetro não tem nenhuma indicação ele é requerido.

O elemento `out` define os parâmetros de saída do opcode (ver ex. 4.2) e o atributo `outnumber` define a quantidade de saídas. A maioria dos opcodes do Csound tem apenas uma saída, mas alguns tem saídas múltiplas como `convolve` e `inq`.

Finalmente, o elemento `<description>` contém uma breve descrição do opcode.

Exemplo 4.1 Definição do opcode `oscil` em CXL

```
<defOpcode name="oscil">
  <par order="1" type="x" name="amplitude">10000</par>
  <par order="2" type="x" name="frequency">440</par>
  <par order="3" type="i" name="function">1</par>
  <par order="4" type="i" optional="yes" name="phase">0</par>
  <out type="ak" outnumber="1"/>
  <description>
    Creat a basic oscilator
  </description>
</defOpcode>
```

A figura 4.1 mostra o mapeamento do opcode `oscil` do Csound para CXL.

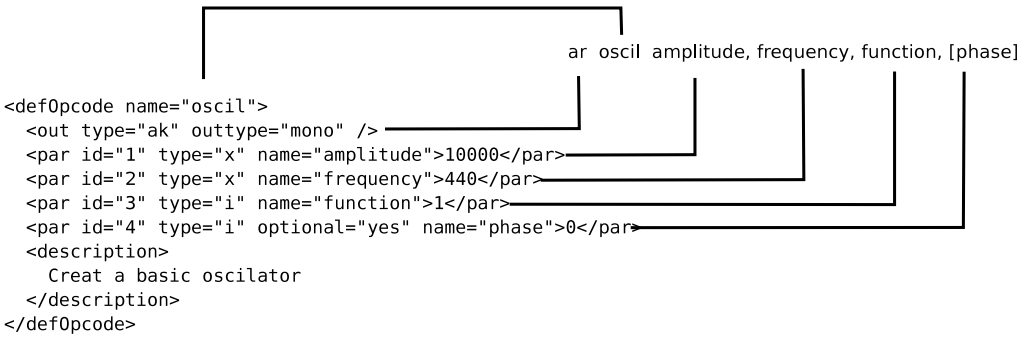

Figura 4.1: Mapeamento do opcode `oscil` do Csound para CXL

Como visto anteriormente, o elemento `<out>` determina as características da saída do opcode. O atributo `outnumber` indica o números de canais possíveis no opcode. Cada

canal pode ser configurado com o elemento `<channel>`. No ex. 4.2 os canais 2, 3, e 4 são opcionais, enquanto o canal 1 é obrigatório.

Exemplo 4.2 Definição do opcode `convolve` em CXL

```
<defOpcode name="convolve">
  <par order="1" type="a" name="audioInput"/>
  <par order="2" type="i" name="impulseFile"/>
  <par order="3" type="i" optional="yes" name="whichChannel"/>
  <out type="a" outnumber="4">
    <channel order="1" />
    <channel order="2" optional="yes" />
    <channel order="3" optional="yes" />
    <channel order="4" optional="yes" />
  </out>
  <description>
    Convolves a signal and an impulse response
  </description>
</defOpcode>
```

Os atributos `fromId` e `toId` permitem uma configuração mais fácil dos canais. Eles selecionam um âmbito de canais que devem ter a mesma configuração, como visto no ex. 4.3. Isso é particularmente útil quando o opcode lida com diversos números de canais, como o `in32` que espera 32 canais de entrada. Entrar cada configuração de canal separadamente seria exagerado.

Exemplo 4.3 Configuração de canais

```
<out type="a" outnumber="4">
  <channel order="1" />
  <channel fromId="2" toId="4" optional="yes" />
</out>
```

Naturalmente mais dados podem ser inseridos, como valores padrão (valores que serão automaticamente selecionados se nenhum valor for alimentado), exemplos, *presets*, e mais dados sobre o opcode como autor, data de criação, revisão, e assim por diante.

4.4 Tipos de dados

Os valores dentro do atributo `name` são na verdade *tipos de dados* definidos pelo elemento `defpar`. Por exemplo, `amplitude` em `<par name="amplitude"/>`. A

definição desses tipos de dados permite estabelecer o tipo de parâmetro que cada opcode recebe.

O elemento `defpar` define os elementos `description`, que contém uma breve descrição do tipo de dado definido; `range`, que especifica o âmbito de valores do tipo de dado; e `default`, que define um valor padrão para ser usado caso nenhum valor seja definido pelo usuário (ex. 4.4).

Exemplo 4.4 Definição de um tipo de dado

```
<defpar name="clockNumber">
  <description>
    There are 32 clocks numbered 0 through 31.
    All other values are mapped to clock number 32
  </description>
  <range from="0" to="31"/>
  <default>0</default>
</defpar>
```

4.5 Conversão de csoundXML para csound

A principal tarefa do CXL é auxiliar na conversão de instrumentos do csoundXML para o Csound. Por exemplo, para converter o instrumento do ex. 4.5 para o Csound o *parser* primeiramente verifica se o opcode está definido na biblioteca CXL (através do atributo `name`, na linha 1), em seguida determina se os parâmetros são válidos (também através do atributo `name`), e em que ordem devem aparecer, através do atributo `order` do CXL (ex. 4.6). Uma visão mais completa de como o csoundXML e a CXL se relacionam é vista no capítulo 7.

4.6 Conclusão

A CXL permite criar programas que “entendam” a sintaxe do Csound em um nível mais alto. Isso pode ser útil na criação de assistentes, por exemplo, e está sendo fundamental no desenvolvimento dos mega-instrumentos (capítulo 5, p. 61). É também possível separar

Exemplo 4.5 Instrumento em csoundXML

```
<opcode name="oscil" id="foo" type="a">
  <out id="foo_out"/>
  <par name="amplitude">
    <number>10000</number>
  </par>
  <par name="frequency">
    <number>440</number>
  </par>
  <par name="function">
    <number>1</number>
  </par>
  <comment>some comment here</comment>
</opcode>
```

Exemplo 4.6 Definição do opcode `oscil` em CXL

```
<defOpcode name="oscil">
  <par order="1" type="x" name="amplitude">10000</par>
  <par order="2" type="x" name="frequency">440</par>
  <par order="3" type="i" name="function">1</par>
  <par order="4" type="i" optional="yes" name="phase">0</par>
  <out type="ak" outnumber="1"/>
  <description>
    Creat a basic oscilator
  </description>
</defOpcode>
```

o conteúdo da forma, utilizando o csoundXML para descrever *apenas* o conteúdo e o CXL para estabelecer a forma.

CAPÍTULO 5

Mega-instrumentos

5.1 Introdução

Desenvolvemos o conceito de mega-instrumento com o intuito de resolver alguns dos problemas colocados na seção 2.1.2.1 (pg. 9). O mega-instrumento é uma forma de descrever instrumentos usando blocos em diferentes níveis (layers). O termo foi tomado por empréstimo da programação para GUI, onde um *mega-widget* é uma coleção de *widgets* trabalhando juntos. Um exemplo disso é a coleção de *mega-widgets* implementados em *[incr TCL]* (Smith 2000; Welch 1999; Ousterhout 1993).

Em geral algoritmos de síntese são implementados em um bloco único, ou seja, em um único *instrumento.* A fig. 5.1 mostra a implementação de Dodge para o instrumento de cordas de Schottstaedt (Dodge e Jerse 1997, p. 125).

Podemos observar na figura 5.1 que o instrumento é constituído de quatro partes básicas, uma gera o ruído de ataque (*attack-noise*), outra o vibrato (*vibrato*), outra o envelope geral (*envelope*), e finalmente outra gera o timbre geral de cordas através de FM (*fm-instrument*). A idéia dos mega-instrumentos é poder definir blocos menores de instrumentos

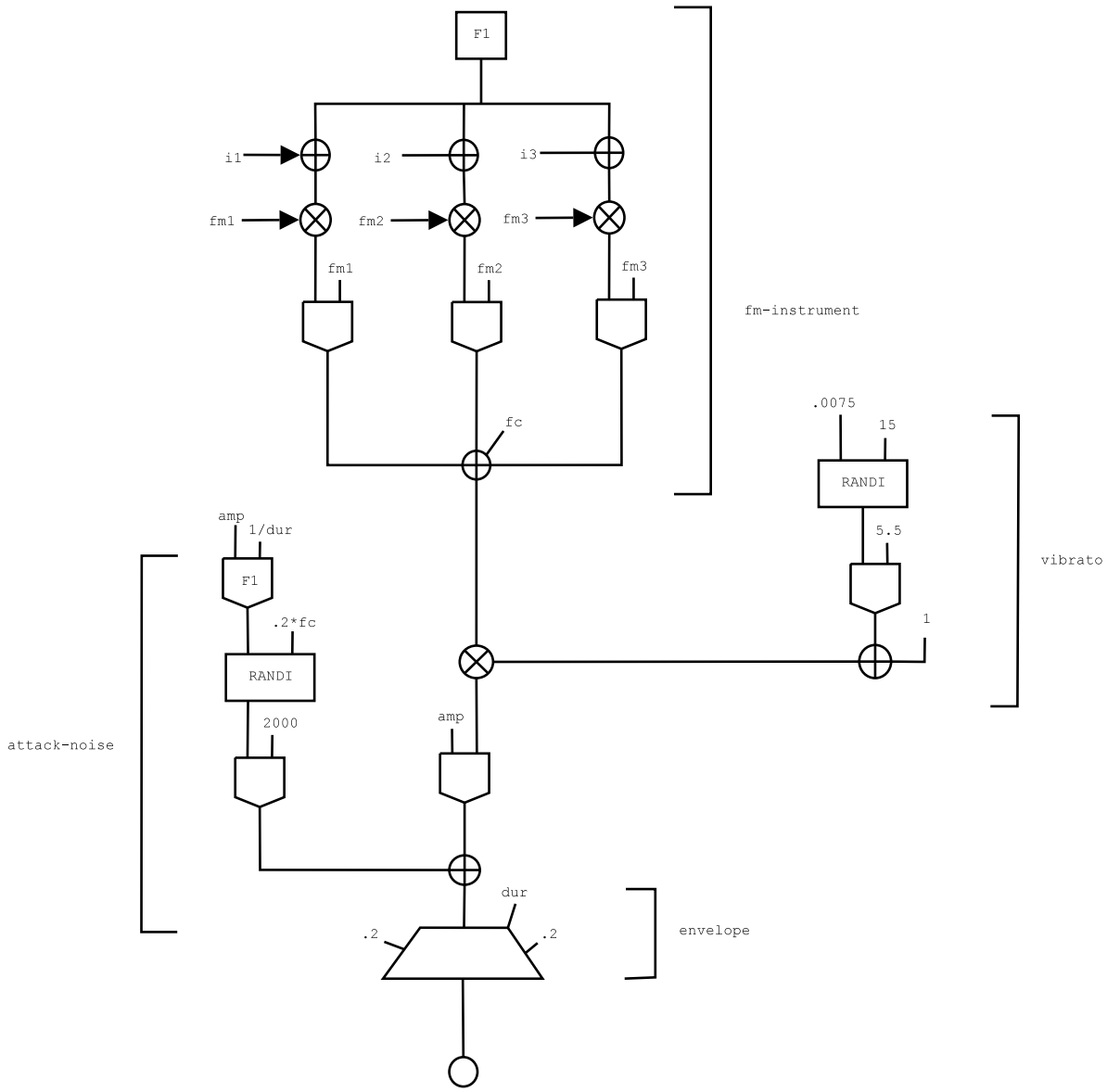

Figura 5.1: Instrumento em um bloco único

e conectá-los em instrumentos maiores, como visto na fig. 5.2. Cada “caixa” representa a implementação separada de uma parte constituinte do mega-instrumento. Uma das vantagens dessa abordagem é que os blocos podem facilmente ser substituídos. O gerador de vibrato, por exemplo, poderia ser substituído por outro ao gosto do compositor.

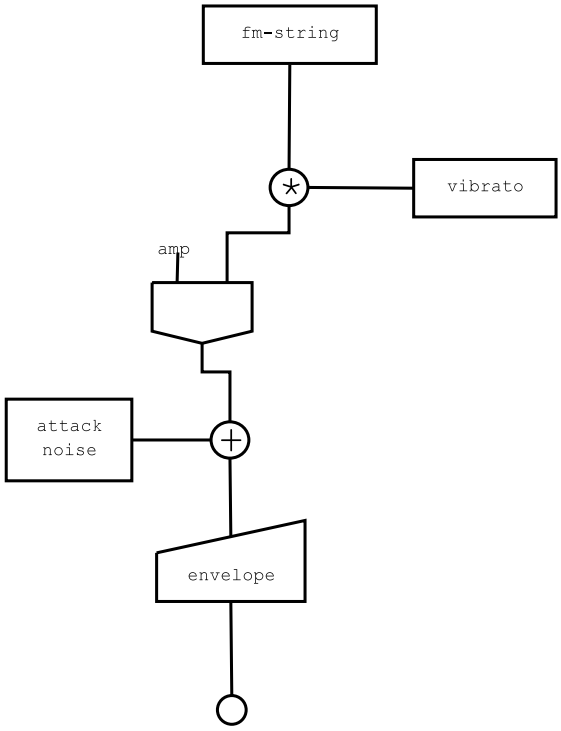

Figura 5.2: Instrumento dividido em blocos

5.2 Interface

A interface para o mega-instrumento pode se dar de diferentes maneiras, daí o poder do XML. O XML apenas *descreve* o mega-instrumento, o programa gráfico decidirá como essa informação será mostrada. O programa pode mostrar essa informação como um fluxograma (como o *patchwork* (Pinkston 1995)), uma estrutura em lista de notas (como o Csound), ou uma estrutura mais descritiva como:

```
schottstaedt-string -carrier 440 -modulator 440
-envelope {.4 1 .1 4},
```

tudo isso sem alterar uma única linha do mega-instrumento. E, assim como na programação orientada a objetos, o aninhamento dos dados permite que o usuário/compositor possa usar o instrumento pensando em termos como `envelope` ou `carrier`, ao invés de *campos-p*.

5.3 Programas externos

O mega-instrumento é um conceito implementado em XML, incorporando a CXL e o csoundXML. Mas ele não é usado apenas para definir opcodes do csound. Ele pode utilizar programas externos, como um *mixer*, ou um programa de análise. Dessa maneira mega-instrumentos complexos podem ser criados, por exemplo, para executar um programa de análise e criar automaticamente a resíntese, integrando o resultado com outros opcodes. Programas externos podem ser executados como se fossem opcodes do Csound. A saída de um opcode pode ser a entrada de um desses programas ou vice-versa, as possibilidades são praticamente ilimitadas.

5.4 Sintaxe

O mega-instrumento herda toda a sintaxe do csoundXML com a diferença que enquanto este define um instrumento através de uma coleção de opcodes, aquele define *mega-instrumentos* através de uma coleção de *instrumentos.*

O elemento raiz do mega-instrumento é `mi`, que pode conter um ou mais instrumentos definidos pelo elemento `instr` (ex. 5.1).

Exemplo 5.1 Mega-instrumento

```
<mi>
  <instr name="adsyn" loadpar="all">
    <par name="filename">
      <in id="hetro_output"/>
    </par>
  </instr>
</mi>
```

O atributo `loadpar` indica se os parâmetros do instrumento devem ser “carregados” nesse mega-instrumentos. Esse esquema é similar à herança na programação orientada a objetos. Assim como no `csoundXML`, a conexão entre instrumentos é dada pelo elemento `in`.

Um outro tipo de arquivo pode ser definido além dos instrumentos, os *programas.* Os

programas são armazenados em um arquivo específico mas se integram ao mega-instrumento como se fossem instrumentos.

Cada programa é definido pelo elemento raiz `program` e pelos elementos `tag`. Tipicamente um programa de linha de comando *à la* UNIX tem opções de execução como `programa -c 1 -d 2` onde os elementos com um traço indicam o tipo de configuração seguido por seu valor. O elemento `tag` descreve cada uma dessas opções, como pode ser visto no ex. 5.2.

Exemplo 5.2 Definindo opções de execução

```
<tag name="c" id="channel">
  <default>10000</default>
  <description>
    channel number sought. The default is 1.
  </description>
</tag>
```

5.5 Exemplo

Provavelmente o conceito de mega-instrumento e sua sintaxe serão melhor compreendidos através de um exemplo. Nós implementaremos um mega-instrumento que utiliza o programa de análise Hetro, que recebe como entrada um arquivo de som, o decompõe em componentes senoidais e gera um arquivo com a descrição dos componentes na forma de faixas de amplitude e freqüência. Esse arquivo gerado é geralmente utilizado pelo opcode `adsyn` para fazer síntese aditiva. Nosso mega-instrumento vai unir ambos, o programa e o opcode em um único (mega-)instrumento que aceita um arquivo de som como entrada e retorna o som resintetizado. Os parâmetros do mega-instrumento serão os parâmetros do hetro e do opcode `adsyn` combinados.

O instrumento `adsyn` é um instrumento do csoundXML comum e pode ser visto no ex. 5.3.

O programa `hetro` mapeia cada opção de execução de Hetro em atributos `tag`, como mostra o ex. 5.5.

Exemplo 5.3 Instrumento `adsyn`

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE instr SYSTEM "instr.dtd">
<instr name="adsyn">
  <opcode name="adsyn" id="foo" type="a">
    <out id="foo_out" />
    <par name="ampfactor" auto="yes"/>
    <par name="pitchfactor" auto="yes"/>
    <par name="timefactor" auto="yes"/>
    <par name="filename" auto="yes"/>
  </opcode>

  <output>
    <outtype name="mono">
      <in id="mono" vvalue="foo_out"/>
    </outtype>
    <outtype name="stereo">
      <in id="left" vvalue="foo_out"/>
      <in id="right" vvalue="foo_out"/>
    </outtype>
  </output>
</instr>
```

Exemplo 5.4 mega-instrumento `additive`

```
<?xml version="1.0" encoding="utf-8"?>
<mi name="additive">
  <instr name="hetro" loadpar="all">
    <par name="output" noexport="yes"/>
    <out id="hetro_output"/>
  </instr>

  <instr name="adsyn" loadpar="all">
    <par name="filename">
      <in id="hetro_output"/>
    </par>
  </instr>
</mi>
```

Finalmente, o mega-instrumento `additive` (ex. 5.4) utiliza tanto `hetro` quanto `adsyn` como instrumentos comuns (fig. 5.3). Todos os parâmetros de ambos instrumentos são carregados (`loadpar="all"`) com excessão do “output” no `hetro`. Todos os parâmetros de `hetro` estarão acessíveis ao usuário, mas esse parâmetro será conectado à entrada de `adsyn`. O atributo `noexport` previne a exportação de um parâmetro.

Exemplo 5.5 Programa `hetro`

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<program name="hetro">
  <defpar id="input"/>
  <defpar id="output"/>
  <tag name="s" id="samplerate">
    <default>10000</default>
  </tag>
  <tag name="c" id="channel">
    <default>10000</default>
  </tag>
  <tag name="b" id="begin">
    <default>0.0</default>
  </tag>
  <tag name="d" id="duration">
    <default>0.0</default>
  </tag>
  <tag name="f" id="begfreq">
    <default>100</default>
  </tag>
  <tag name="h" id="partials">
    <default>10</default>
  </tag>
  <tag name="M" id="maxamp">
    <default>32767</default>
  </tag>
  <tag name="m" id="minamp">
    <default>128</default>
  </tag>
  <tag name="n" id="brkpts">
    <default>256</default>
  </tag>
  <tag name="l" id="cutfreq">
    <default>0</default>
  </tag>
</program>
```

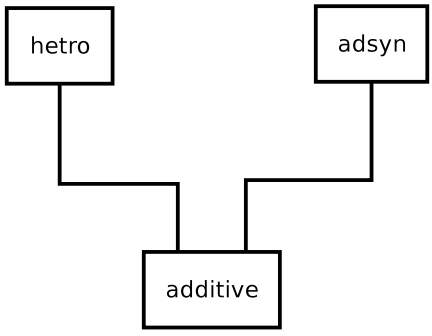

Figura 5.3: O mega-instrumento additive “herda” as características de hetro e adsyn

CAPÍTULO 6

Descrição hierárquica e modular de eventos[1]

6.1 Introdução

Conforme visto anteriormente, o Csound e os programas da família Music N usam dois arquivos, a orquestra, onde são definidos os instrumentos, e a partitura, que contém listas de notas. Tradicionalmente utiliza-se apenas uma partitura "monolítica" para toda uma composição. Infelizmente os programas da família Music N como o Csound não possuem nenhum recurso para compilar[2] apenas trechos separados. Toda a música é compilada, mesmo que apenas um trecho tenha sido modificado. Contudo não é muito produtivo esperar que toda a composição seja compilada apenas para ouvir uma curta seção em que se está trabalhando.

Nesse capítulo apresentaremos algumas soluções dividindo a partitura em partes menores e utilizando o utilitário para recompilação *make*. O objetivo é reduzir o tempo de *recompilação* ao mínimo, ou seja, provavelmente a primeira vez em que a composição for

[1] Uma versão preliminar desse capítulo foi originalmente publicada em (Kröger 2003b).

[2] Nesse trabalho o termo "compilar" é usado como sinônimo para "renderizar", i.e., executar o Csound para obter um arquivo sonoro.

compilada levará o tempo habitual, mas as subseqüentes compilações terão seu tempo reduzido.

Finalmente, será introduzida uma solução que envolve o uso de eventos, uma estrutura entre a lista de notas e a seção. A definição de eventos não só resolve elegantemente o problema anterior como possibilita criar partituras com estrutura hierárquica. Além de poder definir blocos de eventos, pode-se também determinar relações de tempo entre esses eventos.

6.2 Soluções

As soluções aqui apresentadas baseam-se na divisão da partitura em partes menores. Em algumas essa divisão é efetuada manualmente enquanto em outras é feita automaticamente.

6.2.1 Divisão manual

Uma maneira primitiva de selecionar seções para se compilar é comentar aquilo que ficará de fora. Ainda que esse procedimento funcione com pequenos arquivos, é impraticável para arquivos maiores, com centenas de linhas. Uma variante dessa solução é usar o comando `#include` do Csound. Seções são salvas em arquivos separados que são chamadas em um arquivo principal com o `#include` (ex. 6.1). As seções que não serão compiladas podem facilmente ser "comentadas". Por exemplo, a seção 3 no ex. 6.1 não será gerada.

Exemplo 6.1 Arquivo de partitura principal

```
1 #include :section-1.sco:
2 s
3 #include :section-2.sco:
4 s
5 #include :section-3.sco:
6 e
```

A vantagem desse procedimento é que utiliza o próprio Csound, não precisando de ferramentas externas; e pode-se compilar seções arbitrárias (e.g., seções 1 e 3).

Dentre as maiores desvantagens, as seções escolhidas são sempre recompiladas *in toto*,

mesmo que nada tenha sido modificado; tem-se mais arquivos para gerenciar; e o comando `#include` tem um longo histórico de bugs.

Utilizando o make. A mesma versão anterior pode ser muito mais automatizada com make. A mesma estrutura de arquivos do ex. 6.1 é mantida, porém o arquivo principal de partitura é descartado.

O utilitário `make` foi criado para automatizar o processo de compilação de programas. Ele é capaz de recompilar apenas os arquivos necessários baseado nos arquivos fontes que foram modificados. Apesar de ser largamente usado para gerenciar programas de computador, "make is not limited to programs. You can use it to describe any task where some files must be updated automatically from others whenever the others change"[3] (Stallman e McGrath 1998, p. 1).

Uma descrição mais detalhada do make está fora do escopo deste texto. Por hora é suficiente saber que ele executa "regras" que são descritas em um arquivo geralmente chamado `makefile` ou `Makefile`. As regras do make têm o formato visto no exemplo 6.2, onde

> a target is usually the name of a file that is generated by a program, ... a dependency is a file that is used as input to create the target. A target often depends on several files, ... a command is an action that make carries out. A rule may have more than one command, each on its own line[4] (Stallman e McGrath 1998, p. 3).

Exemplo 6.2 Regra do make

```
1  target: dependencies
2          command
```

O ex. 6.3 mostra um exemplo simples de uso do make. Tendo definido as regras é só digitar `make sec1.wav` no terminal. Dessa maneira cada tag para cada seção gerará um arquivo wav separado.

[3] "make não é limitado a programas. Você pode usá-lo para descrever qualquer tarefa onde alguns arquivos devem ser atualizados automaticamente a partir de outros em qualquer momento que esses outros arquivos sejam modificados".

[4] "um alvo é geralmente o nome de um arquivo que é gerado por um programa, ... uma dependência é um arquivo que é usado como entrada para criar o alvo. Um alvo freqüêntemente depende de inúmeros arquivos, ... um comando é uma ação que o progama make executa. Uma regra pode ter mais que um comando, cada um em sua própria linha".

Exemplo 6.3 Regra para uma seção

```
1 sec1.wav: sec1.sco
2    csound -Wo sec1.wav Main.orc sec1.sco
```

Como não temos mais a partitura principal precisamos de algum modo para mixar as seções juntas. Uma maneira é usar o utilitário *mixer* que vem com o Csound ou algum outro programa como o *ecasound* ou *sox*. No exemplo 6.4 podemos ver o poder real do make em ação. A regra `Main.wav` é feita, tendo as regras de seção sec1.wav, sec2.wav, e sec3.wav como dependências. Isso significa que, para efetuar o comando, as três dependências tem que estar completas. Caso alguma não esteja o make automaticamente compilará **apenas** as seções que faltam. Uma visão geral do processo pode ser vista na fig. 6.1.

Exemplo 6.4 Mixador

```
1 Main.wav: sec1.wav sec2.wav sec3.wav
2    mixer -T 0 sec1.wav \
3    -T 120 sec2.wav -T 160 sec3.wav
```

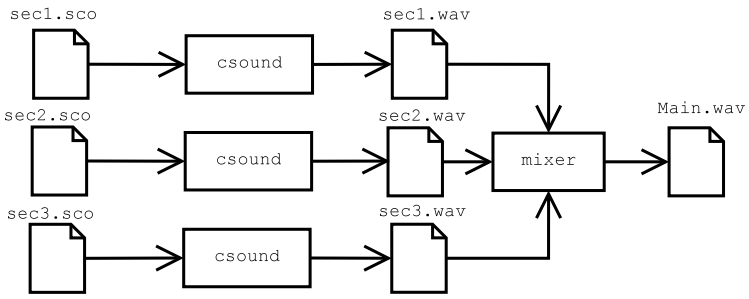

Figura 6.1: Visão geral do processo

Conclusão. Como pudemos ver, separando a partitura em arquivos diferentes e utilizando o make podemos compilar cada seção independentemente obtendo mais flexibilidade e velocidade. Essa segunda solução apresenta um grande avanço em relação à primeira, já que o tempo de renderização não é o menor. Se for recompilar as seções 1 e 3 e apenas a seção 1 tiver sido modificada, a seção 3 também será recompilada. Contudo, em ambas soluções tem que se nomear arquivos manualmente. Esse problema se torna particularmente agudo quando se deseja inserir uma seção entre duas seções já existentes. Os arquivos tem que ser renomeados, assim como as regras do make tem que ser modificadas para comportar a

mudança.

6.2.2 Divisão automática

Na seção anterior pudemos ver quão prático e flexível é o uso do make para compilar partituras do Csound. O maior problema é ter que gerenciar e editar um arquivo para cada seção, o que pode ser irritante em uma composição longa. Nesta seção vamos apresentar algumas soluções que usam a mesma idéia anterior, porém dessa vez usando apenas uma partitura. A geração de arquivos de partitura secundários para cada seção é feita automaticamente.

Utilizando o comando de seções `s`. Provavelmente a solução mais direta é criar um *script* que lê a partitura e cria automaticamente um arquivo para cada seção definida com `s`. Dessa maneira o problema de gerenciar arquivos é resolvido, os arquivos são criados automaticamente e nomeados de acordo com um prefixo dado. (ex. 6.2)

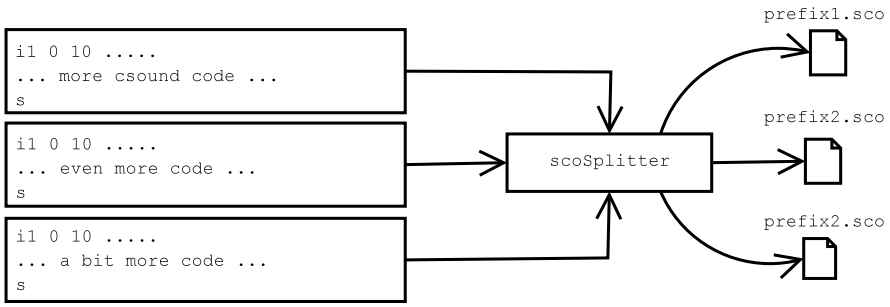

Figura 6.2: Dividindo a partitura

Utilizando um novo comando de seções. A solução anterior é um avanço em relação as outras, mas ter os dados para mixagem separados da música é um problema, já que é necessário manter documentos diferentes. Uma solução mais completa é definir um novo comando `section`, que aceita um nome e o tempo onde a seção se inicia. Naturalmente o Csound não tem esse comando e nós não vamos implementá-lo no núcleo do Csound. Ao invés disso vamos modificar nosso script para extrair as seções lendo esse comando. Como queremos manter a compatibilidade reversa, o comando `section` será precedido do caractere

`;` o caractere de comentário do Csound. Como uma medida extra de segurança, usamos o caractere `|` depois do `;`, apenas para evitar que nosso script reconheça um comentário válido que tenha a palavra “section”. O ex. 6.5 mostra como funciona essa sintaxe. As linhas 1 e 5 definem seções válidas, enquanto na linha 9 temos apenas um comentário comum.

Exemplo 6.5 O comando `section`

```
1 ;|section foo 0
2 i1 0 10 ...
3 ....  more notes here ....
4
5 ;|section bar 10
6 i1 0 10 ...
7 ....  more notes here ....
8
9 ; section, sweet section
10 i1 0 10 ....
11 ......
```

Agora não só os arquivos para cada seção são gerados automaticamente, como também as regras para mixagem. Um bom efeito colateral dessa abordagem é que o usuário não entra os comandos diretamente para um mixador específico. Dessa maneira pode-se mudar o programa de mixagem que está sendo usado sem conhecimento do usuário.

Essa é a melhor entre as quatro soluções. Ela mantem a compatibilidade com a partitura enquanto provê mais poder, flexibilidade e conveniência que as soluções prévias.

6.3 Do conceito de seções para eventos

6.3.1 Introdução

As soluções anteriores dividem a partitura em seções para poder recompilar apenas as partes necessárias. Contudo esse procedimento é ineficiente quando o objetivo é manipular diversos elementos. Na fig. 6.3 os retângulos representam eventos no tempo. O longo retângulo representa um pedal enquanto os outros representam eventos de menor duração. Durante o processo de composição é comum redispor os elementos no tempo até achar o *timing* desejado. Efetuar essas alterações com o Csound é trabalhosa porque o tempo das

durações tem que ser recalculados manualmente. Dividir em seções, como nos exemplos anteriores, não funcionaria, já que o pedal delimita uma seção. A melhor solução é poder definir *eventos.* Dessa maneira uma seção pode conter inúmeros eventos, que definem estruturas *hierárquicas.*

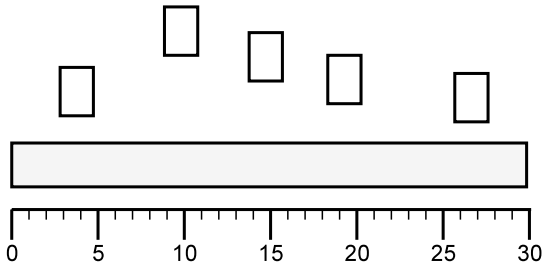

Figura 6.3: Eventos

O ex. 6.6 mostra a sintaxe básica para criar eventos. O evento é definido com `event` seguido do nome do evento. O início de cada evento pode ser definido com a opção `start`. O código do Csound é inserido como parâmetro entre colchetes para o comando `body`. A opção `gain` determina o valor de ganho do evento na mixagem final (fig 6.1).

Exemplo 6.6 Sintaxe para eventos

```
event foo -body {
  i1 0 2 ...
  ...
}
event bar -body {
  i1 0 3 ....
  ...
}
foo config -start 0 -gain .5
bar config -start 30
```

Naturalmente os eventos podem ser aninhados, como visto no ex. 6.7. É possivel representar estruturas hierárquicas e mais complexas que com o Csound, ou mesmo com as soluções anteriores baseadas em seções.

6.3.2 Recursos avançados

Os tempos entre os eventos podem ser implementados como *relações* ao invés de apenas durações determinadas. A implementação foi livremente baseadanas relações propostas

Exemplo 6.7 Eventos aninhados

```
event foobar {
  event foo -body {
  ...
  }
  event bar -body {
  ...
  }
  foo config -start 0
  bar config -start 30
}

foobar config -start 30
```

por Allen (Allen 1991; Allen e Ferguson 1994). Pode-se indicar, por exemplo, que um evento inicia após outro evento, ou que um evento inicia junto com outro. No ex. 6.8 o evento “bar” inicia logo após “foo”, enquanto o evento “chords” inicia ao mesmo tempo que “bar”. A figura 6.4 mostra a representação gráfica do ex. 6.8.

Exemplo 6.8 Relações entre eventos

```
foo config -start 0
bar config -start {after foo}
chords config -start {with bar}
```

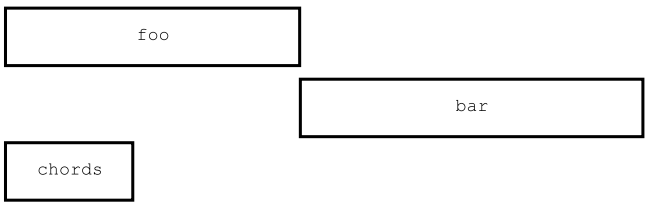

Figura 6.4: Relações entre eventos

Para maior flexibilidade pode-se ter um *padding*, tanto positivo quanto negativo, entre eventos. No ex. 6.9 ambos os eventos “bar” e “chords” iniciam depois de “foo”, contudo “bar” tem um *padding* de 2 segundos enquanto “chords” um *padding* de -2 segundos. Dessa maneira quando “chords” inicia “foo” ainda está sendo tocado, e quando “bar” inicia, “foo” terminou há 2 segundos. (fig. 6.5)

Exemplo 6.9 *Padding* entre eventos

```
bar config -start {after foo} -pad {2s}
chords config -start {after foo} -pad {-2s}
```

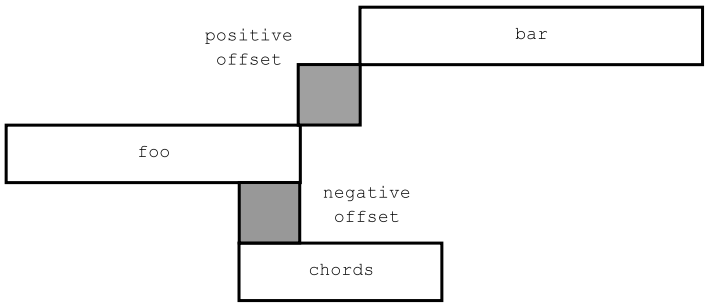

Figura 6.5: *Padding* entre eventos

Outras relações como `before`, `finishes`, e `meets` estão disponíveis e o usuário pode criar suas próprias relações.

CAPÍTULO 7

Exemplos de aplicação: O programa Monochordum

Nos capítulos 3, 4, e 5 apresentamos nossa implementação de uma linguagem de síntese em XML e no capítulo 6 demonstramos o processo de recompilação distribuída e o uso de eventos. Este capítulo mostrará algumas possibilidades de utilização desse tipo de tecnologia.

Naturalmente essas tecnologias fazem mais sentido quando implementadas em conjunto como parte de um sistema para composição. Esse sistema está sendo implementado pelo autor desta tese e encontra-se em fase inicial de desenvolvimento. Por ora implementamos algumas bibliotecas para lidar com aspectos específicos e serão aqui vistas como exemplos de aplicação. Contudo nada impede que partes das tecnologias sejam usadas para fins diversos e específicos, em programas já existentes.

7.1 Bibliotecas

As bibliotecas são implementadas em *[incr TCL]*, uma extensão orientada a objetos do *TCL*. Com o *[incr TCL]* é possível criar código fácil de entender e manter, além do

que ele oferece um conjunto de *mega-widgets* (*[incr Widgets]*), uma estrutura para construir *mega-widgets* (*[incr Tk]*), e suporte para integrar código em *C/C++*.

7.1.1 Xmlparser

A biblioteca Xmlparser define uma classe chamada `xmlparser` para lidar com os instrumentos do csoundXML, o DTD, e a bilioteca CXL. O pacote Tdom (Zerbst 2003) é usado para "parsear" o código em XML.

O método público `readxmlfile` lê o arquivo do disco rígido para a memória e designa uma variável (*channel*, no jargão do *TCL*) a ele. O método `createRoot` "parseia" o arquivo de entrada e retorna o nome DOM designado (automaticamente) à raiz do documento (geralmente similar a `domNode0x811aea0`).

O método `xml2csound` lê o nome de um nó do DOM e converte para o Csound. Ele procura pelos nós com elementos `opcode`, `defpar`, `description`, e `output`. Para cada nó achado os métodos `xmlOpcode`, `xmlDefpar`, `xmlDescription`, e `xmlOutput` são chamados.

O método `xmlOpcode` lê todos os nós dentro de um elemento `opcode` e converte os dados relevantes para Csound. Para cada parâmetro, o atributo `name` é lido, a ordem do parâmetro é "perguntada" à CXL através do método `whatOrder`, verifica-se se o atributo `reserved` é definido (ou seja, se os campos-p p3 ou p4 devem ser usados) utilizando o método `checkReserved`, e finalmente verifica-se se o atributo `auto` é definido com "yes". Caso não seja, o valor está no atributo `vvalue` ou nos nós `<number>` ou `<expr>`.

O método `statistics` extrai informações sobre o instrumento, como número de funções, opcodes, parâmetros, etc. (Um exemplo de uso desse método pode ser visto na figura 7.3). Essa funcionalidade, bastante difícil de ser implementada acessando a linguagem de orquestra diretamente (i.e., usando a orquestra do Csound), é extremamente fácil na versão "xmlificada" (i.e., usando o csoundXML). O método completo pode ser visto no ex. 7.1. A maior parte do trabalho é feita por expressões *Xpath* (Clark e DeRose 1999), como `//par[@name='function']`, que nesse caso procura por *todos* os parâmetros

que tenham o atributo `name` com o valor “function”.

Exemplo 7.1 O método `statistics`

```
public method statistics { root } {
  set functions [ llength [ $root selectNodes {// par[@name='function']}]]
  set opcodes [ llength [ $root selectNodes {// opcode}]]
  set parameters [ llength [ $root selectNodes {// defpar }]]
  set expressions [ llength [ $root selectNodes {// expr}]]
  set export [ llength [ $root selectNodes {// *[@auto='yes']}]]
  set outputs [ llength [ $root selectNodes {// outtype }]]
  return " functions $functions
    opcodes $opcodes
    variables $parameters
    expressions $expressions
    parameters [ expr $export-$functions ]
    outputs \ $outputs"
}
```

O método `readParameters` retorna todos os parâmetros “exportáveis” de um instrumento. Esse é o método principal para construir um editor de parâmetros. Ele procura por parâmetros no elemento `defpar`, depois pelo atributo `auto="yes"` nos opcodes. Ele tem o cuidado de colocar descrição vazia onde não tem, caso contrário o *parser* retornaria um erro.

Outros métodos secundários não serão aqui discutidos já que os comentários no apêndice B.4, p. 118 devem ser suficientes.

7.1.2 Musiclib

A Musiclib implementa uma biblioteca básica para lidar com dados musicais. Até o momento foram criadas classes para lidar com o parâmetro altura. Prevê-se a criação de classes para ritmo e outros parâmetros.

A Musiclib permite a criação de pequenas linguagens para composição musical. Diferentes tipos de codificações como `re#` (*italian*), `ré#` (*portuguese*), `d#` (*cifra*), `dis` (*lily*), `3` (*equal*), e `15` (*tonal*) foram implementadas. O objetivo da Musiclib é fornecer um *framework* básico e comum para a criação de codificações de modo que as codificações possam ser intercaladas e conversíveis entre si.

7.1.2.1 A classe pitch

A classe base Pitch lida com a codificação de notas únicas. As notas são guardadas em duas "tabelas" (*arrays*), uma para definir a codificação das notas tonais (Oliveira 1995, p. 17, tab. 3) e outra para definir a codificação das notas temperadas (Oliveira 1995, p. 17, tab. 4). Cada item do *array* tem o formato `$n(c)$f(3)` ou `$n(c)$s(3)`, onde `$n(c)` indica a nota `c` (nesse caso dó) e `$s(3)` o número de sustenidos (nesse caso 3) ou bemóis (em `$b(3)`). O valor de cada item é dado pelo comando `array set`, que recebe o nome da variável que contém o *array* e um par de valores, sendo o primeiro o índice (que pode ser não-numérico) e o segundo o valor do item. Por exemplo, um comando como:

```
array set n "c dó d ré e mi f fá g sol a lá b si"
```

designará o valor de `dó` para `$n(c)`, `ré` para `$n(d)` e assim por diante. O comando:

```
array set s "1 # 2 ## 3 t# 4 q# 5 p# 6 s# 7 h#"
```

designará o valor de # para `$s(1)`, ## para `$s(2)` e assim sucessivamente. O mesmo se aplica para os bemóis.

O método `setCod` é responsável por atribuir os valores aos *arrays*, de acordo com o tipo de codificação.

O método `changeCod` é responsável por modificar a codificação atual por outra, convertendo a representação.

O método `name2number` converte um valor não-numérico, como `dó`, para seu equivalente numérico, dependendo da codificação, como `0`. O método `number2name` faz exatamente o contrário, converte um valor númerico como `3` para sua representação não-numérica, dependendo da codificação, como `ré#`.

No *[incr TCL]* as variáveis podem executar comandos. A variável `codification` (ex. 7.2) executa os métodos `setCod` e `changeCod` quando tem seu valor modificado, e retorna o valor da nota corrente (i.e., depois da conversão).

Outras variáveis são definidas em `pitch`, os comentários no código fonte devem ser sufucientes para entendê-las (apêndice B.3).

Exemplo 7.2 A variável `codification`

```
public variable codification lily {
  setCod $codification
  changeCod $codification
  return \$currentNote
}
```

7.1.2.2 A classe Set

A classe `Set` (tem que ser maiúscula porque o `TCL` já possui um comando com esse nome) herda todas as características de `pitch` e acrescenta alguns métodos e variáveis para lidar com conjuntos de notas adequadamente.

O método `applyFunction` efetua uma operação linear em cada elemento do conjunto de notas. Ele é usado, por exemplo, para modificar a codificação do conjunto.

Outros métodos como `changeCod` e `setCod` são praticamente iguais àqueles na classe `pitch`, com a diferença que aqui eles foram adaptados para lidar com conjunto de notas.

Enquanto a classe `pitch` tem as variáveis `currentNote` para o valor da nota corrente, `origNote` para o valor original da nota, e `origNoteCode` para o código original da nota; `Set` possui as variáveis equivalentes `currentset`, `origset`, e `origsetCode` que se aplicam a conjuntos ao invés de notas.

7.1.2.3 A classe phrase

A classe `phrase` herda as características da classe `Set` e permite definir “frases”, ou codificar trechos musicais. Ela foi pensada para ser usada com a classe `event`, onde o usuário teria “frases” dentro de “eventos”. `Phrase` expande a classe `Set` permitindo a codificação de articulação e durações rítmicas. No futuro esses parâmetros serão definidos em classes separadas.

O método principal de `phrase` é `parser`. Ele usa expressões regulares para ler e dividir a entrada de dados. Uma nota pode ser codificada como `dó#'4.-.`, onde o `dó#` indica a nota, o `4.` indica a duração (semínima pontuada), e o traço `-` indica articulação. A

expressão regular:

```
([A-Za-záéíóú]*)?([#]*)?(['’,]*)?([0-9]*)?([̇]*)?(-.)?
```

é usada para ler a entrada, onde

sustenido é lido por `([#]*)?`

durações é lido por `([0-9]*)?`

ponto é lido por `([̇]*)?`

sinais de oitava é lido por `(['’,]*)?`

sinais de articulação é lido por `(-.)?`

O método completo pode ser visto no ex. 7.3.

O método `oct2code` converte os sinais da entrada para uma codificação matemática interna.

O método `octcode2lily` converte o código número para o sinal de oitava do Lilypond.

Os métodos `dur2code` e `applydot` são métodos simples para converter a representação rítmica da entrada para valores numéricos. No futuro devem ser substituídos por uma classe robusta e única, que implementará os princípios de *representação com frações* apresentados na seção 2.1.3.7, p. 28.

Os métodos `test(csoundOut)` e `test(lilyOut)` convertem a entrada para código do Csound e Lilypond, respectivamente. Desse modo é possível, com uma única descrição, ter elementos de síntese e notação simultaneamente.

7.1.3 Event

O código fonte completo com comentários de `Event` pode ser visto no apêndice B.2 na página 100.

Exemplo 7.3 O método `parser`

```
public method parser { input } {
  set parserString {([A-Za-záéíóú]*)?([\#]*)?([',]*)?([0-9]*)?([\.]*)?(-.)?}
  set origphrase (note ) {}
  set origphrase ( oct ) {}
  set origphrase ( dur ) {}
  set origphrase ( art ) {}
  set origphraseCode( note ) {}
  set origphraseCode( oct ) {}
  set origphraseCode( dur ) {}
  set origphraseCode( art ) {}
  set lastitem ( oct ) ""
  set lastitem ( dur ) ""

  foreach item $input {
    regexp -nocase -all -- $parserString $item match note sharp oct dur dot art

    append tmp $note $sharp $oct $dur $dot $art

    if {[ string compare $tmp $item] ≠0} {
      puts "erro no parser"
    } else {
      lappend origphrase ( note ) $note$sharp
      lappend origphrase ( oct ) $oct
      lappend origphrase ( dur ) $dur$dot
      lappend origphrase ( art ) $art

      if {$oct == ""} {
        set oct $lastitem ( oct )
      }
      if {$dur == ""} {
        set dur $lastitem ( dur )
      }

      lappend origphraseCode( note ) [ name2number $note$sharp]
      lappend origphraseCode( oct ) [ oct2code $oct ]
      lappend origphraseCode( dur ) $dur$dot
      lappend origphraseCode( art ) [ art2code $art ]

      set lastitem ( oct ) $oct
      set lastitem ( dur ) \ $dur
    }
    set tmp ""
  }
}
```

A biblioteca `events` define uma classe chamada `event` para a criação de objetos de evento. O comando `event nome` cria um objeto chamado “nome” da classe `event`. Cada objeto pode ser manipulado (configurado) usando seu nome e o comando `config`.

No *[incr TCL]* opções de configuração como `start` e `body` começam com um traço

(`-`) e na verdade se referem a variáveis definidas na classe. A classe Event define as variáveis públicas `pad`, `mi`, `start`, `dur`, `body`, e `gain`, que são acessíveis aos usuários através do esquema de configuração citado.

Na verdade `event` tem três variáveis para armazenar o valor do início do evento, `start`, que contém o valor inicial como entrado pelo usuário e nunca é modificado; `fstart`, que é o resultado de `rstart` modificado por operações como `after` e `with`; e `rstart` que é o resultado de `fstart` somado a valores de *padding* (i.e. `pad`). Ambos (`rstart` e `fstart`) têm seus valores modificados após cada operação, enquanto `start` nunca tem seu valor modificado. Contudo, a implementação procura ser a mais transparente possível e espera que o usuário use `config get -start` e não `config get -rstart` ou `config get -fstart`, o que seria muito confuso. Quando o usuário pede o valor de `start` ele obtém, na verdade, o valor de `rstart`. Idealmente as variáveis `rstart` e `fstart` deveriam ser privadas, mas como elas precisam ser acessadas por outros objetos, elas são públicas.

As relações entre eventos são definidas como *métodos* públicos: `after`, `with`, `before`, `finishes`, e `middle`. Eles aceitam um parâmetro de entrada que é o objeto (evento) a que eles se referem. O *[incr TCL]* permite uma escrita muito limpa e a implementação de cada um dos métodos não tem mais do que oito linhas. A implementação completa pode ser vista no apêndice B.2.

Como foi visto anteriormente, o método `config` é responsável por efetuar a configuração das variáveis. Contudo esse método é apenas um *container* para outros métodos, como pode ser visto no ex. 7.4.

Exemplo 7.4 O método `config`

```
public method config {args} {
  evalDurBeforePad $args
  eval configure $args
  evalStart
  applyPad
}
```

O método `evalDurBfrPad` garante que a variável `dur` será calculada antes de `pad`,

e ambas antes das outras opções. De outro modo problemas podem acontecer com métodos que precisam desses valores, como `before`. O comando `eval configure $args` é a maneira tradicional de calcular os valores de entrada no [incr TCL]. O método `evalStart` verifica se a entrada contém os métodos `after`, `with`, `before`, `finishes`, `middle` ou apenas valores numericos. No primeiro caso os métodos correspondentes são chamados e a operação é efetuada. No segundo caso o valor de `start` é atribuído à `fstart`. Finalmente, o método `applyPad` soma o valor atual do início do evento ao valor de `pad` em `rstart`.

O método `get` permite que o usuário obtenha o valor de uma variável pública. O *[incr TCL]* tem o comando `cget` para esse mesmo fim, mas como é preciso retornar o valor de `rstart` quando o usuário pede `start`, um outro método precisou ser criado.

7.2 Editor de parâmetros

Um editor de parâmetros básico foi implementado na classe `gui` (apêndice B.1).

Após o instrumento ter sido validado contra o DTD, dados são extraídos do instrumento, como os parâmetros que deverão ser mostrados pelo editor. A classe `gui` procura por elementos que tenham o atributo `auto="yes"` gera *sliders* para cada parâmetro, além de caixas de opção para as funções. Como as funções ficam armazenadas em um arquivo à parte, o programa lê as funções nesse arquivo e deixa todas disponíveis na caixa de opções. A figura 7.1 mostra uma visão geral da criação da unidade gráfica.

A maior vantagem dessa abordagem é que a GUI é criada de um código como o visto no ex. 7.5, ou seja, nenhuma informação específica sobre a GUI precisa ser codificada no instrumento. A GUI é gerada automaticamente.

Depois que o editor de parâmetros é criado, ele mantém uma ligação entre o instrumento em csoundXML e o CXL para poder gerar o código do csound necessário para a renderização, como visto na fig. 7.2.

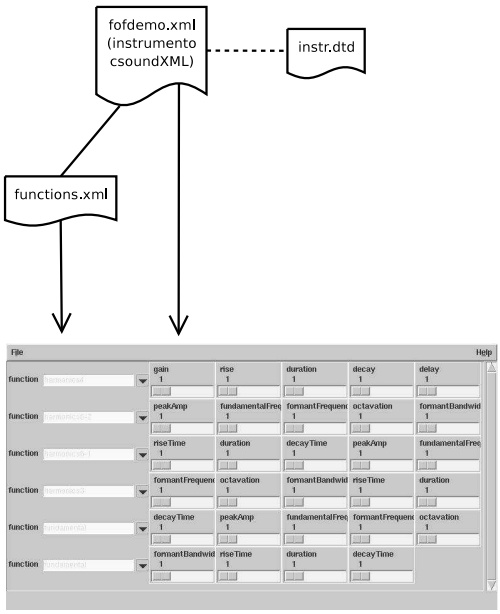

Figura 7.1: Editor de parâmetros—criação da GUI

Exemplo 7.5 Trecho do instrumento fofdemo.xml

```
<opcode name="oscil1i" id="vibctl " type="k">
    <par name="delay" auto="yes"/>
    <par name="amplitude">
      <number>1</number>
    </par>
    <par name="duration">
      <number>.25</number>
    </par>
    <par name="function" auto="yes"/>
</opcode>
```

7.3 Estatísticas

Dados estatísticos sobre o instrumento podem ser facilmente obtidos usando *Xpath queries*. O método `drawStatistics` desenha uma caixa de listas e nela insere os dados sobre o instrumento. Ele utiliza o método `statistics` da classe xmlparser para extrair as informações do instrumento. Observe (ex. 7.6) que a parte do código que lida com o código XML tem apenas 3 linhas! Todo o resto lida com o desenho do *widget*. Um exemplo de como dados estatísticos podem ser incorporados a um EP pode ser visto na fig. 7.3. Esse tipo de dado pode ser muito útil para criar depuradores para instrumentos, para saber quais opcodes são mais usados, para controlar funções, etc.

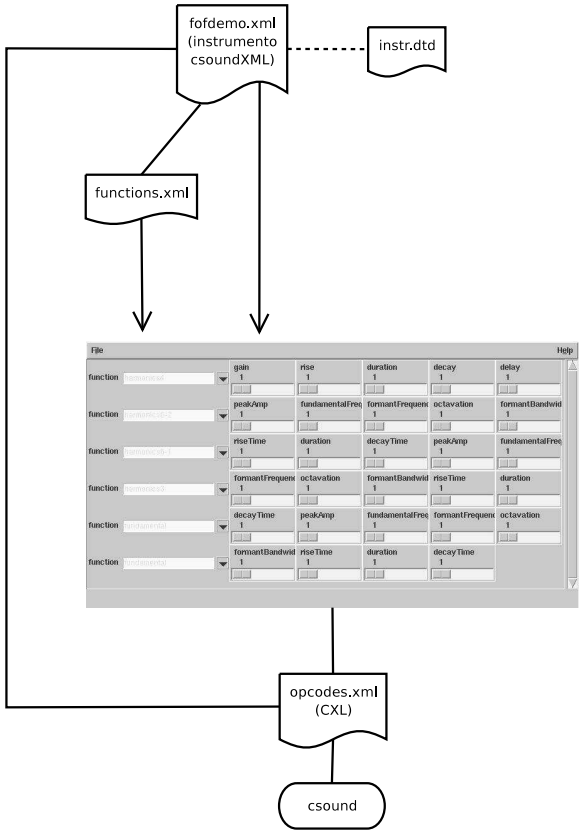

Figura 7.2: Editor de parâmetros

Exemplo 7.6 O método `drawStatistics`

```
public method drawStatistics {rootwin} {
    iwidgets::scrolledlistbox $rootwin.slb \
     -hscrollmode dynamic \
     -vscrollmode dynamic \
     -scrollmargin 5 \
     -labelpos n \
     -selectforeground white \
     -labeltext Statistics:

  pack $rootwin.slb

  set file [ readxmlfile $cfile ]
  set root [ createRoot $file ]
  set list [ statistics $root ]

  foreach {nome value} $list {
    $rootwin.slb insert end "\$nome: $value"
  }
```

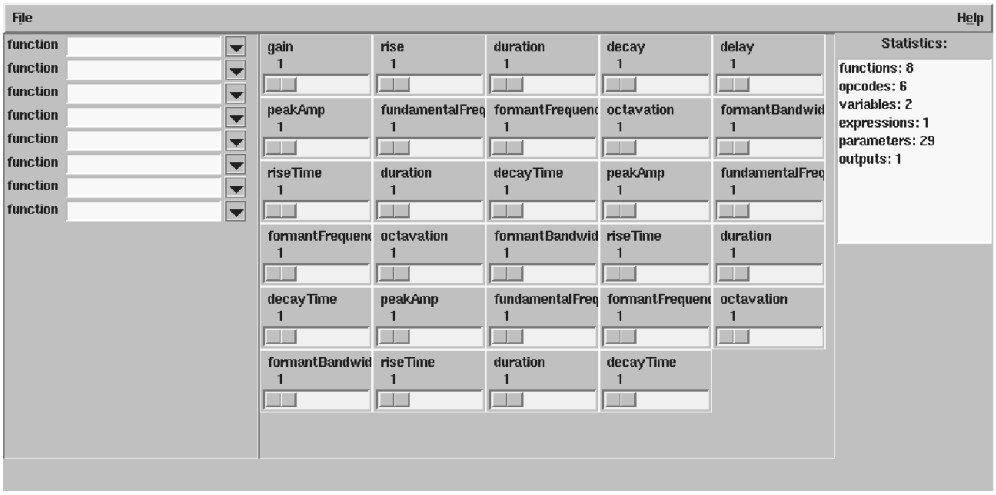

Figura 7.3: Estatísticas incorporadas no editor de parâmetros

CAPÍTULO 8

Conclusões

Conforme vimos, a criação de uma meta-linguagem para síntese sonora resolve alguns dos problemas levantados na seção 1.1.

A reutilização de instrumentos é facilitada por uma descrição de alto-nível, instrumentos nomeados, entrada e saída de sinais flexíveis, e sobretudo a possibilidade de poder definir múltiplas saídas dependendo do contexto.

O uso de uma sintaxe estruturada como a do XML permite sair das limitações impostas pelas listas estruturadas do Music N. Conforme demonstrado no capítulo 7, é possível extrair informações de instrumentos com facilidade.

Diferentemente de outras soluções que inserem comandos gráficos diretamente nos instrumentos, a estruturação da meta-linguagem em XML permite que instrumentos gráficos sejam criados automaticamente, sem a necessidade de opcodes extras (seção 7.2).

O capítulo 6.3 demonstra que o processo de dividir a partitura em arquivos menores e utilizar um utilitário de recompilação como o *make* pode dar mais flexibilidade e velocidade ao processo de composição com o Csound. Contudo faz-se necessário criar algum mecanismo para dividir a partitura e criar os arquivos secundários automaticamente. A solução

final é poder definir eventos, uma estrutura entre a lista de notas e a seção. Além de resolver elegantemente o problema anterior, a definição de eventos possibilita criar partituras com estrutura hierárquica. Além de poder definir blocos de eventos, pode-se também determinar relações de tempo entre esses eventos.

Finalmente, o problema da falta de integração entre as soluções para a partitura (como pré-processadores) e a orquestra é resolvido com a descrição de ambas em um nível mais alto e a utilização de automação de parâmetros e contexto. A solução proposta permite a criação de um sistema integrado, semelhante ao proposto por Deyer (1984), que pode ser acessado com diversas interfaces.

8.1 Contribuições

Meta-linguagem para síntese sonora As unidades geradoras de síntese são descritas em uma linguagem neutra, permitindo uma descrição gráfica automática, a conversão para outras linguagens—como Csound, cmix, etc—, a criação de banco de dados de instrumentos, e a criação de depuradores mais robustos.

Meta-descrição dos blocos componentes de síntese sonora Além da descrição das unidades geradoras, pode-se descrever e conectar em um nível mais alto blocos ou instrumentos de síntese.

Biblioteca com descrição em alto-nível dos opcodes e parâmetros do Csound Essa biblioteca descreve os opcodes e parâmetros, não apenas dizendo como eles devem ser parseados (como a sintaxe BNF), mas descrevendo como cada opcode se comporta e que tipo de entrada é esperada. Por exemplo, se a entrada é em graus ou em decibéis.

Descrição hierárquica e modular de eventos Os eventos—notas, acordes, seções inteiras, arquivos midi—ao contrário do Music N, podem ser agrupados em unidades que por sua vez podem ser transformados, reagrupados, etc.

Criação de uma "pequena linguagem" para definir os eventos Essa pequena linguagem permite a criação simultânea de notação gráfica e código do Csound. Cabe salientar que o código descrito nessa linguagem pode ser convertido para outras linguagens como Lilypond, Csound, Finale, etc.

Renderização distribuída e modular Tradicionalmente as partituras do Csound são compiladas como um único bloco. Nós apresentamos uma proposta onde cada evento é compilado como um arquivo separado do Csound. Assim, não só eventos diferentes podem ser compilados ao mesmo tempo, como uma relação de dependência entre os eventos é criada, onde apenas os eventos modificados precisam ser recompilados.

8.2 Considerações finais

As soluções apresentadas neste trabalho podem ser aplicadas em diferentes contextos. Elas podem

- ser implementadas em ferramentas para expandir programas de síntese sonora já existentes (como o Csound), funcionando como *front-ends*. Ferramentas como editores de parâmetros, editores de instrumento, e editores de funções podem ser implementadas com relativa facilidade
- constituir a base para um sistema composicional completo
- ser incorporadas (no todo ou em parte) em programas de síntese já existentes
- servir de base para a criação de novas linguagens para síntese
- ser transformadas para utilizar outra linguagem de síntese como base

Acreditamos que sistemas para síntese sonora podem ser beneficiados pelo uso das soluções apresentadas neste trabalho.

APÊNDICE A

Convenções usadas neste documento

No corpo do texto, ao menos que seja indicado o contrário, os termos *orquestra* e *partitura* se referem aos termos usados pelo Music N.

Todas as traduções são nossas, a menos que outrem seja indicado.

Esta é uma lista das convenções tipográficas usadas neste documento. Na primeira coluna estão os tipos de elementos enquanto na segunda coluna estão exemplos com os caracteres correspondentes.

Tipos	exemplos
Nomes de programas	Csound
Linguagens de programação	*pascal*
Siglas	LADSPA
Funções e elementos do Csound	`pvoc`
Funções de programação	`procedure`
Conceitos	Mega-instrumento
Estrangeirismos não registrados em dicionário	“renderizar”
Palavras estrangeiras em geral	*Tool Command Language*

APÊNDICE B

Listagem de programas

B.1 Monochordum

B.1.1 Introdução

O programa Monochordum é composto na verdade de diversas bibliotecas. Algumas delas tem seu código listados nas seções B.2, B.3, e B.4. Essa seção mostra o código da classe `gui`, que implementa um simples editor de parâmetros.

Os arquivos `xmlparser.itcl` e `balloon.tcl` são acrescentados ao código.

```
source libs/xmlparser/ xmlparser.itcl
source libs/balloon/ balloon.tcl
```

B.1.2 Classe gui

A classe é criada dentro do *namespace* `gui` e herda código da classe xmlparser.

```
itcl::class monochordum::gui {
  inherit xmlparser
```

Variável `widgets` Armazena os valores dos *widgets* gráficos.

```
public variable widgets
```

Variável `toplevel` Define o valor da janela de nível mais alto.

```
public variable toplevel ".."
```

Variável `env` Armazena valores de variáveis de sistema.

```
public variable env
```

Variável `cfile` Cópia do arquivo de entrada.

```
private variable cfile
```

Variável `funcList` Armazena o nome das funções disponíveis.

```
private variable funcList ‘‘’’
```

Construtor Lê um arquivo opcional de instrumento do csoundXML de entrada, lê as funções no arquivo `functions.xml`, e desenha a tela.

```
constructor {{ file ‘‘’’}} {
  set env(imageDir) images
  wm title $toplevel [ ::msgcat::mc ‘‘ Monochordum’’]
  option add ∗background LightSlateBlue
  . configure −background LightSlateBlue
  wm protocol . WM_DELETE_WINDOW {
    exit
  }

  parseFunctions data / functions.xml
  drawall

  set cfile $file
  if { $cfile ≠ ‘‘’’} {
    parseInstr $cfile
    drawInstr ‘‘’’
    drawStatistics ‘‘’’
  }
}
```

Método `drawall` *Container* para outros métodos que desenham a tela principal; o menu e a barra de estado.

```
public method drawall {} {
  mainMenu .mainmenu
  statusBar .statusBar
  drawMenu
  }
```

Método `drawMenu` Cria os menus usados pelo programa. Os métodos `addMenu`, `addSubMenu`, e `addSeparator`, são usados para esse fim.

```
public method drawMenu {} {
  addMenu .mmenu File 1
  addSubMenu .mmenu.openinstr ‘‘Open Instrument’’ 1 { Open instrument } \
    [code $this selectInstr ]

  addMenu help Help 1
  addSubMenu .help.about { About... } 1 { Some informations about this program}
}
```

Método `parseInstr` Lê o arquivo de instrumento e prepara-o para processamento. Os valores dos parâmetros são gardados na variável `par`, definida na classe xmlparser.

```
public method parseInstr { input } {
  set file [ readxmlfile $input ]
  set root [ createRoot $file ]
  readParameters $root
}
```

Método `parseFunctions` Lê o arquivo de funções e prepara-o para processamento.

```
public method parseFunctions { input } {
  set file [ readxmlfile $input ]
  set root [ createRoot $file ]
  set funcList [ readFuncNames $root]
}
```

Método `updateList` Atualiza a lista de funções com a informação contida no arquivo `functions.xml`

```
public method updateList {rootwin} {
  $rootwin delete list 0 end
  $rootwin delete entry 0 end

  eval $rootwin insert list end $funcList

  return
}
```

Método `drawInstr` Lê os valores da variável `pars` definidos por `parseInstr`. Se o valor for uma função, cria uma lista de funções, caso contrário cria um *scale*. Quando o instrumento tem muitos parâmentros, o método procura ajustá-los adequadamente na tela.

```
public method drawInstr {rootwin} {
   iwidgets::scrolledframe $rootwin.fr1 \
     -width 400 \
     -height 1050 \
     -vscrollmode dynamic \
     -hscrollmode dynamic \
     -borderwidth 1
   iwidgets::scrolledframe $rootwin.fr2 \
     -vscrollmode dynamic \
     -hscrollmode dynamic \
     -borderwidth 1
  set frame [ $rootwin.fr1 childsite ]
  set frame2 [ $rootwin.fr2 childsite ]
  set rown 1
  set coln 1
  set pass 1
  set rown2 1
  set totalColumns 5
```

```
    for { set x 0} {$x < $pars( size )} { incr x} {
20    if {$pars($x,name) == '' function ''} {
        set w($x) [iwidgets::combobox $frame2.instr$x \
22                  -labeltext '' $pars($x,name )'' \
                    -editable false \
24                  -arrowrelief ridge \
                    -popupcursor hand2 \
26                  -selectioncommand [code $this updateList $frame.instr$x ]]
        grid config $w($x) -column 0 -row $rown
28      incr rown
      } else {
30      set w($x) [ scale $frame.instr$x -from 1 -to 10 \
                    -label '' $pars($x,name )'' -orient horizontal ]
32
        if {$coln > $totalColumns} {
34        incr rown2
          set coln 1
36        incr pass
        }
38
        grid config $w($x) -column $coln -row $rown2
40      incr coln
      }
42
      if {$pars( $x,descr ) == ''''} {
44      set_balloon $w($x) '' no description, maybe you should \n \
        add one to this instrument ''
46    } else {
        set_balloon $w($x) '' $pars( $x,descr )''
48    }
    }
50
    button $frame.but1 -text test
52  button $frame.but2 -text reread -command [code $this parseInstr $cfile ]

54  pack $rootwin.fr2 -side left -expand 1 -fill both
    pack $rootwin.fr1 -side left -expand 1 -fill both
56 }
```

Método `drawStatistics` Utiliza o método `statistics` da classe xmlparser para desenhar uma caixa com informações sobre o instrumento.

```
  public method drawStatistics {rootwin} {
2   iwidgets::scrolledlistbox $rootwin.slb \
      -hscrollmode dynamic \
4     -vscrollmode dynamic \
      -scrollmargin 5 \
6     -labelpos n \
      -selectforeground white \
8     -labeltext Statistics:
```

```
    pack $rootwin.slb

    set file [ readxmlfile $cfile ]
    set root [ createRoot $file ]
    set list [ statistics $root ]

    foreach {nome value} $list {
      $rootwin.slb insert end ``$nome: $value''
    }
  }
```

Método Cria uma caixa para seleção de instrumentos.

`selectInstr`

```
  private method selectInstr {} {
    iwidgets::fileselectiondialog .fsd \
    -mask ``*.xml'' \
    -directory ``./instruments'' \
    -fileslabel ``Instrumentos'' \
    -dirslabel ``Diretório de Instrumentos'' \
    -modality application

    wm title .fsd ``Seleciona instrumento''

    if [ .fsd activate ] {
      return [ .fsd get]
    }

  }
```

Método Cria a barra de estado.

`statusBar`

```
  public method statusBar {rootwin} {
    set widgets( statusbar ) [ label $rootwin]

    pack configure $widgets( statusbar ) \
      -fill x \
      -side bottom \
      -padx 5 \
      -pady 5
  }
```

Método Cria o menu principal.

`mainMenu`

```
  public method mainMenu {rootwin} {
    set widgets(menu) [iwidgets::menubar $rootwin]
    pack configure $widgets(menu) -fill x -side top

  }
```

Método `addMenu` Método usado para acrescentar menus. Ele permite uma maneira mais clara para criar menus e submenus.

```
private method addMenu {name label underline} {
    set space ‘‘ ’’
    $widgets(menu) add menubutton $name \
        −text ‘‘[ ::msgcat::mc "$label" ]’’ \
        −underline $underline
}
```

Método `addCascade` Método usado para acrescentar menus dentro de menus. Ele permite uma maneira mais clara para criar menus e submenus.

```
private method addCascade {name label underline } {
    $widgets(menu) add cascade $name \
        −label ‘‘[ ::msgcat::mc "$label" ]’’ \
        −underline $underline \
    }
```

Método `addSubMenu` Método usado para acrescentar menus. Ele permite uma maneira mais clara para criar menus e submenus. Esse método *deve* ser seguido dos métodos `addMenu` ou `addCascade`.

```
private method addSubMenu {name label underline helpstr {command ‘‘’’}} {
    $widgets(menu) add command $name \
        −label $label \
        −underline $underline \
        −helpstr ‘‘[ ::msgcat::mc "$helpstr" ]’’ \
      −command $command
}
```

Método `addSeparator` Cria uma linha separadora entre menus.

```
private method addSeparator {name} {
    $widgets(menu) add separator $name
}

}
```

B.2 Eventos

B.2.1 Introdução

O pacote `events` contém uma única classe, Event. Mais detalhes sobre a implementação podem ser vistos na seção 7.1.3.

B.2.2 Classe event

Classe para lidar com eventos, definida dentro do *namespace* `events`.

```
::itcl::class events::event {
```

Variável `pad` Armazena o valor numérico do *padding*.

```
public variable pad 0
```

Variável `mi` Define o mega-instrumento que o evento se relaciona.

```
public variable mi ‘‘’’
```

Variável `type` Define o tipo de código que será exportado, como csound e lilypond.

```
public variable type ‘‘ csound’’
```

Variável `rstart` Armazena o resultado de `fstart` somado a valores de *padding* (i.e. `pad`). Tem que ser pública porque algumas operações precisam acessar dados de outros objetos.

```
public variable rstart ‘‘’’
```

Variável `fstart` Armazena o resultado de `rstart` modificado por operações como `after` e `with`. Tem que ser pública porque algumas operações precisam acessar dados de outros objetos.

```
public variable fstart ‘‘’’
```

Variável `start` Armazena o valor inicial de início. Nunca é modificada.

```
public variable start ‘‘’’
```

Variável `dur` Existem três maneiras de determinar o valor da duração de um evento; especificando com `-dur`, parseando o código (csound) e determinando o valor, e gerando o arquivo de saida (wav) do evento e lendo seu valor. Infelizmente apenas o último é o mais preciso. Por ora o valor total tem que ser informado.

```
public variable dur ‘‘’’
```

Variável `body` Armazena o código da pequena linguagem ou código “cru” do Csound.

```
public variable body ‘‘’’
```

Variável `gain` Armazena o valor de ganho do evento.

```
public variable gain ''1''
```

Variável `tmpdir` Indica o diretório para gerar arquivos temporários. No UNIX é `/tmp` por padrão.

```
protected variable tmpdir ''/ tmp''
```

Construtor Apenas executa o código de entrada. Incluir verificação para erro no futuro.

```
constructor { args } {
  eval config $args
}
```

Método `evalStart` Verifica se `start` é um valor numérico simples ou se tem um código válido que precisa ser computado (como `before object`).

```
private method evalStart {} {
  set regval {( after |with| before | finishes |middle )[ \ t]+[a-zA-Z0-9\_\.\-]*}
  if {[ regexp $regval $start ]} {
    eval $start
  } else {
    set fstart $start
  }
}
```

Método `evalDurBfrPad` Faz com que `dur` seja computado antes de `pad`, e ambos antes das outras opções. De outro modo problemas podem acontecer com métodos que precisam desses valores, como `before`.

```
public method evalDurBfrPad {input} {
  set durpos [ lsearch $input -dur]
  set padpos [ lsearch $input -pad]

  if {$durpos ≠ ''-1''} {
    set durvalue [ lindex $input [ expr $durpos + 1]]
    set tmp [regexp -nocase -all -line -- {^[0-9.-]*$} $durvalue]
    if {$tmp == 1} {
      if {$durvalue ≥ 0} {
        set dur $durvalue
      } else {
        error {should be integer}
      }
    } else {
      error {should be a number}
    }
  }

  if {$padpos ≠ ''-1''} {
    set pad [ lindex $input [ expr $padpos + 1]]
  }
}
```

Método `export` Método básico para exportar código a partir dos formatos “mono”, “csound”, e “lilypond”.

```
public method export { outdir } {
  switch -- $type {
    mono {exporter mono $outdir}
    csound { exporter sco $outdir }
    lily { exporter ly $outdir }
  }
}
```

Método `exporter` Código preliminar para exportar dados. se old.ext existe faz um diff entre new.ext e old.ext, caso old.ext não exista cria o arquivo novo. O `ext` é a extensão do arquivo (e.g. sco).

```
public method exporter { ext outdir } {
  set text [ $this cget -body]
  set filename [ lindex [ split $this :: ] end]
  set exportedfile [ file join $outdir $filename.$ext ]
  if {[ file exists $exportedfile ]} {
    set newsco [ file join $tmpdir $filename.$ext-new]
    set tmp [open $newsco w]
    puts $tmp $text
    close $tmp
    catch {exec diff -B -w -E -q $newsco $exportedfile} differ
    file delete $newsco
    if { $differ ≠ ‘’’} {
      set sco [open $exportedfile w]
      puts $sco $text
      close $sco
    }
  } else {
    set sco [open $exportedfile w]
    puts $sco $text
    close $sco
  }
}
```

Método `after` Calcula o valor inicial em relação a outro evento.

```
public method after {event} {
  set evst [$event cget -rstart]
  set evdur [$event cget -dur]
  $this configure -fstart [expr $evst + $evdur]
}
```

Método `with` Determina que o valor inicial seja igual a de outro evento.

```
public method with {event} {
  set st [$event cget -rstart]
  $this configure -fstart $st
}
```

Método `before` Faz com que esse evento comece antes do evento indicado.

```
public method before {event} {
  set edur [$this cget -dur]
  set st [$event cget -rstart]
  set value [expr $st - $edur]
  if {$value >= 0} {
    $this configure -fstart $value
  } else {
    error "should be integer"
  }
}
```

Método `finishes` Faz com que ambos eventos terminem ao mesmo tempo. Aparentemente se colocar o $this (edur) depois do $event dá problema

```
public method finishes {event} {
  set edur [$this cget -dur]
  set st [$event cget -rstart]
  set evdur [$event cget -dur]
  $this configure -fstart [expr $st + $evdur - $edur]
}
```

Método `middle` "Justifica" dois eventos, a partir do meio.

```
public method middle {event} {
  set edur [$this cget -dur]
  set st [$event cget -rstart]
  set evdur [$event cget -dur]
  set addst [expr ($evdur - $edur)/2.0]
  $this configure -fstart [expr $addst + $st]
}
```

Método `config` Permite modificação de valores de eventos.

```
public method config {args} {
  evalDurBfrPad $args
  eval configure $args
  evalStart
  applyPad
}
```

Método `applyPad` Soma o valor de `pad` à duração total.

```
private method applyPad {} {
  set st [$this cget -fstart]
  set pd [$this cget -pad]
  if {$st != ""} {
    $this configure -rstart [expr $st + $pd]
  }
}
```

Método `get` Permite a extração de valores de variáveis. Quando o usuário pede `get -start` ele obtem o valor de `-rstart` na verdade.

```
public method get {input} {
  switch -- $input {
    -start {
      evalStart
      applyPad
      return [ $this cget -rstart ]
    }
    -dur {return [ $this cget -dur]}
    -type {return [ $this cget -type]}
    -mi {return [$this cget -mi]}
    -body {return [$this cget -body]}
    -gain {return [ $this cget -gain]}
    -pad {return [$this cget -pad]}
  }
}
```

B.3 Musiclib

B.3.1 Introdução

A Musiclib implementa uma biblioteca básica para lidar com dados musicais. Até o momento foram criadas classes para lidar com o parâmetro altura. Prevê-se a criação de classes para ritmo e outros parâmetros.

O pacote Musiclib contém três classes; Pitch define codificações e operações com uma única nota, Set define codificações e operações com um conjunto de notas, e Set define codificação em um nível mais alto, com indicação de duração e articulação. A relação de herança entre as classes pode ser vista na figura B.1.

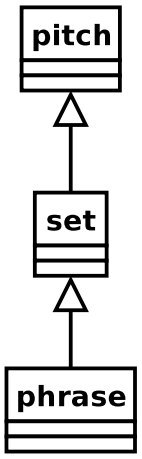

Figura B.1: Herança no pacote Musiclib

B.3.2 Classe pitch

Define codificação e operações em uma única nota.

```
itcl::class musiclib::pitch {
2  inherit errorlib::errorlib
```

Variável Sustenido.
`s` **public common** s

Variável Bemol (*flat*).
`f` **public common** f

Variável Intervalo.
`i` **public common** i

Variável Qualidade do intervalo.
`q` **public common** q

Variável `n` Nota.

```
public common n
```

Variável `centralC` Valor do dó central,

```
public common centralC 3
```

Variável `baseLa` Valor em hertz do lá base.

```
public common baseLa 440
```

Variável `csoundCentralC` Valos do dó central no csound.

```
public common csoundCentralC 8
```

Variável `codification` Define a codificação da nota. O padrão é “lily”.

```
public variable codification lily {
  setCod $codification
  changeCod $codification
  return $currentNote
}
```

Variável `direction` Considera a direção do intervalo.

```
public variable direction {} {
  switch $direction {
    true {}
    false {}
  }
}
```

Variável `octaveEquiv` Equivalencia de oitava ativada ou não.

```
public variable octaveEquiv {} {
  switch $octaveEquiv {
    true {}
    false {}
  }
}
```

Variável `system` Define se o sistema é tonal (mod96) ou igual (mod12).

```
public variable system {} {
  switch $system {
    equal {
      set sysMod 12
      set notelist $notesEqual
```

```
6         }
          tonal {
8           set sysMod 96
            set notelist $notesTonal
10        }
        }
12    }
```

Variável `origNote`
Guarda o valor orignal da nota.

public variable origNote

Variável `origNoteCode`
Guarda o código original da nota.

public variable origNoteCode ‘‘’’

Variável `currentNote`
Valor da nota corrente.

public variable currentNote

Variável `sysMod`
Valor do módulo de sistema. O padrão é “96”.

protected common sysMod 96

Variável `notelist`
Guarda um *array* de notas dependendo do sistema.

protected common notelist

Variável `nonNumCodes`
Lista com o nome de codificações não numéricos.

protected common nonNumCodes {italian cifra lily german portuguese}

Variável `numericCodes`
Lista com o nome de códificações numéricas.

protected common numericCodes {hertz num12 num96}

Variável `notesTonal`
Lista o nome das notas mantendo a diferença entre sustenidos e bemois. Cada nota recebe um número único de 0 a 95. Aqui esses números combinam com o índice da lista, i.e., a primeira nota tem 0 como número, a segunda 1, e assim por diante. Essa codificação foi criada pelo Prof. Jamary Oliveira (Oliveira 1995; Oliveira 2001a).

```
public common notesTonal {
2   $n(c) $n(c)$s(1) $n(c)$s(2) $n(c)$s(3) $n(c)$s(4) $n(c)$s(5) $n(c)$s(6)
    $n(d)$f(7) $n(d)$f(6) $n(d)$f(5) $n(d)$f(4) $n(d)$f(3) $n(d)$f(2) $n(d)$f(1)
4   $n(d) $n(d)$s(1) $n(d)$s(2) $n(d)$s(3) $n(d)$s(4) $n(d)$s(5) $n(d)$s(6)
    $n(e)$f(7) $n(e)$f(6) $n(e)$f(5) $n(e)$f(4) $n(e)$f(3) $n(e)$f(2) $n(e)$f(1)
6   $n(e) $n(e)$s(1) $n(e)$s(2) $n(e)$s(3) $n(e)$s(4) $n(e)$s(5) $n(e)$s(6)
    $n(f)$f(6) $n(f)$f(5) $n(f)$f(4) $n(f)$f(3) $n(f)$f(2) $n(f)$f(1)
8   $n(f) $n(f)$s(1) $n(f)$s(2) $n(f)$s(3) $n(f)$s(4) $n(f)$s(5) $n(f)$s(6) $n(f)$s(7)
```

```
    $n(g)$f(6) $n(g)$f(5) $n(g)$f(4) $n(g)$f(3) $n(g)$f(2) $n(g)$f(1)
    $n(g) $n(g)$s(1) $n(g)$s(2) $n(g)$s(3) $n(g)$s(4) $n(g)$s(5) $n(g)$s(6)
    $n(a)$f(7) $n(a)$f(6) $n(a)$f(5) $n(a)$f(4) $n(a)$f(3) $n(a)$f(2) $n(a)$f(1)
    $n(a) $n(a)$s(1) $n(a)$s(2) $n(a)$s(3) $n(a)$s(4) $n(a)$s(5) $n(a)$s(6)
    $n(b)$f(7) $n(b)$f(6) $n(b)$f(5) $n(b)$f(4) $n(b)$f(3) $n(b)$f(2) $n(b)$f(1)
    $n(b) $n(b)$s(1) $n(b)$s(2) $n(b)$s(3) $n(b)$s(4) $n(b)$s(5) $n(b)$s(6)
    $n(c)$f(6) $n(c)$f(5) $n(c)$f(4) $n(c)$f(3) $n(c)$f(2) $n(c)$f(1)
}
```

Variável As 12 notas da escala cromática temperada.

`notesEqual`

```
public common notesEqual {
    $n(c) $n(c)$s(1) $n(d) $n(d)$s(1) $n(e) $n(f) $n(f)$s(1)
    $n(g) $n(g)$s(1) $n(a) $n(a)$s(1) $n(b)
}
```

Variável Apenas as notas das teclas brancas do teclado.

`notesWhiteKeys`

```
public common notesWhiteKeys {
    $n(c) $n(d) $n(e) $n(f) $n(g) $n(a) $n(b)
}
```

Variável Apenas as notas das teclas pretas do teclado escritas apenas com sustenidos.

`notesBlackKeysS`

```
public common notesBlackKeysS {
    $n(c)$s(1) $n(d)$s(1) $n(f)$s(1) $n(g)$s(1) $n(a)$s(1)
}
```

Variável Apenas as notas das teclas pretas do teclado escritas apenas com bemois.

`notesBlackKeysF`

```
public common notesBlackKeysF {
    $n(d)$f(1) $n(e)$f(1) $n(g)$f(1) $n(a)$f(1) $n(b)$f(1)
}
```

Variável Intervalos tonais, em módulo 96, descrito em (Oliveira 1995; Oliveira 2001a).

`intervalsTonal`

```
public common intervalsTonal {
    $i(1)$q(J) $i(1)$q(1+) $i(1)$q(2+) $i(1)$q(3+) $i(1)$q(4+) $i(1)$q(5+) $i(1)$q(6+)
    $i(2)$q(6d) $i(2)$q(5d) $i(2)$q(4d) $i(2)$q(3d) $i(2)$q(2d) $i(2)$q(d) $i(2)$q(m)
    $i(2)$q(M) $i(2)$q(1+) $i(2)$q(2+) $i(2)$q(3+) $i(2)$q(4+) $i(2)$q(5+) $i(2)$q(6+)
    $i(3)$q(6d) $i(3)$q(5d) $i(3)$q(4d) $i(3)$q(3d) $i(3)$q(2d) $i(3)$q(d) $i(3)$q(m)
    $i(3)$q(M) $i(3)$q(1+) $i(3)$q(2+) $i(3)$q(3+) $i(3)$q(4+) $i(3)$q(5+) $i(3)$q(6+)
    $i(4)$q(6d) $i(4)$q(5d) $i(4)$q(4d) $i(4)$q(3d) $i(4)$q(2d) $i(4)$q(d) $i(4)$q(J)
    $i(4)$q(1+) $i(4)$q(2+) $i(4)$q(3+) $i(4)$q(4+) $i(4)$q(5+) $i(4)$q(6+) $i(4)$q(7+)
    $i(5)$q(6d) $i(5)$q(5d) $i(5)$q(4d) $i(5)$q(3d) $i(5)$q(2d) $i(5)$q(d) $i(5)$q(J)
    $i(5)$q(1+) $i(5)$q(2+) $i(5)$q(3+) $i(5)$q(4+) $i(5)$q(5+) $i(5)$q(6+)
    $i(6)$q(6d) $i(6)$q(5d) $i(6)$q(4d) $i(6)$q(3d) $i(6)$q(2d) $i(6)$q(d) $i(6)$q(m)
    $i(6)$q(M) $i(6)$q(1+) $i(6)$q(2+) $i(6)$q(3+) $i(6)$q(4+) $i(6)$q(5+) $i(6)$q(6+)
    $i(7)$q(6d) $i(7)$q(5d) $i(7)$q(4d) $i(7)$q(3d) $i(7)$q(2d) $i(7)$q(d) $i(7)$q(m)
    $i(7)$q(M) $i(7)$q(1+) $i(7)$q(2+) $i(7)$q(3+) $i(7)$q(4+) $i(7)$q(5+) $i(7)$q(6+)
```

```
    $i(8)$q(6d) $i(8)$q(5d) $i(8)$q(4d) $i(8)$q(3d) $i(8)$q(2d) $i(8)$q(d) $i(8)$q(J)
}
```

Variável `intervalsEqual` Intervalos nos sitema igual.

```
public common intervalsEqual {
  $i(1)$q(J) $i(2)$q(m) $i(2)$q(M) $i(3)$q(m) $i(3)$q(M) $i(4)$q(J)
  $i(4)$q(1+) $i(5)$q(J) $i(6)$q(m) $i(6)$q(M) $i(7)$q(m) $i(7)$q(M) $i(8)$q(J)
}
```

Construtor Aplica os valores padrão e lê os dados de entrada.

```
constructor { args } {
  loadDefaults
  readInput $args
}
```

Método `loadDefaults` Aplica os valores padrão.

```
protected method loadDefaults {} {
  setCod italian
  configure -direction true
  configure -octaveEquiv false
  configure -system tonal
}
```

Método `readInput` Lê a entrada de dados e verifica erros.

```
protected method readInput { input } {
  if { $input == ''} {
    error '' no argument''
  } elseif {[ check $input 1] == 0} {
  } else {
    set origNote $input
    set origNoteCode [name2number $input]
  }
}
```

Método `setCod` Define o tipo de codificação.

```
protected method setCod {input} {
  switch $input {
    italian {
      array set s  ''1 \ 2 \#\# 3  t \# 4 q \# 5 p \# 6 s \# 7 h \#''
      array set f ''1 b 2 bb 3 tb 4 qb 5 pb 6 sb 7 hb''
      array set n '' c do d re e mi f fa g sol a la b si ''
      array set i ''1 prima 2 seconda 3 terza 4 quarta 5 \
                    quinta 6 sexta 7 setima 8 octava ''
      array set q ''6 o sd 5o pd 4o qd 3o td 2o dd 1o d m m J J \
                    M M 1+ A 2+ dA 3+ tA 4+ qA 5+ qA 6+ sA 7+ hA''
    }
```

```
12      cifra {
          array set s   ‘‘1 \ 2 \#\# 3   t \# 4 q \# 5 p \# 6 s \# 7 h \#’’
14        array set f ‘‘1 b 2 bb 3 tb 4 qb 5 pb 6 sb 7 hb’’
          array set n ‘‘ c c d d e e f f g g a a b b’’
16        array set i ‘‘1 1 st 2 2 th 3 3 th 4 4 th 5 5 th 6 6 th 7 7 th 8 8 th’’
          array set q ‘‘6 o sd 5o pd 4o qd 3o td 2o dd 1o d m m J J \
18                    M M 1+ A 2+ dA 3+ tA 4+ qA 5+ qA 6+ sA 7+ hA‘‘
        }
20      lily {
          array set s ‘‘1 is 2 isis 3 t \ 4 q \# 5 p \# 6 s \# 7 h \#’’
22        array set f ‘‘1 es 2 eses 3 tb 4 qb 5 pb 6 sb 7 hb’’
          array set n ‘‘ c c d d e e f f g g a a b b’’
24        array set i ‘‘1 1 st 2 2 th 3 3 th 4 4 th 5 5 th 6 6 th 7 7 th 8 8 th’’
          array set q ‘‘6 o sd 5o pd 4o qd 3o td 2o dd 1o d m m J J \
26                    M M 1+ A 2+ dA 3+ tA 4+ qA 5+ qA 6+ sA 7+ hA‘‘
      }
28    german {
          array set s ‘‘1 is 2 isis 3 t \ 4 q \# 5 p \# 6 s \# 7 h \#’’
30        array set f ‘‘1 es 2 eses 3 tb 4 qb 5 pb 6 sb 7 hb’’
          array set n ‘‘ c c d d e e f f g g a a b h’’
32        array set i ‘‘1 1 st 2 2 th 3 3 th 4 4 th 5 5 th 6 6 th 7 7 th 8 8 th’’
          array set q ‘‘6 o sd 5o pd 4o qd 3o td 2o dd 1o d m m J J \
34                    M M 1+ A 2+ dA 3+ tA 4+ qA 5+ qA 6+ sA 7+ hA‘‘
        }
36      portuguese {
          array set s   ‘‘1 \ 2 \#\# 3   t \# 4 q \# 5 p \# 6 s \# 7 h \#’’
38        array set f ‘‘1 b 2 bb 3 tb 4 qb 5 pb 6 sb 7 hb’’
          array set n ‘‘ c dó d ré e mi f fá g sol a lá b si ’’
40        array set i ‘‘1 1 a 2 2 a 3 3 a 4 4 a 5 5 a 6 6 a 7 7 a 8 8 a’’
          array set q ‘‘6 d sd 5d pd 4d qd 3d td 2d dd d d m m J J \
42                    M M 1+ A 2+ dA 3+ tA 4+ qA 5+ qA 6+ sA 7+ hA‘‘
      }
44    num12 {
        configure −system equal
46    }
      num96 {
48      configure −system tonal
      }
50  }
    set codification $input
52  }
```

Método Modifica a codificação padrão através da mudança de índices dos *arrays*

changeCod

```
    protected method changeCod {input} {
2     if {[ lsearch $nonNumCodes $input] ≠ −1} {
        set currentNote [ number2name $origNoteCode]
4     } elseif {[ lsearch $numericCodes $input ] ≠ −1} {
        set currentNote [ name2number $origNoteCode]
6     } else {
```

```
        return ‘‘error’’
8     }
    }
```

Método `name2number` Converte um item não-numérico para seu valor numérico, dependendo da codificação

```
  public method name2number {input} {
2   if {[ string is digit $input ] == 0} {
      return [check [lsearch [subst $notelist ] $input ] 1]
4   } else {
      return [check [modulo $input ] 1]
6   }
  }
```

Método `number2name` Converte um item numérico para seu valor não-numérico, dependendo da codificação

```
  public method number2name {input} {
2   return [lindex [subst $notelist ] [modulo $input]]
  }
```

Método `modulo` Retorna o módulo de um valor númerico, dependendo da codificação.

```
  protected method modulo {input} {
2   return [expr $input % $sysMod]
  }
```

Método `export` Método usado pelo conjunto de testes para testar os métodos protegidos e não-públicos.

```
  public method export {args} {
2   eval $args
  }
4}
```

B.3.3 Classe set

Define codificações e operações em conjunto de notas.

```
itcl::class musiclib::Set {
2 inherit pitch
```

Variável `currentset` Retorna o conjunto corrente na operação.

```
  public variable currentset
```

Variável `origset` Retorna o conjunto original como entrado pelo usuário.

```
  public variable origset
```

Variável `origsetCode` Retorna o código do conjunto original como entrado pelo usuário.

```
public variable origsetCode
```

Variável `codification` Determina a codificação na qual o conjunto será mostrado.

```
public variable codification lily {
  setCod $codification
  changeCod $codification
  return $currentset
}
```

Construtor Carrega os valores padrões e lê os argumentos de entrada.

```
constructor { input } {
  loadDefaults
  readInput $input
}
```

Método `readInput` Lê a entrada de dados e verifica erros.

```
protected method readInput { input } {
  if { $input == ‘‘’’} {
    return ‘‘ no argument’’
  } elseif {[ check $input 1] == 0} {
  } else {
    set origset $input
    set currentset $input
    set origsetCode [ applyFunction $origset name2number]
  }
}
```

Método `changeCod` Define o tipo de codificação.

```
protected method changeCod {input} {
  if {[ lsearch $nonNumericCodes $input] ≠ −1} {
    set currentset [ applyFunction $origsetCode number2name]
  } else {
    set currentset [ applyFunction $origsetCode name2number]
  }
}
```

Método `applyFunction` Efetua uma operação linear em cada elemento da lista. Pode ser usado para transposição, multiplicação, etc.

```
protected method applyFunction { inputset operation {modif ‘‘’’}} {
  set result ‘‘’’
  foreach item $inputset {
    set tmp [eval $operation $item $modif]
    lappend result $tmp
  }
```

```
    return $result
8 }
}
```

B.3.4 Classe phrase

Define codificação de “frases” musicais em um nível mais alto, com indicação de duração e articulação.

```
class MusicLib::phrase {
2  inherit set
```

Método `currentphrase` Valor da frase corrente.

```
public variable currentphrase
```

Método `codification` Determina a codificação da frase.

```
public variable codification {
2   setCod $codification
    changeCod $codification
4   return $currentphrase (note)
}
```

Método `origphrase` Retorna a frase original, como entrada pelo usuário.

```
public variable origphrase
```

Método `origphraseCode` Retorna o código da frase original, como entrada pelo usuário.

```
public variable origphraseCode
```

Construtor Carrega os valores padrão e lê a entrada como uma lista válida do TCL.

```
constructor { args } {
2   loadDefaults
    set tmp [lindex [concat $args ] 0]
4   readInput [concat $tmp]
}
```

Método `readInput` *Container* para ler a entrada.

```
protected method readInput { input } {
2   if {[ errorList $input ] == 0} {

do something

      puts erro
2   } else {
      parser $input
4   }
}
```

Método changeCod Modifica a codificação.

```
protected method changeCod {input} {
  if {[lsearch $nonNumericCodes $input] ≠ −1} {
    set currentphrase (note ) [ applyFunction $origphraseCode(note ) number2name]
  } else {
    set currentphrase (note ) [ applyFunction $origphraseCode(note ) name2number]
  }
}
```

Método parser Usa expressões regulares para ler os dados de entrada.

sustenido `([#]*)?`

durações `([0-9]*)?`

ponto `([̇]*)?`

sinais de oitava `(['',]*)?`

sinais de articulação `(-.)?`

```
public method parser { input } {
  set parserString {([ A−Za−záéíóú]∗)?([\]∗)?([' , ]∗)?([0 −9]∗)?([\ . ]∗)?(−.)?}
  set origphrase (note ) {}
  set origphrase (oct ) {}
  set origphrase (dur ) {}
  set origphrase ( art ) {}
  set origphraseCode(note ) {}
  set origphraseCode( oct ) {}
  set origphraseCode(dur ) {}
  set origphraseCode( art ) {}
  set lastitem (oct ) ‘‘’’
  set lastitem (dur ) ‘‘’’

  foreach item $input {
    regexp −nocase −all −− $parserString $item match note sharp oct dur dot art

    append tmp $note $sharp $oct $dur $dot $art

    if {[ string compare $tmp $item] ≠0} {
      puts ‘‘ erro no parser ’’
    } else {
      lappend origphrase (note ) $note$sharp
      lappend origphrase (oct ) $oct
      lappend origphrase (dur ) $dur$dot
      lappend origphrase ( art ) $art

      if { $oct == ‘‘’’} {
        set oct $lastitem (oct )
      }
```

```
30          if {$dur == ‘‘’’} {
              set dur $lastitem(dur)
32          }

34          lappend origphraseCode(note) [name2number $note$sharp]
            lappend origphraseCode(oct) [oct2code $oct]
36          lappend origphraseCode(dur) $dur$dot
            lappend origphraseCode(art) [art2code $art]
38
            set lastitem(oct) $oct
40          set lastitem(dur) $dur
          }
42        set tmp ‘‘’’
        }
44    }
```

Método Converte os sinais de oitava para código numérico.

`oct2code`

```
public method oct2code {input} {
2
  regexp -nocase -all -- {([’]*)?([ ,]*)?} $input match sup inf
4
  if {$sup ≠ ‘‘’’} {
6   return [expr 3 + [string length $input]]
  } elseif {$inf ≠ ‘‘’’} {
8   return [expr 3 - [string length $input]]
  } elseif {$match == ‘‘’’} {
10  return 3
  } else {
12  return error
  }
14 }
```

Método Converte o código numérico para sinal de oitava do lilypond

`octcode2lily`

```
public method octcode2lily {input} {
2   if {$input ≥ 4} {
      set ncomma [expr $input - 3]
4     for {set x 1} {$x ≤ $ncomma} {incr x} {
        append tmp ’
6     }
      return $tmp
8   } elseif {$input < 3} {
      set ncomma [expr 3 - $input]
10    for {set x 1} {$x ≤ $ncomma} {incr x} {
        append tmp ,
12    }
      return $tmp
14  } elseif {$input == 3} {
      set ncomma 0
```

```
16        return $ncomma
        } else {
18        return error
        }
20    }
```

Método Converte o valor da duração para código.

`dur2code`

```
public method dur2code {input} {
2   if {[regexp {[[0-9]*\.]*} $input] == 1} {
      return [applydot $input]
4   } else {
      return [expr 4.0/$input]
6   }
}
```

Método Aplica o valor da duração em pontos de aumento.

`applydot`

```
public method applydot {input} {
2   regexp {([0-9]*)?([\.]*)?} $input string number dots
    set dotcode [string length $dots]
4   set note 0
    for {set x 0} {$x <= $dotcode} {incr x} {
6     set note [expr $note + (1.0/($number * pow(2,$x)))]
    }
8   return $note
}
```

Método Converte a articulação para código numérico.

`art2code`

```
public method art2code {input} {
2 }
```

Método Gera saída do csound para teste.

`test(csoundOut)`

```
public method test(csoundOut) {} {
2
    set csNote $currentphrase(note)
4   set csOct [applyFunction $origphraseCode(oct) `` expr 4 +'']
    set csDur [applyFunction $origphraseCode(dur) dur2code]
6   set lastdur 0
    set start 0
8
    set saida [open test/test.sco w]
10
    puts $saida ``f1 0 8192 10 1''
12  foreach item1 $csNote item2 $csOct item3 $csDur {
      if {$item1 <= 9} {
14      set zero 0
      } else {
```

```
16            set zero ''''
            }
18          puts $saida  '' i1  $lastdur  \t$item3\t$item2.$zero$item1 ''
            set  lastdur  [expr  $lastdur +$item3]
20      }
        puts $saida '' e''
22      close  $saida
    }
```

Método Gera saída do lilypond para teste.

`test(csoundOut)`

```
public method test( lilyOut ) {} {
2
    set lyNote  $currentphrase (note)
4   set lyOct [ applyFunction  $origphraseCode(oct )  octcode2lily ]
    set lyDur $origphraseCode(dur)
6   set lyArt  $origphrase ( art )
    set  saida [ open  test /  test.ly   w]
8
    foreach item1 $lyNote item2 $lyOct item3 $lyDur item4 $lyArt {
10      append notes '' $item1$item2$item3$item4 ''
    }
12
    puts $saida  ''\\ score
14  \{   \\ notes
    \{       \\ time 2/4
16     \\ clef   treble
    $notes
18   \}\}''

20    close  $saida
    }
22}
```

B.4 xmlparser

B.4.1 Introdução

A biblioteca xmlparser define uma classe chamada `xmlparser` para lidar com os instrumentos do csoundXML, o DTD, e a bilioteca CXL. Para detalhes ver a seção 7.1.1.

B.4.2 Classe xmlparser

Define métodos para ler e exportar dados do csoundXML e CXL.

```
class xmlparser::xmlparser {
```

Variável `rootout` O node raiz do documento de saída.

```
public variable rootout
```

Variável `pars` Array para valores de parâmentros.

```
public variable pars
```

Variável `cxlfile` Arquivo da CXL.

```
public variable cxlfile
```

Variável `pfield` Primeiro campo-p “livre”, i.e., não restrito.

```
public variable pfield 5
```

Construtor Cria uma raiz DOM pronta para receber dados, e define o arquivo CXL.

```
constructor {} {
  set doc [dom createDocument ‘‘opcodes’’]
  set rootout [$doc documentElement]

  dom createNodeCmd elementNode defpar
  dom createNodeCmd elementNode opcode
  dom createNodeCmd elementNode defOpcode
  dom createNodeCmd elementNode default
  dom createNodeCmd elementNode par
  dom createNodeCmd elementNode out
  dom createNodeCmd elementNode output
  dom createNodeCmd elementNode outtype
  dom createNodeCmd elementNode description
  dom createNodeCmd textNode t

  set cxlfile ~/TESE/Monochordum/src/monochordum/data/opcodes.xml
}
```

Método `readxmlfile` Lê arquivo XML como entrada, mapeia ele na memória e fecha arquivo no disco rígido.

```
public method readxmlfile { input } {
  set tmp $input
  set size [ file size $tmp]
  set fd [ open $tmp]
  return [ read $fd $size ]
  close $fd
}
```

Método `createRoot` Lê arquivo mapeado como DOM e cria elemento raiz.

```
public method createRoot { input } {
  set xmldoc [dom parse $input ]
  return [ $xmldoc documentElement]
}

public method exportXML {} {
  return [ $rootout asXML]
}
```

Método `readParameters` Procura por parâmetros no elemento `defpar`, depois pelo atributo `auto="yes"` nos opcodes. Tem o cuidado de colocar descrição vazia onde não tem.

```
public method readParameters { root } {
  set param [ $root selectNodes {/ instr / defpar [@auto='yes']}]
  set opcodepars [ $root selectNodes {/ instr /opcode/par[@auto='yes']}]
  set x 0
  set size [ llength $opcodepars]
  foreach item $opcodepars {
    set parentnode [ $item parentNode]
    set pars ( size ) $size
    set pars ($x,opcodeId ) [ $parentnode getAttribute id ]
    set pars ($x,name) [ $item getAttribute name]
    set descriptionnode [ $item selectNodes description / text ()]
    if { $descriptionnode ≠ ‘’’} {
      set pars ( $x,descr ) [ $descriptionnode nodeValue]
    } else {
      set pars ( $x,descr ) ‘’’
    }
    incr x
  }
  return 0
}
```

Método `readFunctions` Lê as funções definidas no arquivo de funções.

```
public method readFuncNames {root} {
  set param [$root selectNodes {/ functions / deffunc [@name]}]
  foreach item $param {
    lappend tmp [$item getAttribute id]
  }
  return $tmp
}
```

Método `whatOrder` Pergunda que ordem o parâmetro deve ter para CXL.

```
public method whatOrder {input} {
  set tmp [ readxmlfile $cxlfile ]
  set root [ createRoot $tmp]
  set opcode [$root selectNodes \
      ''/ opcodes/defOpcode\[@name='oscil'\]/par\[ @name='$input'\]'']
  return [$opcode getAttribute order]
}
```

Método `xmlExpression` Aceita um node de entrada que contem uma expressao para ser convertida para csound.

```
public method xmlExpression {input} {
  return expressao
}
```

Método `parValue` Retorna o valor de vvalue, expr, ou number.

```
public method parValue {input} {
  set value [$input getAttribute vvalue '']
  set expr [$input selectNodes expr]
  set number [$input selectNodes expr]
  if {$value ≠ ''} {
    return $value
  } elseif {$expr ≠ ''} {
    return [xmlExpression $expr]
  } elseif {$number ≠ ''} {
    return $number
  }
}
```

Método `checkReserved` Verifica se o parametro é p3 ou p4 (dur ou amp).

```
public method checkReserved {input} {
  set name [$input getAttribute name]
  if {$name == ''amplitude ''} {
    return p4
  } elseif {$name == '' duration ''} {
    return p3
  }
}
```

Método `statistics` Retorna dados estatísticos sobre o instrumento.

```
public method statistics { root } {
  set functions [ llength [ $root selectNodes {// par[@name='function']}]]
  set opcodes [ llength [ $root selectNodes {// opcode}]]
  set parameters [ llength [ $root selectNodes {// defpar }]]
  set expressions [ llength [ $root selectNodes {// expr}]]
  set export [ llength [ $root selectNodes {// *[@auto='yes']}]]
  set outputs [ llength [ $root selectNodes {// outtype }]]
  return `` functions $functions opcodes $opcodes variables $parameters \
          expressions $expressions parameters [ expr $export-$functions ] \
          outputs $outputs ``
}
```

Método `xmlOpcode` Converte elemento opcode para csound.

```
public method xmlOpcode {root} {
  set nodes [ $root selectNodes {*}]
  set type [ $root getAttribute type ]

  set params ```

  foreach j $nodes {
    set node [ $j nodeName]
    switch $node {
      out { set out [ $j getAttribute id ]}
      par {
```

le atributo nome

```
        set name [$j getAttribute name]
```

pergunda ordem para CXL

```
        set order [ whatOrder $name]
```

verifica se é reservado (p3 ou p4)

```
        set reserved [ $j getAttribute reserved `` no '']
        set auto [ $j getAttribute auto `` no '']

        if { $reserved == `` yes ''} {
          set value [ checkReserved $j ]
          lappend params $order $value
        } elseif {$auto == `` yes ''} {
```

verfifica se é auto, se nao for o valor está ou em vvalue (variavel) ou em <number> ou <expr>

```
          set value p$pfield
          lappend params $order $value
          incr pfield
        } elseif {$auto == `` no ''} {
          set value [ parValue $j ]
```

```
6                lappend params $order  $value
               }
8            }
             description  {}
10         }
       }
12     puts $params
       puts $out
14   }
```

Método `xml2csound` Converte instrumento descrito em XML para csound.

```
public method xml2csound {root} {
2    set param [$root  selectNodes  {/ instr /*}]
     foreach item  $param {
4      set  nodename [$item nodeName]
       switch  $nodename {
6        opcode {xmlOpcode $item}
         defpar {}
8        description  {}
         output {}
10       default {ups}
       }
12   }
}
```

Método `defOpcodeXML` Lê uma entrada `opcode` do arquivo XML e "converte" para uma entrada que define opcodes no CXL.

```
public method defOpcodeXML {root base} {
2    set name [$root  selectNodes  $base/name/text ()]
     set  description  [ $root  selectNodes  $base/ description / text ()]
4    set  par [ $root  selectNodes  $base/par/ text ()]
     set  out [ $root  selectNodes  $base/out/ text ()]
6    set  par2 [ $root  selectNodes  $base/par]

8    if {$out ==  ‘‘’’} {
       set  outvalue  \‘‘\’’
10   } else {
       set  outvalue  [$out  nodeValue]
12   }

14   if {$name ==  ‘‘’’} {
       set  namevalue  \‘‘\’’
16   } else {
       set  namevalue [$name nodeValue]
18   }

20   if { $description  ==  ‘‘’’} {
       set  descvalue  \‘‘\’’
```

```
22      } else {
          set descvalue [ $description nodeValue]
24      }

26      if {$name ≠ ‘‘’’} {
          $rootout appendFromScript {
28          defOpcode ‘‘name $namevalue’’ {
              set x 0
30            if {$par ≠ ‘‘’’} {
                foreach item $par {
32                set parnode [lindex $par2 $x]
                  set stype [string range [$item nodeValue] 0 0]
34                set parnamevalue [lindex [$item nodeValue] 0]
                  par ‘‘ order [$parnode getAttribute id] type $stype name $parnamevalue’’ {
36                  t ‘‘’’
                  }
38                incr x
                }
40            }
              out ‘‘ type $outvalue outtype mono’’ {}
42            description {} { t $descvalue}
            }
44        }
        } else {
46        puts stderr ‘‘ não há o elemento $base’’
        }
48    }
  }
```

B.5 Conversor orc2xml

B.5.1 Introdução

A biblioteca Orc2xml é uma re-implementação em *[incr TCL]* de um conversor de orquestras do Csound para XML escrito em Bison e Flex (Levine, Mason, e Brown 1992) pelo autor dessa tese. Apesar do uso de geradores de *parser* como Flex e `Bison` permite a criação de um *parser* robusto e confiável, a implementação em *[incr TCL]* permite um código mais fácil e rápido de ler e modificar, conversão mais fácil para XML (graças à bilioteca `tdom`), e melhor integração com as outras bibliotecas por nós desenvolvidas.

B.5.2 Classe orc2xml

Define a classe orc2xml dentro do *namespace* `orc2xml`.

```
::itcl::class orc2xml::orc2xml {
```

Variável `opcodelist` Lista com opcodes reconhecidos do csound. Apenas um pequeno grupo é necessário para reconhecer os instrumentos do ACCCI.

```
public variable opcodelist {
  balance buzz envlpx expon expseg fof gbuzz init
  line linen linseg oscil oscil1 oscil1i oscili phasor
  pluck pvoc rand randh randi reson reverb start table
  tablei out outs outs1 outs2 display timout soundin
}
```

Variável `functionslist` Lista com funções do csound. Apenas um pequeno grupo é necessário para reconhecer os instrumentos do ACCCI.

```
public variable functionslist {
  cpspch cpsoct int log cos exp
}
```

Variável `doc` Variável que conterá a raiz do DOM. No caso, corresponde ao elemento `<instr>`. Ela é inicializada no construtor.

```
public variable doc
public variable root
public variable channelnames {mono stereo quad hex oct}
```

Construtor Inicializa as variáveis `doc` e `root` e cria os elementos básico do csoundXML.

```
constructor {instrName} {
  set doc [dom createDocument instr]
  set root [$doc documentElement]

  $root setAttribute name ''$instrName''
  dom createNodeCmd elementNode defpar
  dom createNodeCmd elementNode opcode
  dom createNodeCmd elementNode default
```

```
    dom createNodeCmd elementNode par
    dom createNodeCmd elementNode output
    dom createNodeCmd elementNode outtype
    dom createNodeCmd textNode t
}
```

Método `cleaInstr` Prepara o instrumento de entrada eliminando linhas em brando e comentários.

```
public method cleaInstr { input } {
    set tmpfile /tmp/tmp.1
    set tmp [open $tmpfile w]
    puts $tmp $input
    close $tmp
    return [exec grep -v ''^;" $tmpfile | sed "s /; .* // " | grep -v " ^\[ \ t \] *$ ''']
}
```

Método `readInstBlock` Retorna um bloco de instrumento da entrada, se houverem diversos blocos de instrumento, chamar novamente esse método com o restante da entrada (i.e., a entrada original menos o bloco extraído). `posEnd` tem que ser somado 4 para pegar a palavra "endin", de outro modo só pegaria a primeira letra, "e". Usava `set posStart [string first "instr" $input]` para achar o começo do bloco, mas estava dando falsos positivos dentro de comentários. O ideal é remover todos os comentários e linhas em branco primeiro, o que está sendo feito por `cleanInstr`.

```
public method readInstBlock { input } {
    set input [ cleaInstr $input ]
    regexp -indices -- {[ \t ] *instr [ \ t ]+[0-9]+} $input da1
    set posStart [lindex $da1 0]
    set posEnd [string first '' endin '' $input ]
    set posEnd [expr $posEnd + 4]
    return [ string range $input $posStart $posEnd]
}
```

Método `trimComment` Corta os comentários no final de linha.

```
public method trimComment {input} {
    set tmp [ split $input ''; '']
    return [ lindex $tmp 0]
}
```

Método `extractVarType` Extrai a primeira letra de uma variável do csound (e.g. `i` em `ivar`).

```
public method extractVarType { input } {
    return [ string range $input 0 0]
}
```

Método `extractVarCore` Extrai o nome de uma variável do csound sem a letra de tipo (e.g. `var` em `ivar`).

```
public method extractVarCore { input } {
        return [ string range $input 1 end]
}
```

Método `defparXML` Gera um elemento de parâmetro em XML.

```
public method defparXML {input} {
  set var [lindex $input 0]
  set value [lindex $input 2]
  set name [extractVarCore $var]
  set type [extractVarType $var]
  $root appendFromScript {
    defpar ‘‘name $name type $type’’ {
      default { t ‘‘$value’’ }
    }
  }
}
```

Método `ifOpcode` Verifica se a entrada contém um opcode do csound.

```
public method ifOpcode {input} {
  set test1 [lindex $input 0]
  set test2 [lindex $input 1]
  if {[lsearch $opcodelist $test1 ] ≠ −1} {
    return $test1
  } elseif {[lsearch $opcodelist $test2 ] ≠ −1} {
    return $test2
  } else {
    return −1
  }
}
```

Método `parseVar` Parseia uma variável para ver se é um valor númerico, uma expressão matemática, ou uma variável simples. No futuro usará o MathML, agora ele só retira a primeira letra das váriaveis usando “força bruta” (expressões regulares). Aas duas primeiras expressões regulares tem que ler desde o início para garantir que a expressão não será achada no meio.

```
public method parseVar {input} {
  if {[regexp −− {^[0−9\.]+$} $input]} {
    return $input
  } elseif {[regexp {^p[0−9]+$} $input]} {
    return $input
  } elseif {[regexp {^(i|g|a|k)[a−zA−Z0−9\.]+$} $input]} {
    return [string range $input 1 end]
  } elseif {[regexp {[\/\ ∗\− \+\(\)]+} $input]} {
    regsub −all −− {(i|g|a|k)([a−zA−Z0−9\.]+)} $input ‘‘\\2’’ tmp
    return $tmp
  } elseif {[regexp {[a−zA−Z0−9]+\.[a−zA−Z0−9]+} $input]} {
    return $input
  } else {
    error {error in regular expression}
  }
}
```

Método opcodeXML

Gera um elemento com opcode. `opcname` extrai o nome do opcode, `opcpos` extrai a posição, se a posição for 0 não tem saida, se for 1 tem. Cada item do `foreach` tem que ser validado para verificar se tem expressões matemáticas, se não tiver é só veriricar se é um valor numérico ou uma variável. Se for uma variável, extrair o tipo. Um caso especial é se o opcode for `out`, porque tem um elemento próprio. Por padrão coloca cada saida em um canal separado.

```
public method opcodeXML {input} {
  set opcname [ifOpcode $input]
  set opcpos [lsearch $input $opcname]
  if {[regexp {out} $opcname]} {
    set inlist [lrange $input 1 end]
    set outlist [split $inlist ‘‘,’’]
    set outsize [llength $outlist]
    $root appendFromScript {
      output ‘‘’’ {
        set x 0
        foreach item $outlist {
          outtype ‘‘name [lindex $channelnames $x]’’ {
            t ‘‘[parseVar [string trim $item]]’’
            incr x
          }
        }
      }
    }
  } elseif {$opcpos == ‘‘0’’} {
    set inlist [lrange $input 1 end]
    foreach item [split $inlist ‘‘,’’] {
      $root appendFromScript {
        opcode ‘‘name $opcname’’ {
          par {name ‘‘’’} {
            t ‘‘[parseVar [string trim $item]]’’
          }
        }
      }
    }
  } elseif {$opcpos == ‘‘1’’} {
    set outvar [lindex $input 0]
    set inlist [lrange $input 2 end]
    set id [extractVarCore $outvar]
    set type [extractVarType $outvar]
    set node [$root selectNodes /instr/opcode]
    $root appendFromScript {
      opcode ‘‘name $opcname id $id type $type’’ {
        foreach item [split $inlist ‘‘,’’] {
          par {name ‘‘’’} {
            t ‘‘[parseVar [string trim $item]]’’}
          }
        }
      }
```

```
44      } else {
          error { this shouldn't happen!}
46      }
    }
```

Método convert2xml Método usando força bruta para converter para XML. Converte linha-por-linha. O primeiro `if` detecta o início do instrumento. Os `elseif` restantes parseam os opcodes e elementos. Cada linha é lida como lista, desse modo é possível controlar o que será lido e ignorar o que não ser quer (e.g. comentários, que são retirados com `trimComment`).

```
  public method convert2xml {input} {
2   set instrBlock [ readInstBlock $input ]
    foreach line [ split $instrBlock ''\n''] {
4     if {[ regexp { instr } $line ]} {
      } elseif {[ regexp {^[ \t]*;} $line ]} {
6     } elseif {[ regexp {^[ \t]*$} $line ]} {
      } elseif {[ regexp {=} $line ]} {
8       defparXML [trimComment $line]
      } elseif {[ ifOpcode $line ] ≠ -1} {
10      opcodeXML [trimComment $line]
      } elseif {[ regexp {endin} $line ]} {
12    } else {
        error {error parsing line}
14    }
    }
16  return [ $root asXML]
  }
18
}
```

Referências Bibliográficas

Allen, James F. 1991. “Time and time again: the many ways to represent time.” *International Journal of Intelligent Systems* 6 (4): 341–355.

Allen, James F., e G. Ferguson. 1994. “Actions and events in interval temporal logic.” *Journal of Logic and Computation* 4, no. 5.

Anderson, David P., e Ron Kuivila. 1986. “Accurately timed generation of discrete musical events.” *Computer Music Journal* 10 (3): 48–55.

Assayag, Gérard, Carlos Agon, Joshua Fineberg, e Peter Hanappe. 1997. “An object oriented visual environment for musical composition.” *Proceedings of the 1997 International Computer Music Conference*. The Computer Music Association, 364–367.

Balaban, Mira. 1996. “The music structures approach to knowledge representation for music processing.” *Computer Music Journal* 20 (2): 96–111.

Bartetzki, Andre. 1997a, Julho. *CMask a stochastic event generator for Csound*. Disponível em `http://www.kgw.tu-berlin.de/~abart/CMaskMan/CMask-Manual.htm` (2001-04-10).

———. 1997b. *Csound score generation and granular synthesis with cmask*. Disponível em `http://www.kgw.tu-berlin.de/~abart/CMaskPaper/cmask-article.html` (2002-09-12).

Barton-Davis, Paul. 2001. *Quasimodo*. Disponível em `http://quasimodo.org/` (2001-04-29).

Bianchini, Riccardo. 2002. *CSGraph*. Disponível em `http://www.geocities.com/Heartland/Acres/4768/` (2002-11-27).

Bilmes, Jeff. 1992. “A model for musical rhythm.” *Proceedings of the 1992 International Computer Music Conference*. The Computer Music Association, 207–210.

Blasser, Peter. 1999. *Rocky*. Disponível em `http://www.oberlin.edu/~pblasser/rocky.html` (2001-04-10).

Boulanger, Richard, ed. 2000. *The Csound Book*. Boston: MIT Press.

Brandon, Stephen, e Leigh M. Smith. 2000. “Next steps from NeXTSTEP: music kit and sound kit in a new world.” *Proceedings of the 2000 International Computer Music Conference*. San Francisco: The Computer Music Association, 503–506.

Brandt, Eli. 2000. "Temporal type constructors for computer music programming." *Proceedings of the 2000 International Computer Music Conference*. San Francisco: The Computer Music Association, 328–331.

———. 2001. "Implementing temporal type constructors for music programming." *Proceedings of the 2001 International Computer Music Conference*. San Francisco: The Computer Music Association, 99–102.

———. 2002. "Temporal type constructors for computer music programming." Ph.D. diss., Carnegie Mellon University.

Brinkman, Alexander R. 1981. "Data structures for a music-11 preprocessor." *Proceedings of the 1981 International Computer Music Conference*. San Francisco: The Computer Music Association, 178–95.

———. 1984. "A data structure for computer analysis of musical scores." *Proceedings of the 1984 International Computer Music Conference*. San Francisco: The Computer Music Association, 233–242.

———. 2000. *Score11*. Disponível em `http://www.esm.rochester.edu/download.html` (2001-04-10).

Burton, Alexandre, e Jean Piché. 1998a. *Cecilia*. Disponível em `http://www.musique.umontreal.ca/electro/CEC/index.html` (2001-04-10).

———. 1998b. *The Cybil Composition Language*. Disponível em `http://www.musique.umontreal.ca/electro/CEC/man/cybil.html` (2001-04-10).

Buxton, W., W. Reeves, R. Baecker, e L. Mezei. 1978. "The use of hierarchy and instance in a data structure for computer music." *Computer Music Journal* 2 (4): 10–20.

Buxton, W., R. Sniderman, W. Reeves, R. Patel, e R. Baecker. 1979. "The evolution of the SSSP score-editing tools." *Computer Music Journal* 3 (4): 14–25.

Byrd, Donald. 1994. "Music notation software and intelligence." *Computer Music Journal* 18 (1): 17–20.

Cahill, M, e D. Ó Maidín. 2001. "Score processing for MIR." *Proceedings of the Second Annual International Symposium on Music Information Retrieval*. Bloomington, 59–64.

Cahill, Margaret. 1998. "The translation of finale's enigma file format for CPNView." Master's thesis, University of Limerick.

Castan, Gerd. 2002. *Music notation*. Disponível em `http://www.music-notation.info/` (2002-12-13).

Castan, Gerd, Michael Good, e Perry Roland. 2001. "Extensible markup language (XML) for music applications: an introduction." In *The virtual score: representation, retrieval, restoration*, edited by Walter B. Hewlett e Eleanor Selfridge-Field, Volume 12 of *Computing in Musicology*, 95–102. Massachusetts: The MIT Press.

Cemgil, A., P. Desain, e B. Kappen. 2000. "Rhythm quantization for transcription." *Computer Music Journal* 24 (2): 60–76.

Cemgil, A., B. Kappen, P. Desain, e H. Honing. 2001. "On tempo tracking: tempogram representation and Kalman filtering." *Journal of New Music Research* 29 (4): 259–273.

Chicha, Yannis, Florence Defaix, e Stephen M. Watt. 1999. *A C++ to XML translator*. The FRISCO consortium. Disponível em `http://www.nag.co.uk/Projects/Frisco/reports/cpp2xml.ps` (2002-09-12).

Clark, James, e Steve DeRose. 1999. "XML Path Language (XPath)." Technical Report, The World Wide Web Consortium.

Comajuncosas, Josep M. 2002a. *DirectHammond v1/v2*. Disponível em `http://www.csounds.com/jmc/index.htm` (2002-12-11).

———. 2002b. *JCM Strings*. Disponível em `http://www.csounds.com/jmc/index.htm` (2002-12-11).

———. 2002c. *JCN Synthesis Tutorial*. Disponível em `http://www.csounds.com/jmc/index.htm` (2002-12-11).

———. 2002d. *VoxDream*. Disponível em `http://www.csounds.com/jmc/index.htm` (2002-12-11).

Cooke, Andrew. 2001. *Rytmo*. Disponível em `http://www.andrewcooke.free-online.co.uk/jara/rytmo/index.html` (2001-04-10).

D., Michael, e I. Fujinaga. 2001. "Interpreting the semantics of music notation using an extensible and object-oriented system." *Ninth International Python Conference*.

Dahan, Kevin. 2001. "Csound: from the acoustical compiler to the sound synthesis ecosystem." *Csound Magazine*, Winter.

Dannenberg, R. B. 1986. "A structure for representing, displaying and editing music." *Proceedings of the 1986 International Computer Music Conference*. San Francisco: The Computer Music Association, 153–60.

———. 1989. "The canon score language." *Computer Music Journal* 13 (1): 47–56.

———. 1991. "Expressing temporal behavior declaratively." In *CMU Computer Science, A 25th Anniversary Commemorative*, edited by Richard F. Rashid, 47–68. ACM Press.

———. 1993a. "The implementation of nyquist, a sound synthesis language." *Proceedings of the 1993 International Computer Music Conference*. San Francisco: The Computer Music Association, 168–171.

———. 1993b. "Music representation: issues, techniques, and systems." *Computer Music Journal* 17 (3): 20–30.

———. 1994. "Abstract time warping of compound events and signals." *Proceedings of the 1994 International Computer Music Conference*. San Francisco: The Computer Music Association, 251–254.

———. 1997. "Machine tongues XIX: nyquist a language for composition and sound synthesis." *Computer Music Journal* 21 (3): 50–60.

Dannenberg, R. B., C. L. Fraley, e P. Velikonja. 1991. "Fugue: a functional language for sound synthesis." *Computer Music Journal* 24 (7): 36–42.

———. 1992. "A functional language for sound synthesis with behavioral abstraction and lazy evaluation." In *Readings in Computer-Generated Music*, edited by Denis Baggi. Los Alamitos, CA: IEEE Computer Society Press.

Dannenberg, R.B., P.W.M. Desain, e H.J. Honing. 1997. "Programming language design for music." In *Musical signal processing*, edited by C. Roads, S.T. Pope, e G. De Poli, 271–315. Lisse: Swets and Zeitlinger.

Debril, Didier, e Jean-Pierre Lemoine. 2000, Junho. *HPKComposer, a 3D Art composition tool for Csound*. Disponível em `http://perso.libertysurf.fr/hplank/hpkcomposer.html` (2001-04-10).

Decker, S., e G. Kendall. 1984. "A modular approach to sound synthesis software." *Proceedings of the 1984 International Computer Music Conference*. San Francisco: The Computer Music Association, 243–250.

Desain, P., e H. Honing. 1988. "LOCO: a composition microworld in logo." *Computer Music Journal* 12 (3): 30–42.

———. 1992. "Time functions function best as functions of multiple times." *Computer Music Journal* 16 (2): 17–34.

———. 1993a. "The mins of max." *Computer Music Journal* 17 (2): 3–11.

———. 1993b. "Tempo curves considered harmful." *Contemporary Music Review* 7 (2): 123–138.

———. 1996a. "Functional style." In *Lisp as a second language: composing programs and music*. Lisse: Swets and Zeitlinger.

———. 1996b. "Object-oriented style I." In *Lisp as a second language: composing programs and music*. Lisse: Swets and Zeitlinger.

Desain, P., H. Honing, R. Aarts, e R. Timmers. 2000. "Rhythmic aspects of vibrato." In *Rhythm perception and production*, edited by P. Desain e W. L. Windsor, 203–216. Lisse: Swets and Zeitlinger.

Desain, P., C. Jansen, e H. Honing. 2000. "How identification of rhythmic categories depends on tempo and meter." *Proceedings of the Sixth International Conference on Music Perception and Cognition*. Keele University, Departmentof Psychology, Keele, UK.

Deyer, Lounette M. 1984. "Toward a device independent representation of music." *Proceedings of the 1984 International Computer Music Conference*. San Francisco: The Computer Music Association, 251–255.

Diener, Glendon Ross. 1990. "Modeling music notation: a three-dimensional approach." Ph.D. diss., Stanford university.

Dodge, Charles, e Thomar A. Jerse. 1997. *Computer music: synthesis, composition, and performance*. 2nd. New York: Schirmer Books.

Droettboom, M. *A study of musical notation description languages*. Disponível em `http://gigue.peabody.jhu.edu/~mdboom/` (2002-07-09).

Droettboom, M., I. Fujinaga, K. MacMillan, M. Patton, J. Warner, G. S. Choudhury, e T. DiLauro. 2001. "Expressive and efficient retrieval of symbolic musical data." *Proceedings of the Second Annual International Symposium on Music Information Retrieval*. Bloomington, 173–178.

Gancarz, M. 1995. *The UNIX philosophy*. Boston, MA: Digital Press.

Gogins, Michael. 1998. "Music graphs for algorithmic composition and synthesis with extensible implementation in java." *Proceedings of the 1998 International Computer Music Conference*. San Francisco: The Computer Music Association, 369–376.

———. 2000a, Jan. "Re: SML (Synthesis Modelling Language)." saol-dev@media.mit.edu: MPEG-4 Structured Audio Mailing List.

———. 2000b. *Silence*. Disponível em `http://www.pipeline.com/~gogins/` (2001-04-10).

———. 2001. "Modernizing csound." *Csound Magazine*, Winter.

Good, Michael. 2001. "MusicXML for Notation and Analysis." In *The virtual score: representation, retrieval, restoration*, edited by Walter B. Hewlett e Eleanor Selfridge-Field, Volume 12 of *Computing in Musicology*, 113–124. Massachusetts: The MIT Press.

H., Henkjan. 1993. "Issues on the representation of time and structure in music." *Contemporary Music Review* 9:221–238.

Haken, Lippold, e Dorothea Blostein. 1993. "The tilia music representation: extensibility, abstraction, and notation contexts for the lime music editor." *Computer Music Journal* 17 (3): 43–58.

Hanappe, Peter. 1999. "Design and implementation of an integrated environment for music composition and synthesis." Ph.D. diss., University of Paris.

Hanna, Arne. 1999, Maio. *Mother*. Disponível em `http://www.geocities.com/SiliconValley/Peaks/3346/` (2001-04-10).

Harold, Elliotte Rusty. 1999. *XML bible*. New York, NY: IDG Books Worldwide.

Haynes, Stanley. 1980. "The musician-machine interface in digital sound synthesis." *Computer Music Journal* 4 (4): 23–44.

Honing, H. 1995. "The vibrato problem, comparing two solutions." *Computer Music Journal* 19 (3): 32–49.

———. 2001. "From time to time: the representation of timing and tempo." *Computer Music Journal* 35 (3): 50–61.

Hoos, H., K. Hamel, K. Renz, e J. Kilian. 2001. "Representing score-level music using the GUIDO music-notation format." In *The Virtual Score: Representation, Retrieval, Restoration*, edited by Walter B. Hewlett e Eleanor Selfridge-Field, Volume 12 of *Computing in Musicology*, 75–94. The MIT Press.

Hoos, H., K. Renz, e M. Görg. 2001. "GUIDO/MIR—an experimental musical information retrieval system based on GUIDO music notation." *Proceedings of the Second Annual International Symposium on Music Information Retrieval*.

Hoos, H. H., K. A. Hamel, e J. Kilian K. Renz. 1998. "The GUIDO music notation format—a novel approach for adequately representing score-level music." *Proceedings of the 1998 International Computer Music Conference*. San Francisco: The Computer Music Association, 451–454.

Huron, David Brian. 2002. "Music information processing using the humdrum toolkit: concepts, examples, and lessons." *Computer Music Journal* 26 (2): 11–26.

ISO/IEC. 1995. *Standard music description language*. Disponível em `ftp://ftp.techno.com/pub/SMDL/10743.ps` (2002-09-12).

———. 1999. *Information technology—coding of audio-visual objects*. Disponível em `http://sound.media.mit.edu/~eds/mpeg4/SA-FDIS.pdf` (2003-11-10).

Jaffe, David A. 1989. *From the classical software synthesis note-list to the NeXT score-file*. Redwood City: NeXT Computer, Inc.

———. 1991. "Musical and extra-musical applications of the NeXT music kit." *Proceedings of the 1991 International Computer Music Conference*. San Francisco: The Computer Music Association, 521–524.

Jaffe, David A., e Lee R. Boynton. 1989. "An overview of the sound and music kits for the next computer." *Computer Music Journal* 13 (2): 48–55.

Kaplan, Randy M. 1994. *Constructing language processors for little languages*. John Wiley & Sons.

Kay, Nils, e Peter Heeren. 2000, Julho. *JCself: a generator for Csound using a stochastic method*. Disponível em `http://www.peter-heeren.de/jcself/man.htm` (2001-04-10).

Koenen, Rob. 1999. "Overview of the MPEG standard." Technical Report, ISO/IEC JTC1/SC29/WG11.

Krasner, Glenn. 1980. "Machine tongues VIII: the design of a smalltalk music system." *Computer Music Journal* 4 (4): 4–14.

Kröger, Pedro. 2000, Outubro. "Organizando instrumentos do Csound usando macros." Edited by Silvia Malbrán e Favio Shifres, *IIIa Conferencia Iberoamericana de Investigacion Musical*. 153–157.

———. 2003a. "Definindo Mega-instrumentos com XML." *Anais do XIV Congresso da ANPPOM*.

———. 2003b. "From the concept of sections to events in Csound." *Proceedings of the 2003 International Computer Music Conference*. San Francisco: The Computer Music Association.

Kuehn, Mikel. 2001. *The nGen Manual*. Bowling Green State University.

Lamport, Leslie. 1994. *LaTeX: a document preparation system*. Addison Wesley Professional.

Laurson, M. 1999. "PWCollider: a visual composition tool for software synthesis." *Proceedings of the 1999 International Computer Music Conference*. San Francisco: The Computer Music Association, 20–23.

Lemos, Manuel. 2001, November. "MetaL: XML based meta-programming engine developed with PHP." *PHP Conference 2001*. PHP-Center and Software & Support Verlag, Frankfurt.

Lent, Keith, Russell Pinkston, e Peter Silsbee. 1989. "Accelerando: a real-time, general purpose computer music system." *Computer Music Journal* 13 (4): 54–64.

Levine, John R., Tony Mason, e Doug Brown. 1992. *Lex & Yacc*. 2. O'Reilly.

Loureiro, Maurício Alves. 1996. "necSO - uma linguagem de composição implementada num sistema de síntese e processamento de som do tipo music V (csound)." *Anais do VIII Encontro Anual da ANPPOM*.

Loy, Gareth. 1989. "Composing with computers—a survey of some compositional formalisms and music programming languages." In *Current Directions in Computer Music Research*, edited by M. V. Mathews e J. R. Pierce, 291–396. Cambridge, MA: MIT Press.

———. 2002. "The CARL system: premises, history, and fate." *Computer Music Journal* 26 (4): 52–60.

Loy, Gareth, e Curtis Abbott. 1985. "Programming languages for computer music synthesis, performance, and composition." *ACM Comput. Surv.* 17 (2): 235–265.

Lyon, Eric. 2002. "Dartmouth Symposium on the Future of Computer Music Software: A Panel Discussion." *Computer Music Journal* 26 (4): 13–30.

Makela, Markku. 2003. *Javasynth*. Disponível em `http://javasynth.sourceforge.net/` (2003-04-24).

Maldonado, Gabriel. 2003, Julho. "Re: [Csnd] Zak bugs?" csound@lists.bath.ac.uk: Csound Mailing List.

Marchal, Benoît. 2000. *XML: conceitos e aplicações*. São Paulo: Editora Berkeley. Trad. Daniel Vieira.

Marx. 2002. "XML and verbosity." *Kuro5hin*, Dezembro.

McCartney, James. 1996. "SuperCollider: a new real time synthesis language." *Proceedings of the 1996 International Computer Music Conference*. San Francisco: The Computer Music Association, 257–258.

———. 2002. "Rethinking the computer music language: supercollider." *Computer Music Journal* 26 (4): 61–68.

Meyyappan, Alagappan. 2000. GUI development using XML.

Miranda, Eduardo Reck. 1997, Julho. "Chaosynth." Technical Report, Laboratório de Música Eletroacústica de Santa Maria.

Moore, Richard. 1998. *Elements of computer music*. Pearson Education.

Mosterd, Eric James. 1999. "Developing a new way to transfer sheet music via the internet." Master's thesis, University of South Dakota.

Mounce, Steve. 2002. *Music encoding standards*. Disponível em `http://www.student.brad.ac.uk/srmounce/encoding.html` (2002-12-13).

Newcomb, S. R. 1991. "Standard music description language complies with hypermedia standard." *Computer* 24 (7): 76–80.

Oliveira, Jamary. 1994a. "Regra geral para a representação fracionária da duração musical." Em manuscrito.

———. 1994b. "Representação fracionária de grupos rítmico." Em manuscrito.

———. 1995. *Informática em música: o parâmetro altura*. Mestrado em Música/UFBA.

———. 2001a. "Em busca de uma codificação." *Cuadernos interamericanos de investigación en educación musical* 1 (2): 25–42 (Agosto).

———. 2001b. “Mutação I e II.” In *Música Eletro-acústica na Bahia*. CD com obras eletroacústicas editado pelo Programa de Pós-Graduação em Música da UFBA.

———. 2001c. “Mutação I e II.” Orquestra e Partitura do csound gentilmente cedidos pelo compositor.

Oppenheim, D. 1986. “The need for essential improvements in the machine composer interface used for the composition of electroacoustic computer music.” *Proceedings of the 1986 International Computer Music Conference*. San Francisco: The Computer Music Association, 443–445.

———. 1987. “The P-G-G environment for music composition.” *Proceedings of the 1987 International Computer Music Conference*. San Francisco: The Computer Music Association, 40–48.

———. 1989. “DMIX: an environment for composition.” *Proceedings of the 1989 International Computer Music Conference*. San Francisco: The Computer Music Association, 226–233.

———. 1990. “Quill: an interpreter for generating music objects within the DMIX environment.” *Proceedings of the 1990 International Computer Music Conference*. San Francisco: The Computer Music Association, 256–258.

———. 1991a. “SHADOW: an object-oriented performance-system for the DMIX environment.” *Proceedings of the 1991 International Computer Music Conference*. San Francisco: The Computer Music Association, 281–284.

———. 1991b. “Towards a better software-design for supporting creative musical activity.” *Proceedings of the 1991 International Computer Music Conference*. San Francisco: The Computer Music Association, 380–387.

———. 1992. “DMIX—a multi faceted environment for composing and performing computer music: its philosophy, design, and implementation.” *Proceedings of the Fourth Biennial Arts & Technology Symposium*.

———. 1993. “Slappability: a new metaphor for human computer interaction.” In *Music Education: An Artificial Intelligence Perspective*, edited by M. Smith, G. Wiggins, e A. Smaill. London: Springer-Verlag.

Ousterhout, John K. 1993. *Tcl and the Tk toolkit*. Addison-Wesley Publishing Company, Inc.

Pennycook, Bruce W. 1985. “Computer-music interfaces: a survey.” *ACM Comput. Surv.* 17 (2): 267–289.

Perry, David. 2000. “[Csnd] Idea for feature (long).” Mailing list 14 Fevereiro. csound@lists.bath.ac.uk: Csound Mailing List.

———. 2002. *Visual Orchestra*. Disponível em `http://www2.hku.nl/~perry/visorc/home.htm` (2002-11-27).

Phillips, Dave. 2001. “Linux audio plug-ins: a look into LADSPA.” *Oreillyne*.

Piche, J., e A. Burton. 1998. “Cecilia: a production interface to csound.” *Computer Music Journal* 22 (2): 52–55.

Pinkston, Russell F. 1995. “Rapid instrument prototyping system (RIPS): a new graphical user interface for instrument design in csound.” *1995 SEAMUS National Conference*.

Pope, Stephen Travis. 1989. "Considerations in the design of a music representation language." *Proceedings of the 1989 International Computer Music Conference*. San Francisco: The Computer Music Association, 246–248.

———. 1992. "The SmOKe music representation, description language, and interchange format." *Proceedings of the 1992 International Computer Music Conference*. San Francisco: The Computer Music Association.

———. 1993. "Machine tongues XV: three packages for software sound synthesis." *Computer Music Journal* 17 (2): 23–54.

———. 1995. "Computer music workstations I have known and loved." *Proceedings of the 1995 International Computer Music Conference*. San Francisco: The Computer Music Association, 127–133.

———. 1997. "Overview." In *Musical signal processing*, edited by C. Roads, S.T. Pope, e G. De Poli, 267–269. Lisse: Swets and Zeitlinger.

Puckette, M. 1984. "The M orchestra language." *Proceedings of the 1984 International Computer Music Conference*. San Francisco: The Computer Music Association, 17–20.

———. 1996. "Pure data: another integrated computer music environment." *Proceedings of the 1996 International Computer Music Conference*. San Francisco: The Computer Music Association, 269–272.

———. 1997. "Pure data: recent progress." *Proceedings*. Tokyo, Japan: Third Intercollege Computer Music Festival, 1–4.

———. 2002. "Max at seventeen." *Computer Music Journal* 26 (4): 31–43.

Puckette, M., e T. Apel. 1998. "Real-time audio analysis tools for Pd and MSP." *Proceedings of the 1998 International Computer Music Conference*. San Francisco: The Computer Music Association, 109–112.

Puxeddu, Maurizio Umberto. 2000, Dezembro. *PMask: a Python implementation of CMask*. Disponível em `http://web.tiscalinet.it/mupuxeddu/csound/index.html` (2001-04-10).

———. 2001. *CSFE: a Python/Tk CSound front-end*. Disponível em `http://web.tiscalinet.it/mupuxeddu/` (2001-04-10).

Ramsdell, John D. 2001, Janeiro. *Scheme Score—transform Csound score files using Scheme*. Disponível em `http://www.ccs.neu.edu/home/ramsdell/tools/scmscore/scmscore.html` (2001-04-10).

Renz, K., e H. H. Hoos. 1998. "A web-based approach to music notation using GUIDO." *Proceedings of the 1998 International Computer Music Conference*. San Francisco: The Computer Music Association, 455–458.

Roads, Curtis. 1996. *The computer music tutorial*. Massachusetts: The MIT Press.

Rodet, Xavier, e Pierre Cointe. 1984. "FORMES: composition and scheduling of processes." *Computer Music Journal* 8 (3): 32–50.

Rogers, J., J. Rockstroh, e P. Batstone. 1980. "Music-time and clock-time similarities under tempo changes." *Proceedings of the 1980 International Computer Music Conference*. San Francisco: The Computer Music Association, 404–442.

Roland, Perry. 2001. "MDL and musicat: an XML approach to musical data and metadata." In *The virtual score: representation, retrieval, restoration*, edited by Walter B. Hewlett e Eleanor Selfridge-Field, Volume 12 of *Computing in Musicology*, 125–134. Massachusetts: The MIT Press.

Sarkar, Soumen, e Craig Cleaveland. 2001. Code generation using XML based document transformation. Available at `http://www.theserverside.com/resources/articles/XMLCodeGen/xmltransform%.pdf`.

Scaletti, C. 1989. "The Kyma/Platypus computer music workstation." *Computer Music Journal* 13 (2): 23–38.

Scheirer, Eric. 2000. "Re: SML (Synthesis Modelling Language)." Mailing list 19 Janeiro. saol-dev@media.mit.edu: MPEG-4 Structured Audio Mailing List.

Schiettecatte, Bert. 2000a. *A format for visual orchestras: flowML*. Disponível em `http://wendy.vub.ac.be/~bschiett/saol/FlowML.pdf` (2003-12-21).

———. 2000b. "SML (Synthesis Modelling Language)." Mailing list 19 Janeiro. saol-dev@media.mit.edu: MPEG-4 Structured Audio Mailing List.

Schottstaedt, Bill. 1983. "Pla: a composer's idea of a language." *Computer Music Journal* 7 (1): 11–20.

———. 1994. "Machine tongues XVII: CLM—Music V meets common lisp." *Computer Music Journal* 18 (2): 30–37.

———. 2002. *Common lisp music*. Disponível em `http://www-ccrma.stanford.edu/software/snd/snd/clm.html` (2002-10-11).

Selfridge-Field, Eleanor. 1993–1994. "The musedata universe: a system of musical information." In *Computing in Musicology*, Volume 9, 9–30. The MIT Press.

———, ed. 1997. *Beyond MIDI: the handbook of musical codes*. Massachusetts: The MIT Press.

Shepard, Toby. 1999, Agosto. *Perlscore*. Disponível em `http://moat.tobiah.org/pub/perlscore/` (2001-04-10).

Shmulevich, I., e D. Povel. 2000a. "Complexity measures of musical rhythms." In *Rhythm perception and production*, edited by P. Desain e W. L. Windsor, 239–244. Lisse: Swets and Zeitlinger.

———. 2000b. "Measures of temporal complexity." *Journal of New Music Research* 29 (1): 61–69.

Smith, Chad. 2000. *[incr Tcl/Tk] from the Ground Up*. Berkeley: Osborne/McGraw-Hill.

Smith, J. O. 1991. "Viewpoints on the history of digital synthesis." *Proceedings of the 1991 International Computer Music Conference*. San Francisco: The Computer Music Association, 1–10.

Smith, Leland. 1981. "The 'SCORE' program for musical input to computers." Edited by Hubert S. Howe, *Proceedings of the 1980 International Computer Music Conference*. San Francisco: The Computer Music Association, 226–230.

Stallman, Richard M., e Roland McGrath. 1998. *GNU make: a program for directing recompilation*. Boston, MA: Free Software Foundation.

Timmers, R., e P. Desain. 2000. “Vibrato: the questions and answers from musicians and science.” *Proceedings of the Sixth International Conference on Music Perception and Cognition*. Keele University, Departmentof Psychology, Keele, UK.

Vercoe, Barry. 2001. *The public csound reference manual*. Massachusetts Institute of Technology. Boston. Disponível em `http://www.lakewoodsound.com/csound/` (2001-04-29).

Weinreb, D., e D. Moon. 1981. *Lisp machine manual*. Cambridge, Massachusetts: M.I.T. Artificial Inteligence Laboratory.

Welch, Brent. 1999. *Practical Programming in Tcl and Tk*. 3rd. Prentice Hall.

Whittle, Robin. 2003. “Re: [Csnd] Zak bugs?” Mailing list 7 Julho. csound@lists.bath.ac.uk: Csound Mailing List.

Windsor, W. L., R. Aarts, P. Desain, H. Heijink, e R. Timmers. 2001. “The timing of grace notes in skilled musical performance at different tempi: a preliminary case study.” *Psychology of Music* 29:149–169.

Windsor, W. L., P. Desain, H. Honing, R. Aarts, H. Heijink, e R. Timmers. 2000. “On time: the influence of tempo, structure and style on the timing of grace notes in skilled musical performance.” In *Rhythm Perception and Production*, edited by P. Desain e W. L. Windsor, 217–223. Lisse: Swets and Zeitlinger.

Winkler, Paul M. 2000a, Julho. *Pysco*. Disponível em `http://www.slinkp.com/` (2001-04-10).

———. 2000b, Julho. *Smake*. Disponível em `http://www.slinkp.com/` (2001-04-10).

Wrighta, James, Daniel V. Oppenheim, David Jameson Don Pazel, e Robert M. Fuhrer. 1997. “CyberBand: a "hands-on"music composition program.” *Proceedings of the 1997 International Computer Music Conference*. San Francisco: The Computer Music Association, 383–386.

Zerbst, Carsten. 2003. “Processing XML documents with Tcl and tDOM.” *Linux Magazine*.