Diagrammatic construction of Csound instruments

Christopher Ware

Bachelor of Science in Computer Science with Honours
The University of Bath
April 2009

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed:

Diagrammatic construction of Csound instruments

Submitted by: Christopher Ware

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

Abstract

Csound is a powerful music programming language, capable of emulating any commercial synthesizer. However it is also considered difficult for musicians without programming experience to use. Here, we specify and implement a graphical front end enabling instruments to be constructed as diagrams. Usable Csound code can then be generated from these diagrams. We also lay the foundations for the reverse process: generation of diagrams from existing code.

Contents

List of Figures

List of Tables

Acknowledgements

I am grateful to Professor John Fitch for supervising this project despite the fact he likes neither GUIs nor Java. Also to my parents for making my life at home as comfortable as possible while writing this, and for help with proofreading.

1 Introduction

1.1 Background

Csound is a music programming language. It allows the implementation of synthesizers and digital signal processors in software, and the performance of musical pieces using them. It is very powerful and sufficiently flexible that given the correct programming it can model almost any acoustic instrument, commercial synthesizer or effects processor available today. Csound was originally developed by Barry Vercoe at the M.I.T. Media Laboratory and development is now led by John Fitch.

In Csound terminology the individual synthetic sound generators one implements are known as **instruments**. A collection of instruments used together to play a piece is called the **orchestra** and resides in a file. Instructions to play sounds can be given in real time via MIDI (Musical Instrument Digital Interface) or specified in the Csound language in a separate **score** file. In this way creation of instruments and the score are independent. The system accepts these source files and compiles (or technically "**renders**") them to produce audio, using the orchestra to play the score. (1)

1.2 Problem Description

Csound is considered by some to be a difficult language to learn. Because of this, there are several graphical front ends available. At their most powerful, these allow interactive arrangement of pieces and assignment of instruments to parts, as would be expected in commercial DAW (digital audio workstation) software. (2)

It would appear from the above statement that the problem of building a GUI (Graphical User Interface) for Csound has already been solved. However, current GUIs are still lacking somewhat in their ability to construct new instruments from component parts (**opcodes** in Csound terminology) such as oscillators, modulators, filters etc. Instrument definitions must still be manually created in a text editor, or for those uncomfortable with that, premade instruments are available online.

Traditional hardware-based modular synthesizers define sounds through the "patching" together of components with cables. Each component either generates sound or has some parameters which affect a signal as it passes through. By connecting together multiple components in a chain, the musician can create more acoustically interesting sounds than just simple waveforms.

Hardware modular synthesizers still exist today, but there is now also a proliferation of emulation software. Software versions of specific hardware synthesizers (such as the Korg MS-20 – Figure 1) exist, and also generic synthesis programs such as *Pure Data* or its

predecessor *Max* (3) based on the modular architecture. These GUIs allow drag and drop placement and connection of components.

Figure 1: Software emulator for a Korg MS-20 analogue synthesizer, with virtual interactive patching

Clearly there is a parallel between the Csound instrument architecture and modular synthesis. Unfortunately, though, no graphical tool similar to those described exists to build and write out the code for Csound instruments.

The main aim of this project is to implement such a GUI allowing diagrammatic construction of Csound instruments, supporting both input and output of Csound code. This will increase Csound's appeal to musicians who lack a programming background.

2 Literature Survey

2.1 Introduction

Simply put, the problem addressed by this project is that of the creation of a graphical user interface for the construction of instruments in Csound. Such a problem can be approached from two main directions or viewpoints:

1. Conversion of Csound code (as may be written by a Csound programmer) into a graphical representation *and back*, without loss of semantics.

2. Interactive construction of instruments in a way that is intuitive to musicians (who are perhaps familiar with traditional hardware-based modular synthesizers) – i.e. the use of suitable metaphors in the UI (User Interface).

Clearly an ideal program will address both viewpoints and so result in their convergence into an effective software product. We therefore intend to first review current conventions and standards in the coding of Csound orchestras, followed by established layouts for hardware-based modular synthesizers (particularly how these are drawn in diagrams). In the latter section we will also cover commercial software emulations of such synthesizers.

We will then move on to the centrepiece of the survey: the analysis of existing Csound GUIs and their shortcomings. This will serve firstly to confirm the fact that this project is indeed covering new ground (i.e. that no similar GUI exists) and secondly to highlight strengths and weaknesses of existing offerings.

Finally, the practical side of the project will be addressed by reviewing the Csound orchestra language from a parsing and compilation perspective, before reviewing tools for GUI programming.

2.2 Csound Orchestra Design

The Csound *orchestra* language is used to describe and specify the virtual instruments that will be used to play a piece (i.e. how they will sound). It is distinct from the *score* language which sequences the notes to be played, performing rather the same role as traditional sheet music (1 pp. 6-8). This project will focus primarily on the former.

In terms of structure and appearance, the language does not resemble C (despite the name – which comes from the fact that it is *written* in C). Vercoe, the originator, likens it more to assembler and some macro languages (4). For those familiar with the appearance of assembly language and aware of the complexity of most Csound instruments, this will likely raise the question: *how can instruments best be structured in the language; what are the best*

practices for maintainable code? Whatever the answer, if we are to produce a program which writes out Csound code, it would be as well for such code to follow these established conventions and practices.

From our review it in fact transpires that there are no formally documented conventions for coding style; therefore we have conducted a study of the structure of several sample orchestra files from *The Csound Book* (5) and present the findings here.

The first of these is that orchestra files appear to be formatted into "columns" using spaces or tabs, with each line taking the form:

```
[Result Variable] [Opcode] [Comma Separated Parameters]
[Comments if any]
```

For the purposes of formatting, the = operator used to assign to variables is treated like an opcode. Any arithmetic expressions appear inline in the parameters list as necessary.

In some examples, the `instr` and `endin` keywords are aligned with the opcodes column. While this is perhaps tidier it makes more sense (and many orchestra files agree) to left align these keywords and inset the intervening lines, rather like the curly braces for function definitions in the K&R C programming style (6). This would make it easier to spot distinct instruments in a large orchestra file.

As noted above, inline comments tend to appear at the end of each row, in a separate "column"; but there are also block comments (several lines dedicated to comments) which can describe the instruments or the file as a whole. These are important because we must ensure they are not lost when a file is modified or displayed diagrammatically. Also, some composers prefer to comment on lines on the line above as opposed to in the last column, which will possibly complicate "comment associations" in our system.

Globals (such as `sr`, `kr` etc.) are not associated with a particular instrument and generally appear at the top of the orchestra file (or *CsInstruments section* in Csound unified files[1]).

2.3 Interface/Drawing Conventions

If we are to design a new interface to Csound that resembles traditional hardware synthesizers, it would be sensible to examine some of the conventions in their layout that have become established over their history. This is especially important because if our tool is to be able to convert Csound orchestra code into a graphical representation, we will need

[1] A new file format introduced in Csound 5 that incorporates both orchestra and score in clearly defined sections. Usually these files have the extension *.csd* and are referred to as

to decide how to lay out the components in the most effective way (since Csound makes no provision for storing layout information itself[2]).

A related line of enquiry which we will also explore is how the designs for such synthesizers (and indeed Csound instruments) are *drawn* and notated, and whether there are any recognised practices there.

2.3.1 Hardware Synthesizers & Software Emulation

Richard Boulanger states in his *Introduction to Sound Design in Csound* (1) that Csound draws from a toolkit of over 450 signal processing *modules*. This architecture makes its closest hardware equivalent a class of synthesizers known as **modular synthesizers**, and it is these we will therefore examine. The following informal definition of the term further supports this conclusion (although the phrasing is a little misleading – an infinite number of configurations is only possible with an infinite number of modules!):

> "The modular synthesizer is a type of synthesizer consisting of separate specialized modules connected by wires (patch cords) to create a so-called patch. Every output generates a signal - an electric voltage of variable strength. Combining the signals generated by multiple modules into a common audio output allows a potentially infinite number of configurations, leading to a potentially infinite number of sounds."(7)

The intention in the review was to examine trends in the layout of hardware modular synthesizers, however there is a distinct lack of literature surrounding that particular topic. This suggests that there are no significant conventions and that it is usually left to personal preference on the part of the owner/player as to how the modules will be arranged (since they are generally removable). It is therefore intended that our designs for a new GUI will reflect this and allow arbitrary placement of opcode/module symbols, taking advantage of the fact that no physical work is needed to rearrange software modules!

When we direct our attention to software emulations of modular synthesizers, however, we discover that there have indeed been studies of popular layouts here. Noble and Biddle (8) conducted one such study, investigating the *Nord Modular Patch Language*. This is used to visually program the Nord Modular system (produced by the Swedish company Clavia AB in 1998) using a PC. An example "patch" from such a system is given in Figure 2.

[2] Indeed it should not as a purely audio processing language, so a possible task may be to implement (or reuse) a language on top of Csound which offers layout capabilities

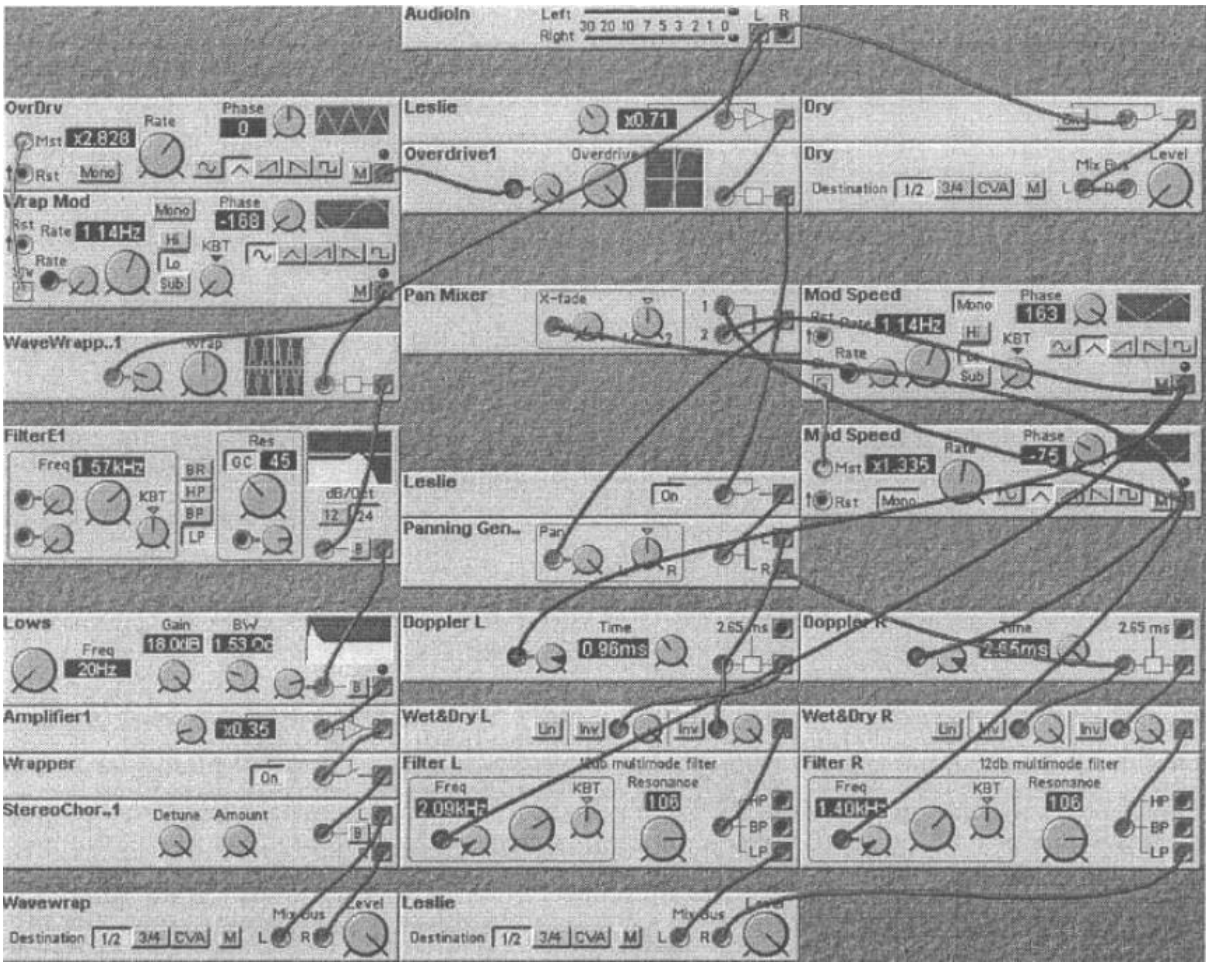


Figure 2: Nord Modular Patch Language – movable modules are represented by labelled rectangular boxes and patches are represented by the arcs between virtual "sockets" on the modules [Sourced from (8)]

Noble and Biddle quantitatively analysed 1051 preset patches to discover what they term "stereotypical layouts" of modules and concluded the following points which are relevant to this project:

1. The most popular module type in terms of the number of occurrences in patches seen across the study is the **oscillator**, followed by envelope modules and low frequency oscillators (LFOs). This may influence our choice of ordering/categorisation of any symbol libraries implemented.

2. Commonly used modules (that is featuring in the widest range of patches) were the ADSR (Attack-Decay-Sustain-Release) envelope, mixer, and of course inputs and outputs

3. The following module location trends:

a. Inputs usually appear at the top of the screen

b. Oscillators and LFOs appear primarily to the left of the screen (presumably near the inputs)

c. Filters usually appear in the centre column

d. Enveloping modules are usually to the top right

e. Outputs are usually in the right hand column towards the bottom

4. Modular programmers appear to prefer to scroll vertically rather than horizontally (and underuse the right hand side of the screen in their patches).

The number of patches examined in the study suggests that this is a good indication of popular layouts among this particular user base. Users of a Csound graphical patching tool would likely share similar preferences and so these are valuable points to consider in the design of such a tool.

The study technique used to analyse the 1051 patches is interesting and would appear to be also useful for analysing trends in any graphical Csound patch layouts. However, unlike the Nord language, Csound does not yet have a widely accepted standard format for *graphical* layouts– indeed this is part of the rationale for this project. Such a technique is therefore of no use to us, since we lack the "corpus" of patches/instruments.

2.3.2 Block Diagrams

The alternative to a full graphical simulation of a hardware device is diagram drawing. It would appear from reading various Csound related documentation that there already exists a diagrammatic convention for representing Csound instrument designs. Although there is apparently no formal specification, this common style is used in examples throughout *The Csound Book* (5), tutorials on the official website (9), and the *Amsterdam Catalog of Csound Computer Instruments* (10) – a widely recognised collection of Csound instruments. It has in a sense become the *de facto* standard, and so it would be sensible to consider this for our graphical representation.

Because of its status as the de facto standard, the Csound flowchart symbols have received some coverage on electronic music courses at various institutions. Figure 3 has been extracted from course materials in use at the University of Florida (11) and summarises some common Csound diagram symbols.

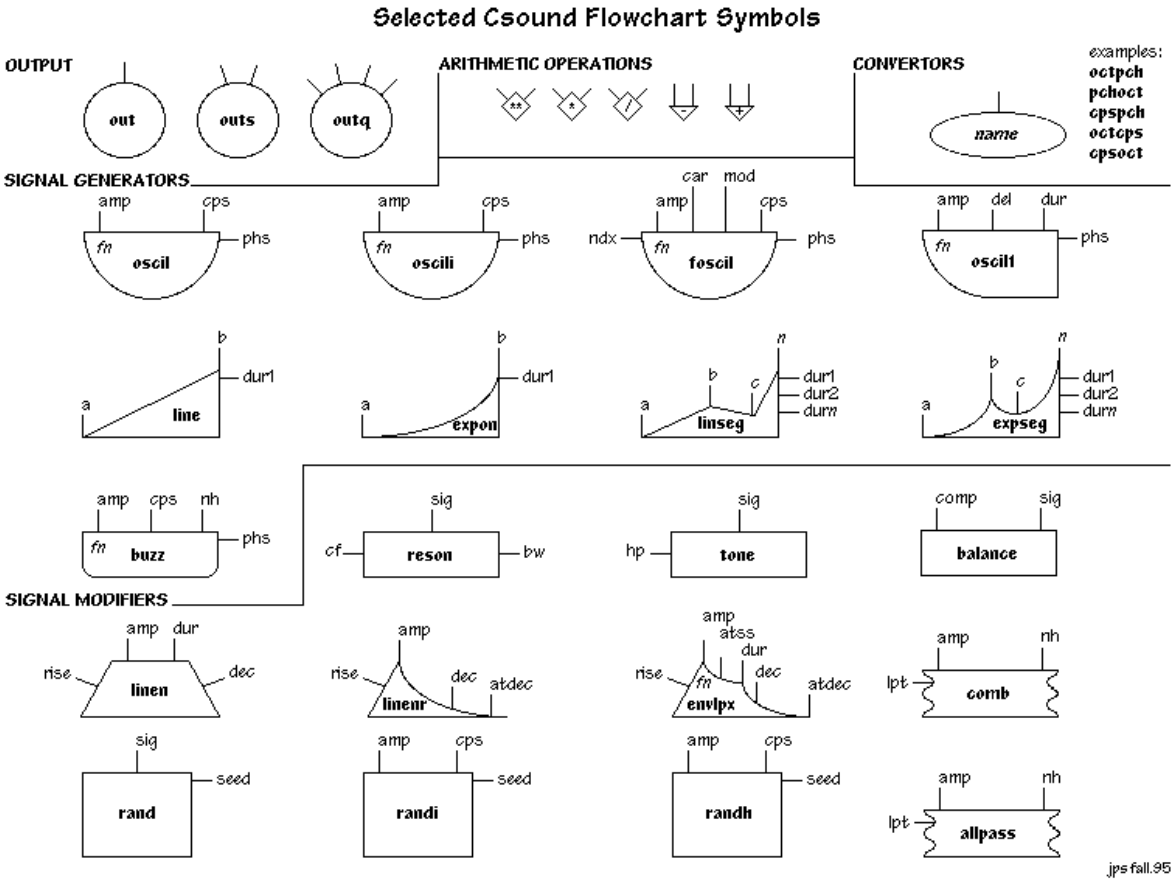


Figure 3: De facto standard Csound diagramming symbols (11)

Note that the lines protruding from the symbols indicate parameters to the opcode as opposed to points for "patch cord" connection. We can see in example diagrams such as Figure 4 how these connections might be made. We also note that there are some shorthand notations such as the sine-wave symbol which presumably means an oscillator (and the related orchestra file indeed confirms that it is `oscili`) or the use of a thick-bordered circle to mean output. Also, not all formal parameters specified in Figure 3 need be used. The latter is simply the concept of optional parameters (12) in action and is not specific to diagramming.

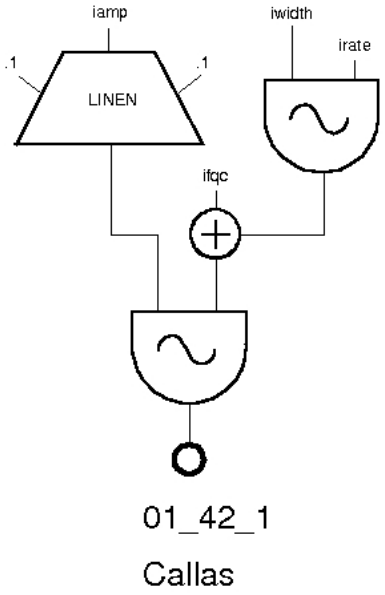


Figure 4: Example diagram from (10), drawn using Csound flowchart symbols

Clearly there are some design decisions to be made around details such as connection points, optional parameters and "shorthand" opcode representations.

It is also interesting to read the orchestra file associated with Figure 4:

```
instr 1
    idur   =  p3
    iamp   =  p4
    ifqc   =  p5
    ifc    =  p6
    iwidth =  p7
    irate  =  p8
    ifm    =  p9

    amod   oscili   iwidth, irate, ifm      ; LFO modulator
    amod   =        ifqc + amod
    aenv   linen    iamp, .1, idur, .1      ; prevent clicks
    a1     oscili   aenv, amod, ifc         ; carrier waveform
           out      a1
endin
```

Immediately we can see that the various `pX` parameters input from the score do not feature on the diagram. They are instead assigned to variables with more meaningful names and *some* of these are notated on the diagram. We would need a way to diagram these if diagrams are to be used to construct instruments from scratch. Also note that the four parameters to `linen` are not all notated on the diagram – an oversight, perhaps? Certainly

the template symbol given in Figure 3 makes provision for them all and the manual confirms they are all required. If we are to produce a system based on this that is truly bidirectional (in the sense that is can convert from Csound to diagram and back), it will not be allowable to lose information in this way.

Similarly the oscillators seem to be missing a third parameter on the diagram – only one of the four parameters to `oscili` is optional according to the manual. Closer inspection reveals that Figure 3 notates the parameter `fn` *inside* the shape – another anomaly. Perhaps the sine wave symbol in Figure 4 is in fact shorthand for `fn` = some sinusoid function.

In conclusion there is strong evidence of an established diagram format for Csound instruments. However this is not sufficiently strict or formal to be able to transfer directly to software without adding some constraints or specifying certain aspects strictly. Our review of current GUI implementations in the following section may reveal existing approaches to dealing with this problem that can be reused, however.

2.4 Current GUI Implementations

As may be expected with a language that is generally considered to be difficult to learn, there have already been many attempts at implementing a GUI for Csound. Some of these are actively maintained, some not; some are directly related to the subject of this project, some are not; but in all cases we can learn something by reviewing them.

We will adopt a systematic approach to evaluation to ensure that we cover each in sufficient detail, but avoid spending an unnecessary amount of time on programs which are not directly relevant. Specifically, for each GUI application, we will review the following points relevant to this project:

- Editing support for orchestra files/sections
- Generation of orchestra code from diagrams (and whether real-time or explicit conversion)
- Generation of diagrams from orchestra code (and whether real-time or explicit conversion)
- On diagrams (if used)
    - Shape conventions
    - Setting of opcode parameters in the diagram
    - Layout and ease of use of the shape “library” and connectors
- GUI manipulation of parameters (e.g. using knobs, sliders etc.)

- Auditioning of instruments (real-time or otherwise)
- Handling of multiple instruments in the orchestra file
- Current activity on the development of the tool
- Csound 5 compatibility
- Platform compatibility/programming language

2.4.1 Winsound

Winsound is a basic GUI included with the Csound distribution. It appears to be designed primarily as a graphical interface to command line parameters (i.e. a "launcher"), rather than providing an interface for authoring scores and more importantly orchestras. It provides several additional utilities such as various analyses and file information extraction, but contains nothing particularly novel that would be beneficial for reuse in this project.

It is written in C and uses Fast Light Toolkit (FLTK), and as the name suggests it is intended to run mainly on Windows, although according to the Csound website (13) it can run on Linux under WINE. There are portions of the source code suggesting Mac OSX compatibility (i.e. detection of Core Audio).

Winsound is Csound 5 compatible but this appears to be an afterthought, at least in terms of the UI presentation.

2.4.2 Csound5GUI

Csound5GUI is very similar to Winsound in that it is included with the official Csound distribution and is essentially a graphical command line launcher. However the overall appearance is generally more sophisticated and it has obviously been designed to take advantage of Csound 5 features such as the new programming interface and CSD files. It allows, for example, simple real-time seeking through the score during rendering of the piece.

Csound5GUI can also launch external user specified programs to allow editing of the rendered sound file or to play it.

Our project is not as concerned with production of finished sound files so much as it is with production of correct orchestra files, but it is possible that our program could allow quick auditioning of the orchestra against a user specified score, and such functionality might use a similar interface to this one.

Csound5GUI displays the console window by default to allow viewing of text output from the renderer and has a useful feature in that it highlights output errors in red.

Like Winsound, Csound5GUI also uses FLTK.

2.4.3 CsoundX

CsoundX is a front-end for Apple Mac OSX only. Access to this operating system was not available, but quoting from a news post on the official Csound website (14) about its release:

> "Things you can do with CsoundX:
>
> - Render simultaneous multiple csd/orc/sco in real-time or to disk.
> - Use a generic GUI Control panel for real-time control that you can replace or modify with your own interface using Apple's Interface Builder application [see README]
>
> Things you can't do with CsoundX:
>
> - Edit [save] your csd/orc/sco files
> - FLTK [hangs CsoundX]"

This therefore appears to be a similar tool to Csound5GUI and does not offer the graphical instrument editing functionality we are looking for.

2.4.4 Blue

Blue is a large, complex and powerful frontend for Csound written in Java. It is referred to as a "music composition environment" (15) and is similar to commercial Digital Audio Workstation (DAW) packages such as Steinberg's Cubase or Cakewalk Sonar. As such, we will not be examining every feature in depth, but will focus on those related to the orchestra.

It is, however, worth mentioning that "Blue interacts with Csound by generating CSD files, which it then feeds to Csound for compilation" (15). Code generation for CSD/ORC files is something we are intending to implement and so there may be reusable code in Blue since it is open source, assuming we choose Java as the implementation language. Similarly, Blue can also import CSD files to presumably some kind of internal representation, suggesting that it is able to parse the language – more potentially reusable functionality.

Blue separates its interface with tabs, and most work with orchestras and instruments is unsurprisingly carried out on the **orchestra** tab. The left hand side of takes the form of a "librarian" style interface where instruments can be organised, numbered, named, selected for editing and stored for later use etc. We will likely need a similar interface in our diagramming tool, since orchestras generally contain more than one instrument and it is unlikely the user would want them all to be simultaneously onscreen for editing.

The right hand side of the orchestra tab is altogether more interesting – it is where editing of the instrument takes place. Blue defines several different *types* of instrument on top of the basic Csound format. The most interesting to us are the *GenericInstrument*,

BlueSynthBuilder and *BlueX7*, all creatable using the library and each presenting a different editing interface.

GenericInstrument has the simplest editing interface. It is a simply a way to build traditional Csound instruments inside Csound, by just writing out the code in a text area. The code is syntax-highlighted and since Blue manages the instrument naming and numbering, the `instr` and `endin` keywords are unnecessary when editing in this way.

BlueSynthBuilder is similar to the kind of interactive GUI design environment found in Microsoft Visual Studio or Qt Designer, except that the controls are tailored to synthesizer (rather than desktop application) use. It allows the user to lay out knobs, sliders and similar controls and then refer to these as parameters from inside an ordinary Csound text-instrument. The parameters on the instrument can then be adjusted using the graphical view rather than by changing values in the code – a more intuitive approach. This allows the construction of the kind of interface previously seen on the modules in Figure 2 (although note that it does **not** model "patching" of opcodes – the "control panel" is that of the instrument as a whole, not the individual module).

It is worth taking a moment here to examine a comment made by Yi in the manual (15) just before discussing the BlueSynthBuilder.

> "Modular instruments are easier to express connections of modules via text or code rather than visual paradigms (patch cables, line connections), and thus easier to create the instrument by text. Graphical elements, however, excel in relaying information about the configuration of the instrument to the user and also invite experimentation, while text-based configuration of instruments is often more difficult to quickly understand the parameters settings and limits.
>
> Going completely graphical for the building of instruments, in the case of systems like Max/PD/jMax or Reaktor, I've found that the instrument's design no longer become apparent when viewing complicated patches. On the other hand, using completely textual systems such as Csound or C++ coding, the design of the instrument has a degree of transparency, while the configuration of the parameters of the instrument becomes difficult to understand and invites less exploration."

We are inclined to disagree in part with this. If Csound instruments were always easier to express and understand in text, then the diagramming conventions outlined in 2.3.2 would simply not exist, and catalogues of Csound instruments such as (10) would not include diagrams. Therefore *some* users at least find diagrams to be useful and there is value in implementing a diagrammatic means of building or editing instruments. However, a valid point is raised in that access to the instrument code should also be made available, and that has indeed been the intention so far with this project.

Presenting an interactive graphical interface to the parameters of an instrument is an interesting and useful idea, which is worth pursuing as a secondary goal to the visual patching together of modules if time permits. Rather than make controls instrument-scoped as Blue does, it would be more natural for this project to assign controls to each *module* and visually group them with that module on screen, again much like the interface seen earlier in Figure 2.

BlueX7 is a ready-made instrument based on Russell Pinkston's Yamaha DX7 emulation in Csound. Its parameters are highly configurable but does not offer any new insights into graphical instrument creation so we will not investigate in any further detail. Possibly our implementation could ship with a similar instrument included as an example, but time constraints are unlikely to permit this.

Instruments in blue can be applied to existing scores, or played in real-time using MIDI (on a sufficiently powerful computer) using "blue Live". However, unlike some commercial emulations it does not have an audition mode where preset phrases can be played to give a quick preview of the sound.

Blue is actively maintained and fully compatible with Csound 5. Because it is written in Java, it will theoretically run on any platform for which there is a Java VM available. As would be expected from such a complex program, blue maintains its own file format which stores the graphical instrument interfaces and this cannot be read directly by Csound.

2.4.5 Cseditor

Cseditor is a simple syntax-highlighted text editor for Csound that is supplied with the default distribution. It can read and write ORC, SCO and CSD files but has no graphical capabilities so we will not investigate any further.

2.4.6 FLTK Widgets and GUI Controllers

According to the Csound manual (12) FLTK Widgets allow the design of a custom Graphical User Interface to control an orchestra in real-time from within Csound. However, it is not a GUI frontend as such and does not allow "patching together" of modules. Its primary use is the control of instrument parameters and as such we are not especially interested in it with respect to this project.

2.4.7 Patchwork

Patchwork (16) is a flowchart-based instrument design program for Windows 3.1 (although there has been an X-Windows version developed called xPatchwork). Symbols represent opcodes and have connection points representing inputs and outputs. They are patched together with connecting lines.

It can export the diagrams drawn using it to produce Csound orchestra files but is not able to import existing Csound files to create diagrams. Linked to this, it does not support direct

editing of the score file that is under construction, instead making production of the score file an explicit compilation step.

There is no real-time auditioning of instruments, although there is an easily reachable "run" button which will play a specified score with the current orchestra file, which serves a similar purpose.

The interface has a large library of Csound opcodes (over 140) which is fairly complete according to (16). These use symbols similar to those already discussed in 2.3.2. Parameters for the opcodes are set by "patching" simple values to connection points on the shapes; for example, in Figure 5 a frequency is set by simply attaching the text 1000 to one of the terminals. A particularly irritating feature of the interface is that it would appear that symbols cannot be moved without severing the connection to the patch cable – this goes against the intuition of modern flowcharting software.

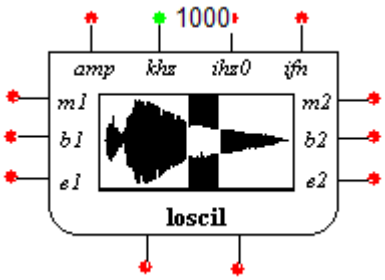


Figure 5: Setting a simple numerical value for a parameter on loscil in Patchwork

Multiple instruments in an orchestra are handled interestingly. There are symbols available from the library for `instr` and `endin` and it appears that physically locating these on the diagram where the equivalent keywords would appear in the code (i.e. above and below the symbols to be contained in the instrument) is how one groups components to form an instrument.

Returning to orchestra compilation, it appears that no checking is performed on the diagram before the code is generated. It is possible to omit required parameters or the `endin` symbol and these errors will not be detected until the file is read with Csound.

Patchwork is no longer updated and has not been for some time – it therefore does not support the new features of Csound 5. The library of devices is stored in a proprietary binary format which is not easily reverse engineered, so extensions of Patchwork with new opcodes are unlikely to be viable to anyone other than the original author.

2.4.8 Visual Orchestra

Visual Orchestra (17) is a graphical design environment based around Csound. It is a commercial product for Windows but a demo version is available. The latest version is

version 2.0 which was released in 1999 – this suggests that it is probably not Csound5-aware, nor actively maintained. In fact Visual Orchestra installs its own copy of Csound.

Visual Orchestra works as a Multiple Document Interface (MDI) application and each instrument is a separate child window inside. The instruments are shown in a tree to the right (Figure 6), and selecting one toggles to the editing window. The aforementioned tree includes the opcodes in use as child nodes, and allows modification of their parameters (in plain text only), which are represented as further child nodes.

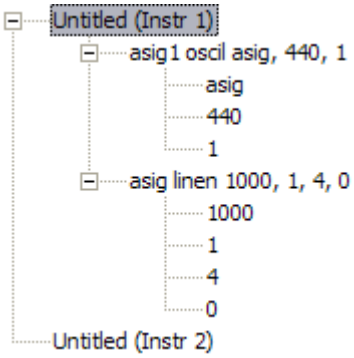


Figure 6: Instrument tree in Visual Orchestra

In addition to the tree, there is also a main editing area where Patchwork-like editing can take place. Updating the tree updates the diagram and vice versa. Unlike Patchwork, the connections and drag and drop mechanism is intuitive and does not break when the components are moved around the screen. However, also unlike Patchwork, the traditional Csound diagram symbols are nowhere to be seen – instead the opcodes are all represented by boxes. Output variables from the various opcodes are clearly displayed and are user definable. Opcodes are selected from a menu where they have been arranged in various categories to speed up finding them.

There is no support for importing an existing Csound orchestra for editing, only the generation of a new one from a representation built in Visual Orchestra.

Despite the name, Visual Orchestra appears to be able to edit scores too, albeit only in a note list format. It also supports real-time MIDI playing of the instruments.

2.4.9 Cabel

Cabel (18) is another interface for building Csound instruments by patching modules similar to modular synthesizers. Cabel does not appear to work directly with the Csound primitive opcodes but redefines them into their own user-defined opcodes for which they present the graphical components. These user defined codes are named more similarly to the components you would find on a hardware analogue synthesizer. They are accessible for insertion from the menu and grouped into categories. All units are rectangular, and do not use the Csound diagram convention discussed previously.

The patching mechanism is very nice, with simple click and drag from socket to socket being sufficient to create a connection. The sockets are colour coded for the expected type of signal and there are tooltips that explain the relevant parameter.

It appears to be the intention that Cabel shall be able to export CSD files but this menu option appeared non-functional in testing so this may still be planned functionality. Cabel cannot import from Csound. There was an option to start Csound from within Cabel but this didn't appear to actually start it with any particular options. As such, Cabel is not actually functional for Csound code generation or parsing at this stage.

Cabel is written in Python and so is technically cross-platform. However, the Windows version at least has many dependencies on UI libraries.

2.4.10 WinXound.Net

WinXound.Net is simply an editor for Csound files written in .NET, with syntax highlighting and other features rather like Cseditor. It does not have graphical capabilities but does give dynamic documentation for the opcode under the cursor which is a useful feature for any program that works with the often cryptic Csound opcodes. Because it is a simple editor it can read and write orchestra, score, and CSD files, but we will not examine it any further because it has no diagramming features.

However, one interesting point worth mentioning is that WinXound.Net comes with a large CSV file containing a listing of all opcodes and their descriptions. We may be able to reuse this (especially since the license permits) in any software of our own creation.

2.4.11 QuteCsound

QuteCsound is a recently developed frontend for Csound written using the Qt GUI library (19). It is unique from the other diagram-capable programs reviewed in that it can actually generate a "code graph" based on a given orchestra file (an example output is shown in Figure 7). This is achieved using the GraphViz graph drawing tool which we review separately later.

Parsing the Csound source code to generate a graph in this way is certainly a major step forward compared to other tools. However using GraphViz means there is no interactivity in the graph – it is just a simple bitmap. This makes it impossible to edit the graph representation and regenerate Csound code in QuteCsound, and therefore it does not completely overlap the aims of this project

We can, though, use the graph layout produced by QuteCsound to influence the format of the diagrams produced by our software. A brief analysis of Figure 7 reveals that round cornered boxes are used to represent opcodes, with the name in the centre, input parameters at the top, and output parameters at the bottom. Parameters on the opcode are shown using their formal names. Actual variables used to store outputs are shown by labels on the graph edges.

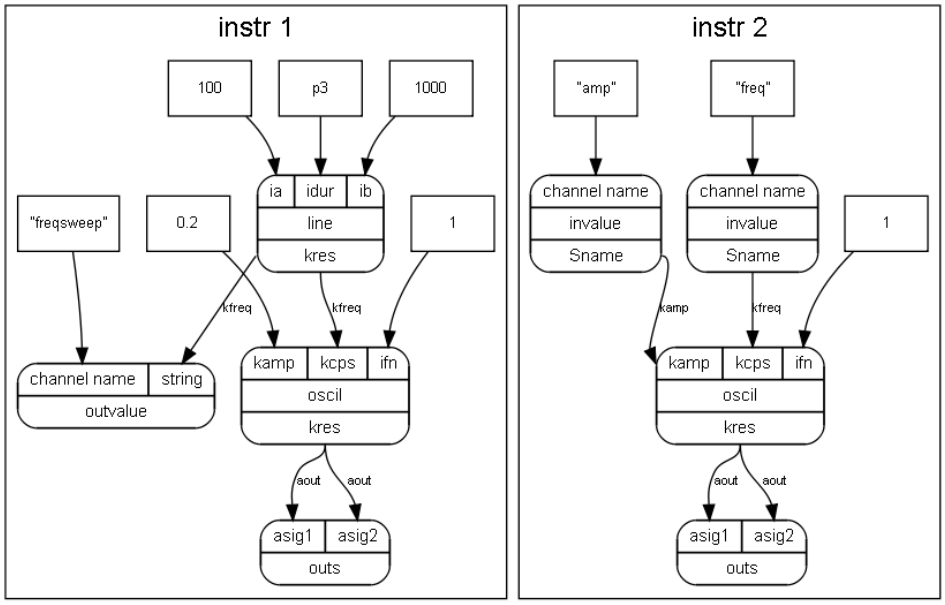


Figure 7: QuteCsound code graph

Note: *This section has been added retrospectively to the literature review because QuteCsound is currently under active development and was not available when this section was originally written or the designs produced for this project. Further discussion in hindsight is given in section 6.3.4.*

2.5 Csound Language

We have already remarked that the Csound language resembles assembly language, but how does one go about parsing and generating it? Specifying this in depth is likely a topic for consideration the system design but here we will briefly review the language structure and then investigate a related language.

2.5.1 Language Structure

The syntax of a typical Csound orchestra file is actually fairly simple (and this should make it equally simple to parse or generate), due to its assembler-type format. It is described informally in most introductory texts such as (20) but there seems to be no formally defined grammar (for example in BNF) available.

To summarise, the only 2D structure as such is the *instrument*, enclosed by `instr x` and `endin`. The remainder of the instructions are the "`variable opcode parameters`" tuples we have seen before. There are also of course the comments prefixed by `;`, and also *labels* which are not generally seen in practical use. The orchestra header contains some assignments to reserved variables.

In terms of semantics, the most significant checking is that of variable types, since there are certain naming conventions that have a bearing on how variables may be used. Table 1, based on a similar one from Pinkston's primer (20) and the Csound manual (12) provides a summary.

Prefix (scope)		Example	Description
Local	Global		
a	ga	asig	Value updated at audio rate, generally used to contain an actual audio signal
k	gk	kmod	Value updated at control rate, generally used to contain a control signal (for example modulation or envelope)
i	gi	iamp	Value set at initialisation time
p	-	p2	Parameter passed from the score – should not be set in the orchestra
x (formal only)	-	xcps	Formal parameter for an opcode, specifying that arguments may be of varying type

Table 1: Csound variable prefixes

It is worth noting that because we will only be parsing Csound for conversion to a graphical representation, the semantics are not as crucial as they are in the real Csound compiler. However, we must still ensure they are retained and can be accurately transformed from our representation back to Csound in order to allow lossless round trip conversion of Csound orchestras. In particular if we will allow the user to patch the output of one opcode to another in our graphical tool, we may wish to ensure that such a variable assignment is actually allowed.

The Csound Reference Manual (12) contains an "opcode quick reference" which specifies the argument type for every opcode. We could possibly import this and parse it to determine the correct type for every opcode argument without manually entering them.

2.5.2 CsoundXML

Worth mentioning in this section is CsoundXML (21), a meta-language for Csound designed by Pedro Kröger and based on XML. According to Kröger it is intended to "describe the Csound orchestra language with a few added features." This new language was designed to elegantly solve several problems with the Music V derived series of languages, the most important of which are:

1. **Instrument reuse** – generally instruments are numbered not named, and they do not have context-dependent sound output (i.e. flexibility between routing output to the DAC or to another instrument/signal processor);

2. **Parameters as an ordered list** – difficult for the user to recall the order and function of all parameters for an opcode or instrument;

3. **Lack of graphical scalability** – tools used to describe instruments graphically need to have a deep understanding of the language syntax and often implement another full parser for the language. Csound actually has opcodes for graphical widgets which mixes graphics functionality and synthesis in the same code, reducing maintainability;

4. **Lack of score-orchestra integration** – specifically the lack of integration between any pre-processors for the score and the orchestra. Tools for score processing usually define musical representation in a higher level than the flat note list. However this breaks the communication between the pre-score (the file to be processed and converted into the score) and the orchestra.

Clearly point 3 is a crucial issue in this project, and hence CsoundXML is worthy of further investigation if it promises to solve this problem – the other points also have relevance.

Unfortunately CsoundXML does not appear to have gained widespread acceptance in the 4 years since its conception: web search results for "CsoundXML" are mainly for (21) itself – no actual implementations are apparent. In any case, though, we are likely to need an intermediate representation, and even if we need to write our own parser and compiler, it is still valuable for this area to have already been investigated in research. Of course in this latter case we are still not starting from scratch – there are many XML parsers available.

In discussing CsoundXML's application to graphical tools, two problems are highlighted:

1. Design decisions to define how elements will be drawn. Sound generators such as oscillators are easy to represent while opcodes that convert values, and flow control, are hard to represent graphically.

2. Producing algorithms to distribute the synthesis elements on the screen avoiding overlap of graphical elements.

These are key issues that this project will need to overcome. We have already addressed a little of the second point in our review of module placement earlier, but the arrangement algorithms are an issue still to be investigated.

Returning to the essence of CsoundXML; information about the opcodes (e.g. their formal parameters) and parameters themselves (e.g. the possible values) is defined in an XML

library for Csound called CXL (22), which was also developed by the author of the CsoundXML paper. This document is, however, only available in Spanish (possibly the reason that adoption of CsoundXML has not been widespread). This is problematic for us, especially since protection on the electronic copy makes automated translation unreasonably difficult! However the portions containing the Csound opcodes and XML are interpretable since these are typically in English, and use of CXL can hopefully still be made in this project.

An example instrument in CsoundXML (taken from the paper) may appear something like this:

```
<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>
```

This would be representative of the classic Csound example – note that output is not necessarily named identically in the XML: this is part of the idea of making outputs more flexibly routable:

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

CsoundXML also makes provision for defining i-variables, as follows:

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

In conclusion, having an intermediate representation of this kind is of value if we can easily translate it to both a graphical representation and Csound. We will see that there are XML diagramming tools that would be ideally suited to such a transformation in the next section. As to XML-to-Csound translation we may well need to further Kröger's work on CsoundXML and implement such a compiler since none appears to be in existence. Also, a more difficult task, Csound to XML conversion remains unsolved – all we have gained from CsoundXML is a specification for the intermediate language and a justification for it. It seems more likely that we will just reuse certain ideas from this effort in a more specific solution, rather than attempting to implement the general XML scheme proposed here.

2.6 Diagramming Tools

Practically since the invention of graphical computer interfaces, there have been software tools for drawing diagrams. This is an obvious task for a computer. Well known packages include AutoCAD and Visio (now a Microsoft product). In the open source world (and hence more suited to Csound's philosophy) there is a product called Dia, which is in fact modelled after Visio.

2.6.1 Dia

This is part of the GNOME project and has been in development for at least 5 years (23). There have recently been some discussions online (24) about using Dia for Csound instrument design but no action as of yet.

Dia is particularly relevant to our project because it stores its diagram data in XML files, more discussion of which we shall see shortly. If we were able to make use of it, it would eliminate much work that would be needed to implement a drag and drop diagramming application from scratch so it is clearly worthy of further investigation.

2.6.1.1 Shapes

Dia is distributed with a wide range of prebuilt diagramming shapes; for example UML, network diagrams etc. Some of these come with extended property pages allowing, for example, the classes and fields to be formally specified on a UML class.

Perhaps unsurprisingly Csound is not among the included shapes and so we would need to create our own to use this tool. According to the FAQ on (23) there are two ways to do this.

The first is the apparently "easier" method of drawing the shape *in* Dia and then exporting it to a shape file using the ordinary graphics export feature of the program. This disadvantages of this are that it is a manual method, and there appears to be no way to place "connection points" other than those already supplied in the primitives being used. Also there seems to be no way to build the extended property pages seen in some of the supplied shapes.

The second approach is writing the shape files directly in XML (25). The XML files use a subset and an extension of the Scalable Vector Graphics (SVG) format. This seems to be a

more flexible way to generate the shapes and opens up the possibility of programmatic generation (since we have a large number of opcodes to encode as shapes).

However the document referred to still sheds no light on how custom property pages can be created. This will need to be investigated further if we are to provide a means to write parameters directly into shapes and so model the opcodes more intuitively. An alternative to property pages may be to have separate text areas in the shape for the different parameters but this seems untidy. Inspection of the source code for Dia reveals that the extended property pages are actually coded directly in C using the GTK+ UI widgets. Although highly flexible, this complicates matters and is (in the author's opinion) something which should be abstracted into the shape XML in future Dia releases.

2.6.1.2 File Format

Dia diagram files are unsurprisingly also based on XML, utilising the Dia namespace `http://www.lysator.liu.se/~alla/dia`. The format is simple enough to understand. Inside the top level there are elements for the diagram metadata, such as paper size etc. and for each layer. Inside the layers are references to the "objects" or shapes that make up the drawing, including sizing and positioning information, the values of custom parameters, and details of connections to other objects.

2.6.1.3 Python Scripting

Dia supports extensions (26) written in Python, and this capability has been used successfully to allow import and export of SVG for example. It is possible that we could utilise this to allow Csound import and export, if we wrote the parser or code generator in Python.

2.6.2 Microsoft Visio

Visio, a part of Microsoft Office, is Microsoft's diagram drawing package. It is a commercial product with a similar purpose to Dia which we have already described, although it has in general a greater range of features.

Visio allows the linking of shapes to data, mainly intended for database connectivity but this could possibly be made use of for generation of Csound code. Further investigation would be required. Visio can also store its diagrams in XML files which gives the possibility of transforming them as discussed above for Dia.

However, Visio is not freely available which makes it unsuitable for this project where we wish to make available a program with the same philosophy as Csound itself in terms of redistribution and extension.

2.6.3 JGraph

JGraph(27) is a Java based graphing library. It handles the drawing and arrangement of shapes, connections between them, and extraction of data from the graph.

According to the manual (28):

> "JGraph provides a range of graph drawing functionality for client-side or server-side applications. JGraph has a simple, yet powerful API enabling you to visualize, interact with, automatically layout and perform analysis of graphs. The following sections define these terms in more detail.
>
> Example applications for a graph visualization library include; process diagrams, workflow and BPM visualization, flowcharts, traffic or water flow, database and WWW visualization, networks and telecoms displays, mapping applications and GIS, UML diagrams, electronic circuits, VLSI, CAD, financial and social networks, data mining, biochemistry, ecological cycles, entity and cause-effect relationships and organisational charts.
>
> JGraph, through its programming API, provides the means to configure how the graph or network is displayed and the means to associate a context or metadata with those displayed elements."

In the earlier discussion QuteCsound (section 2.4.11) it was seen that the Csound instrument architecture can be treated as a specialised type of graph, and therefore a graph drawing library such as JGraph seems highly appropriate for consideration alongside Dia. As a library rather than a complete application it may prove more flexible in the event that our requirements cannot be satisfied by Dia etc. and a custom program is necessary.

2.6.4 Graphviz

Graphviz (29) is a package for automatic graph drawing given an abstract representation of the graph. It is in fact used by QuteCsound (discussed above in 2.4.11) to generate its code graph output view. However, Graphviz does not allow interactive *editing* of diagrams and is hence will not be a major candidate for use in this project.

2.6.5 Crocodile Clips

While not a general purpose diagramming program, Crocodile Clips (a product for constructing and simulating electrical circuits) provides some UI features that we may draw influence from for this project. In particular the insertion of components is done by selection from a categorised menu (Figure 8) rather than by any more complex means. This is an easily implemented design feature that would lend itself to selection of opcodes for insertion into diagrams.

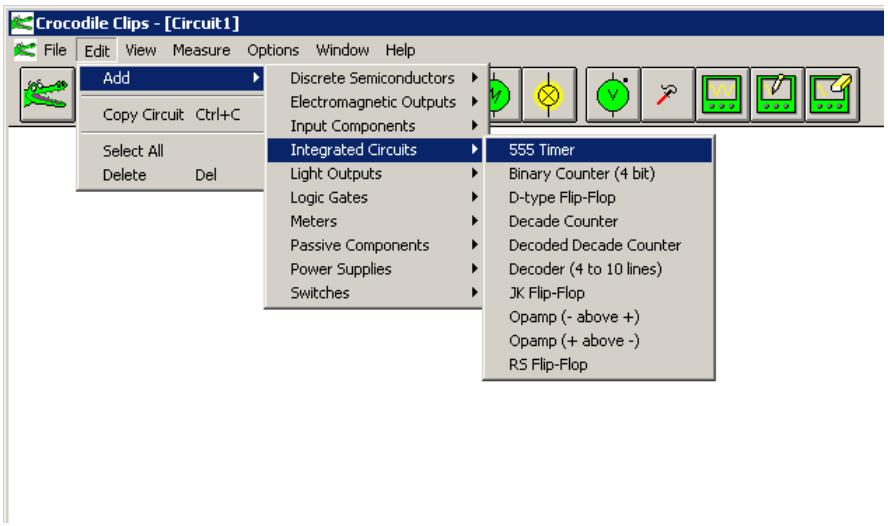


Figure 8: Categorised insert menu in Crocodile Clips

2.7 GUI Libraries

In the case that diagramming tools prove insufficiently flexible for our problem, it may be necessary to write the entire graphical program from scratch. This would require use of a graphics programming API and here we briefly examine some common ones.

2.7.1 Java

The Sun Java API provides the Swing and AWT packages for windowed GUIs. Because Java is cross-platform so are these interfaces, which is advantageous. Java also has the advantage of having XML processing capabilities in other parts of the language. Use of JGraph discussed earlier would involve the use of Java.

2.7.2 .NET

.NET is a Microsoft programming API similar to Java's, except that it is officially Windows only. .NET supports XML manipulation as part of the library functions, and also 2D graphics.

2.7.3 GTK+

GTK+ was originally a toolkit for the X-window system designed for use in programming "The GIMP": a photo editing tool for UNIX. It has since been used to build the GNOME desktop environment and has been ported to Windows and other operating systems. It has bindings for many languages including C (its native language), Java, Perl, Python and PHP.

2.7.4 Qt

Qt is similar to GNOME in that it was originally developed to sit behind KDE, another major UNIX desktop environment. It has since been ported to other operating systems in a similar way to GTK+ and with similar language bindings.

2.7.5 FLTK

We have already seen FLTK in one form as an extension to Csound itself, but we could also use it for a standalone GUI.

Paraphrasing from (30), FLTK is a cross-platform C++ GUI toolkit for UNIX/Linux (X11), Microsoft Windows, and MacOS X. FLTK provides modern GUI functionality without an excessively large code base and supports 3D graphics via OpenGL and its built-in GLUT emulation.

FLTK is designed to be small and modular enough to be statically linked, but also works as a shared library. It includes a UI builder called FLUID that can be used for rapid application development.

2.7.6 OpenGL

OpenGL is a low level graphics library mainly intended for writing 3D applications in C. It is not a windowing toolkit and is probably too low-level for this project so will not be pursued.

2.8 Summary and Conclusions of Literature Survey

By searching the literature we have discovered some of the conventions in Csound orchestra design, and the de facto standard diagramming convention. We also saw a popular layout and format for the instrument source file text. Current GUIs have been investigated, with particular focus on any instrument-editing capabilities – particularly graphical patching. The Csound language has been examined and we reviewed CsoundXML – a language that promised an easy interface to graphical manipulation of instruments if only the relevant parsers and compilers were implemented. Finally, we discussed diagramming tools with particular focus on Dia, and briefly overviewed some graphics toolkits.

The original scope of this project was to build a bi-directional GUI capable of graphical patching of opcodes to create instruments, with the ability to both input and output Csound code. This would be easily attainable if CsoundXML had complete parsers and code generators available, because we could simply apply an XML stylesheet transformation to produce a diagram from the CsoundXML and vice versa. However, these missing links complicate matters and we must now consider how much of the problem it is possible to address with the constraints of this project. For example whether we wish to pursue CsoundXML as a solution, or implement a graphical application that can read and output Csound code directly.

The latter is a more likely choice at this point, and we will shortly give further consideration to a choice of framework on which to construct our program. It should be noted that we wish to avoid use of a raw graphics library (such as those described in 2.7) if at all possible, since this would require manual implementation of low level diagram drawing features such as

drag and drop, resizing and placement of shapes etc. Such work is not in scope of the project aims.

3 Requirements

3.1 Introduction

Here we present the requirements for the project. Due to the lack of a real end-user, these are mainly influenced by discussions during the proposal stage and the subsequent literature review. Throughout, consistent use is made of certain words indicating the necessity of the requirement, defined as:

- **Must** – the system must implement the described functionality in order for the project to be considered a success, i.e. they are mandatory requirements
- **Should** – such functionality is desirable and useful but is not absolutely required; the system will still be usable without it
- **May** – such functionality would add value but is not particularly important

Sorting the requirements by these qualifiers leads to an approximate ordering that indicates the main increments of the project. That is, all *must* requirements will be addressed first, followed by *should* and so on. The requirements are also split into functional and non-functional categories.

3.2 Functional Requirements

At a high level, the system is required to provide a graphical environment for constructing and editing Csound instruments by manipulating the components as symbols in a drag and drop manner. Specifically required functionality is as follows.

3.2.1 Mandatory Requirements

1. Any diagrams created **must** be exportable to Csound code in a consistent and understandable way. This is the main project aim.
2. There **must** be a large, resizable, scrollable workspace in which the diagrams can be constructed, which **must** be able to house an unlimited number of connected diagram symbols. This is a standard feature in the domain of diagramming tools, allows the majority of available screen space to be used to display the diagram, and does not restrict the size of the diagram.
3. A library of symbols **must** be available for insertion into the diagram, one for each possible Csound opcode. That is, the entire range of opcodes available in Csound **must** be usable in diagrams.

4. The opcodes **must** be presented in categories for ease of location, since there are a large number available.

5. Opcodes **must** be able to be selected and moved around the workspace using drag-and-drop, in line with expected behaviour in this class of application.

6. In order to allow at-a-glance interpretation of the diagrams by the user, opcode symbols on the workspace **must** display:

    a. The name/type of the opcode

    b. Input parameters and their values

    c. Output variable names

7. The user **must** be able to change the name of output variables from the design workspace, since the formal output parameter will not always be the desired choice name for the result of the opcode.

8. The user **must** be able to drag the mouse to interactively/visually create connections between components.

9. Optional parameters for opcodes **must** be able to be specified in the diagram, including arbitrary length parameter lists, in order to allow full flexibility in the use of opcodes through the diagram.

10. Instrument diagrams **must** be able to be saved and restored (in a possibly program-specific format). Obviously a user may wish to return to a diagram and continue work on it at a later stage.

11. The equivalent Csound code for an opcode symbol added to the diagram **must** be accessible at all times during design, to allow the user to review the code representation of their design at a detailed level.

12. The tool **must** support creation of orchestras (i.e. multiple instruments in one logical collection), not just single instruments. This is mainly for convenience, since compilation of the orchestra could be completed with a text editor.

13. Functions and expressions **must** be handled since Csound allows for more complex inputs to opcodes than simple variables. Specifically the interpretation of functions as either opcodes or part of an expression.

3.2.2 Recommended Requirements

14. An alphabetical list of opcodes **should** also be available for selection to aid in location of known opcodes.

15. The program **should** be able to import existing Csound orchestra code and draw the representative diagram.

    a. There **should** be some algorithm to lay the shapes out that minimises overlap of connecting lines and if possible follows modular synthesizer layout conventions discovered during literature review.

    b. The system **should** not lose any information on an import-export cycle – for example the user should be able to import a Csound orchestra file to a diagram (requirement 15), use the export function (requirement 1), and end up with an identical Csound file.

16. The program **should** be able to record a limited number of comments for each opcode and each instrument, and insert them into the outputted code in appropriate locations, to allow some self-documentation of Csound diagrams.

17. The program **should** allow editing and storage of the orchestra header text as part of the interface, and include this in the generated orchestra file (requirement 1).

18. The program **should** be easily extensible for new opcodes that may be added to Csound in future. This is a shortcoming of many of the current implementations reviewed in 2.4.

19. The program **should** perform basic validation on variable names as a convenience to the user (avoiding them needing to first complete the diagram and attempt Csound compilation of the generated code to detect errors).

3.2.3 Optional

20. An opcode *search* feature **may** be implemented to allow rapid location of a specific opcode.

21. The symbols **may** have different geometric shapes reflecting the type of opcode, based on the de facto standards outlined in 2.3.2 (p7)

22. The program **may** rasterize diagrams and export them as common image formats such as PNG, JPEG if requested by the user. This would be useful for including Csound diagrams produced with this tool in documentation or web pages.

23. The program **may** connect with the Csound executable to provide verification of generated code and possible auditioning of instruments using a MIDI controller or preset score.

24. Functionality to allow the creation and use of user defined opcodes **may** be added.

3.3 Non-functional Requirements

The following requirements are not related to the presence or absence of specific functions of the system:

25. The system **must** be delivered, documented, and tested for compliance with the requirements by the project hand in date of **27 April 2009**.

26. Code generation **should** complete quickly to allow rapid adjustment of the diagram and regeneration of the code by the user. A reasonable average time would one second for each instrument. For individual opcodes there should be no discernible delay in generation and display of code.

27. The program **should** be able to operate in a cross-platform way, so that it is able to support the same operating systems as Csound itself.

3.4 Summary and Discussion of Requirements

One advantage of specification of the system by the programmer and analyst is that there are no obvious conflicts or ambiguities in the requirements at this stage. We have identified all the key user requirements and categorised them according to relative importance. We will now proceed to give a brief outline of the main areas of work for this project, based on the functional requirements. This forms a basis for planning the project increments, and will later assist with logical grouping of design and implementation details.

Requirements with the "may" qualifier are not addressed here, on the assumption that they will be considered later after the more important requirements are satisfied, and will need no special allowances in the earlier design of the system other than good design principles and maintainable code.

- **Workspace and generic diagramming functionality –** for example, symbol insertion, deletion, drag-and-drop, resize etc. (Requirements 2, 5)

- **Opcode acquisition and selection** – the mechanism to make the entire Csound opcode collection available and accessible for use in the UI (Requirements 3, 4, 14, 18)

- **Opcode details and connection** – detailed editing of opcode specifics such as input parameters, output variables and comments, connections and their effects on the diagram appearance (Requirements 6, 7, 8, 9, 16)

- **Orchestra-level features** – management of multiple instruments, and editing of the orchestra header (Requirements 12, 17)

- **Expression entry and validation** – the generalisation of opcode inputs to handle expressions rather than simply the output of another opcode, validation of variable names in these expressions (Requirements 13, 19)

- **Code generation and export** – generation of Csound code for individual opcodes, instruments, and the orchestra as a whole (Requirements 1, 11)

- **Saving/Loading of diagrams** – in either a "proprietary" or preferably an XML format (Requirement 10)

- **Parsing and import** – full parsing of existing Csound orchestra code and conversion to editable diagram format (Requirement 15)

With these requirement related work units in mind, interdependencies and the order in which they should be addressed will now be considered.

A clear starting point would be to set up the **workspace and generic diagramming functionality**, since as we have suggested in the Literature Review this is most likely to be obtained from a ready-made tool or library and at this stage need not involve Csound specific features. In parallel we could at this stage consider a suitable **internal representation for the opcodes** that will be available for insertion and then import the entire list of opcodes to this format, perhaps using automated means.

The next stage would be to decide on the **exact graph/diagram representation for a Csound instrument**, precisely what data would be represented in each element of the diagram, and how the user accesses and edits this. This stage has preliminary links to code generation because the diagram editing must be expressive enough to allow construction of instruments using the full capabilities of the language. In fact once this has been decided, **code generation** for the instrument should follow naturally and simply. This stage would also involve designing the import mechanism to instantiate the reference opcodes catalogued in the previous stage into symbols that can be manipulated on the diagram. With this, we would also address the issue of **presenting the opcode types for selection** in the UI.

Generalisation of the instrument model to an **orchestra** containing multiple instruments and a header could follow this, together with code generation for the entire orchestra and the production of orchestra files. This would then complete implementation of the possible

different scopes for variables, allowing the issue of **expressions and variable validation** to be addressed at an opcode level.

Finally we would implement the **saving and loading** of diagrams, through extension of the functionality provided as part of the generic diagram software. This would complete a working system for the construction of diagrams and generation of code, and we could then turn efforts to **parsing and importing** existing code to diagrams as a further increment.

In terms of the non-functional requirements, finishing by the **deadline** is a matter of project governance rather than a particular unit of work, although care will be taken to address the most critical requirements first and avoid "feature creep" in the designs. To ensure **rapid code generation**, algorithms will be written in an efficient manner and profiling will take place after testing, if required. **Cross-platform compatibility** will be considered in selection of the programming language.

4 Design

4.1 Introduction of Concepts

There are a number of important Csound concepts which we must either encapsulate into our diagrams, or which affect them in some way. These will now be enumerated with a brief description of the data they contain to aid understanding when they are later discussed in more detailed designs. Of necessity, a description of some problems anticipated with encoding these concepts into diagrams will be given.

4.1.1 Orchestra

From the description in the Csound manual (12): the Csound orchestra section contains:

- A header section, which specifies global options for instrument performance
- A list of User defined opcodes and instrument blocks containing UDO and instrument definitions.

User defined opcodes are for the time being out of scope of the main project (as they are not a key requirement). This leaves the header and a collection of instruments to be encoded in the orchestra in our program.

> The *Orchestra Header* contains global information that applies to all instruments and defines aspects of Csound output. For example, a Csound header may look like:
>
> ```
> sr = 44100
> kr = 4410
> ksmps = 10
> nchnls = 2
> 0dbfs = 1
>
> massign 1, 10
> ```
>
> (12)

The header does not have an obvious parallel with block diagrams, unlike the structure of actual instruments which we will discuss next. For this reason plain text storage of the header is suggested.

4.1.2 Instrument

An instrument is a collection of ordered opcode calls which process signals as they pass through the instrument, surrounded by the `instr` and `endin` statement as seen previously in the literature review. Instruments are numbered uniquely in an orchestra.

Representing an instrument purely as a list of opcodes though does not allow an obvious transition to the desired graph like structure of diagrams. Therefore it seems better to represent an instrument as a graph, generating the list of opcodes from a traversal of this graph (section 4.1.4 on variables suggests a possible problem with this, however).

In some examples the instrument has a closely associated comment describing it, and requirement 16 would suggest that we store this with the instrument data structure.

4.1.3 Opcode

An opcode is a unit which generates, changes or consumes a signal in some way. It accepts inputs and produces outputs (although not both in all cases). When called in instrument code, the syntax is as follows:

```
output1, output2, ... opcode input1, input2, ...
```

In the case of multiple inputs and outputs, the position in the ordering determines the interpretation of the value with respect to the opcode's function, just as with function calls in languages such as C.

There are two quite different interpretations of an opcode as a data structure, both of which are important to our application. These are:

- the **call/instantiation** of an opcode in an actual instrument, as seen above, which must contain a way of identifying actual opcode invoked and the actual parameters being passed

- the **definition** of an opcode, including its name and formal parameter information

Clearly we need a way of representing the second of these so that we have a template to present to the user for insertion to the diagram (as per requirement 3). We also need a representation for the first in order to record the specific assignments the user makes to the opcode once it is on the diagram. This is rather like classes and objects in object-oriented programming although it is not anticipated that it be implemented as such.

4.1.4 Variable

A variable stores the output of an opcode. Note that "=" is in fact a valid opcode and expressions (to be dealt with shortly) are valid when passed as parameters. Therefore assignment to variables can always be interpreted as being assignment to the output of an opcode. Following this reasoning we can actually store variable assignments in the opcode that makes them, since there is no further information to be encoded.

A potential problem with this approach, however, is that in the sequential code representation of an instrument, a variable can be redefined after it has been first assigned. Therefore the ordering of the opcode calls is actually important and we cannot completely

eliminate the ordered list structure in favour of a graph without possibly losing information or making certain instrument structures impossible.

Scoping variables is not an issue because there are naming conventions which encode this information, which we can enforce (see Table 1 p19).

4.1.5 Parameter

A parameter is a specification of an expected input for an opcode. When referred to in documentation, these typically have a formal name to suggest the purpose of any value passed as that parameter. This is not technically necessary since the ordering determines the actual interpretation of the value but we will be storing the formal names in order to be able to present them to the user and satisfy requirement 6.

Another consideration with parameters is whether or not they are optional. This would appear to be a simple binary property but Csound in fact has some rather complicated optional parameter arrangements. For example, consider the following syntax definition for `linseg`, the linear segment opcode, taken from (12):

```
ares linseg ia, idur1, ib [, idur2] [, ic] [...]
```

The interpretation of this is that there are three required input parameters followed by an arbitrary number of further *pairs* of parameters (specifying in this case duration of the segment and target level for the signal). In other words, specifying `idur2` alone is not valid, so there is a more complex structure than just a binary property of the parameter.

However, instead of attempting to implement this complex optional parameter structure and risk limiting flexibility we will instead allow the user to add/remove the "slots" for all optional parameters arbitrarily in the UI. It will then be their responsibility to ensure that optional parameters are used in a valid way that allows the generated code to parse successfully in Csound. In this way we can leave the property of a parameter being optional as a Boolean value and simplify coding without compromising the requirements.

4.1.6 Expression

An expression is a combination of mathematical operators, functions and variables which evaluates to a value and can be passed into an opcode as an argument. The availability of expressions threatens to complicate the diagrammatic representation because they remove the constraint that the relationship between an input parameter on an opcode and the output of another opcode be 1:1. Consider this assignment as an example of a simple case:

```
a2r oscil 10000, kcps, 1
```

The second input is simply the variable `kcps`, so the mapping between that input parameter and the output of whatever opcode produced the `kcps` is 1:1 (and could be represented by a single connecting line on our diagram). But now consider a more complicated advanced example using expressions:

```
out (a1l+a2l+a3l)/3 * aenv
```

Now we are passing in an expression containing four variables as a single input and the relationship between input and outputs has become one-to-many. Not only that, but there is additional information to be encoded, namely the arithmetic operations.

Obviously it is therefore insufficient to simply store a reference to another opcode's output in order to record the actual input for a certain parameter. Even a list of references to multiple outputs would also be inadequate because the details of how to combine them arithmetically would not be stored. We therefore propose that input expressions be stored as a string as they would be written natively in Csound. The string would then be parsed when required to identify referenced variables and at this point lines on the diagram could be drawn.

4.1.7 Comment

A comment is a piece of descriptive text prefixed with `;` which will not be parsed. In Csound this is permitted anywhere but for the purpose of diagrams we will restrict this to common locations, for simplicity of storage. There will be slots to allow commentary to be stored for **opcodes** and **instrument** definitions only, since these are the most common use. This is in line with requirement 16.

Recall from the introduction that other concepts, in particular the **score** are out of scope of this project.

4.2 Selection of Diagram Framework

It is at this stage necessary to select a general architecture for the application and also a framework on which to base it, from those overviewed in the literature review. The framework will influence the architecture, and affects how we will address the topics covered in the previous section in more detail, so must be decided first. Note that the choice is currently between two possibilities: extending/supplementing **Dia** with Csound specific features, or implementing a new tool on top of the **JGraph** library.

4.2.1 Dia

We have seen in section 2.6 that Dia is an open source diagram drawing tool offering XML representations of shapes and diagrams. However a thorough investigation with the requirements in mind revealed several deficiencies that make it unsuitable for this project. What follows is not an exhaustive list of criteria against which Dia was evaluated, rather an explanation of several key problems with its use for Csound diagrams as required.

4.2.1.1 Design-Time Shape Editing

Shapes must be predefined as XML files and cannot be modified once inserted into a diagram. This complicates the use of opcodes with infinite optional parameters, such as

`linseg` as seen, because extra component "legs" cannot be added at runtime to connect these extra parameters. The only way to enable this would be to write software which dynamically generates the XML shape files with the correct number of parameters, at the user's request. However we must also assume that they may wish to modify the number of parameters later which would require regeneration of the shape after it has been inserted to the diagram, and it is unclear how Dia deals with this. Alternately we may abandon the idea of individual component legs for the different parameters, but this is a crucial feature of the conventional diagram style.

Formally, this problem would prevent satisfaction of requirement 9.

4.2.1.2 Editable Text Regions

Multiple editable text fields on a shape are not permitted by Dia – a shape has only one, default editable area, and there are no "text boxes" that can be specified as part of a shape. These are required if the user is to be able to individually name the output variables in the diagram or enter expressions for input values (as per requirement 7).

It is possible to label the connecting lines between output and input terminals which would appear to allow entry of the required data. However, as will be seen in more depth later, this is not an appropriate model for the Csound language since it allows the appearance of multiple variables in places where this is not allowed – for example, particular opcode outputs.

4.2.1.3 Geometric Considerations

Sizing and proportioning of components and spacing of the terminals/connection points behaves strangely, and how to achieve the correct positioning of this relative to the size of the symbol. is not well documented. This does not violate a particular requirement but would detract from the usability and aesthetics of the application.

4.2.1.4 Real-time Code Generation

Real-time processing, for example generating Csound code on the fly, is difficult without access to hooks for internal events such as connecting two symbols or renaming an opcode. The only real way would be to achieve code generation would be transform the saved diagram, which makes requirement 11 difficult to satisfy.

It is possible that the Python scripting capabilities would allow access to diagram attributes for the open document (and this is supported by (26)) but this is not well documented and Python is an unfamiliar language to the author which would make this an inefficient route to follow.

4.2.1.5 File Format Expressiveness

As a format intended to represent mainly geometric data on the positioning of shapes, the Dia file format actually lacks sufficient "depth" to be able to represent a Csound instrument

without losing information originally present in the code. The only way to store this extra information is to misuse shape attributes intended to affect visual properties of the diagram.

For example, to store the name of an opcode or parameters etc., we would have to use the text field on a shape which in addition to the earlier considerations on editable text regions results in difficulties with the code representation. For example, setting the name of a symbol to "`oscil`" results in code such as the following in the Dia document XML:

```
<dia:attribute name="text">
<dia:composite type="text">
<dia:attribute name="string">
<dia:string>#oscil#</dia:string>
</dia:attribute>
<dia:attribute name="font">
<dia:font family="sans" style="0" name="Helvetica" />
</dia:attribute>
<dia:attribute name="height">
<dia:real val="0.80000000000000004" />
</dia:attribute>
<dia:attribute name="pos">
<dia:point val="10.1522,7.01745" />
</dia:attribute>
<dia:attribute name="color">
<dia:color val="#000000" />
</dia:attribute>
<dia:attribute name="alignment">
<dia:enum val="0" />
</dia:attribute>
</dia:composite>
</dia:attribute>
```

This representation is obviously lacking in semantics and these must be inferred from presentational characteristics, which is not desirable. Nowhere, for example, does the code state that `oscil` is the name of an opcode. It would be preferable to have a representation where the geometry of the diagram was secondary to the semantic information about the opcodes in use, rather than the opposite.

Connections between symbols in Dia are similarly difficult to extract the semantics for, which would likely cause problems interpreting the diagram structure for code generation.

This inability to store all but very simple items of data (and even then not in a way best suited to programmatic manipulation) makes it unlikely that we can elegantly satisfy any requirements relating to the storage of additional data within the diagram (for example expressions, comments etc.)

Having eliminated or at least discouraged Dia as a choice for code/feature reuse, we will now consider the features of JGraph which appear to make it more appropriate for this task. It is also worth mentioning that use of JGraph will force an object-oriented architecture on the application, which is actually rather appropriate for this problem.

4.2.2 JGraph

The JGraph library is a freely available graph visualisation library for Java. Unlike other graph drawing solutions we have examined, JGraph focuses on interactive design of graph based diagrams, with graph analysis and rendering a secondary concern.

Throughout the rest of the project, use will be made of various pieces of graph terminology, and also some JGraph specific terms. To begin with, let us define the following based on definitions in (31 p. 2):

- **Vertex** – also known as a node, this is a point on the graph which may be considered to be connected to certain other vertices
- **Edge** – the connection between two vertices

The following are JGraph specific extensions to graph theory, explained in (28)

- **Port** – an artificial addition in JGraph used to indicate places on a vertex where an edge may be connected to that vertex. Ports are considered to be children of one vertex. This effectively implements an ordering on connected edges, allowing the differentiation of edges connected to a vertex without needing to inspect the opposite end. This has important implications for Csound diagrams as we will see later.
- **Cell** – a JGraph term used in general to refer to vertices, edges and ports.

With this defined, it should be fairly straightforward to see the correspondence to Csound diagrams as discussed so far. **Opcodes** can be represented as vertices, with the input and output **parameters** as ports and the connections between them as **edges**. The problem of code generation then becomes one of graph analysis to some extent.

Considering this further, we will now investigate some key features in JGraph with brief suggestions on how they may be used to satisfy the mandatory requirements for this project. These suggestions are preliminary and more detailed design involving specific features will be dealt with properly in the remainder of the chapter.

4.2.2.1 The JGraph Class

The `JGraph` class is the top level class for a graph and provides the workspace UI component used to contain the interactively designed graphs. It conforms to the `JComponent` interface and so can be used easily in an application built using the Swing API. It is this component that (by aggregation of other components) contains the features

required by requirements 2 and 5. In particular: we can ensure the workspace is resizable by ensuring that the main application frame is resizable, then inserting the `JGraph` into a `BorderLayout` or similar. Scrolling can be implemented using a `JScrollPane` which contains the `JGraph`. The `JGraph` places no software limit on the number of vertices which may be added to it, which in turn would not restrict the number of opcodes we could place.

A `JGraph` would be a suitable representation for an individual instrument, and we could extend the class to store other instrument specific information. Then, an array of these `JGraph` derivatives with some additional information might form an orchestra representation.

4.2.2.2 Vertices and Views

The default implementation of a vertex can be extended in the usual Java way with additional information and functionality. This allows storage of the extra Csound specific data and methods that are needed to allow us to satisfy requirements 6, 7, 9, 16. All cells in JGraph have views and renderers to display them on the graph UI, and these can be overridden to allow us to display arbitrary information about opcodes to the user with positioning of our choice. This is a great improvement in flexibility over Dia.

The vertex object representing an opcode would also be a suitable place to store a method for generating the Csound source for that opcode, which is requirement 11 and a precursor to requirement 1.

It appears to be difficult in JGraph to insert vertical text into the view and so it must be assumed that the labels for the ports will be horizontal. If an arbitrary number of ports are to be allowed on an opcode vertex, they must therefore be stacked in vertical lists at the left and right hand edges of the vertex (for inputs and outputs respectively). Therefore it is expected that diagrams will flow left to right rather than top to bottom as is tradition. However there is no requirement to the contrary.

4.2.2.3 Edges

Edges can be dragged between ports to connect them, and stay connected when the vertex to which the port belongs is moved. This is generally expected functionality in diagrams, and is provided by default in JGraph. The edges can be interrogated to find the ports and/or vertices connected to either end which allows the graph traversal necessary for code generation. Edges can be labelled, but as will be discussed later this is probably not necessary if we can label the ports.

4.2.2.4 Serialization

Like most Java classes, objects produced from the JGraph classes are serializable and so can be written out to a file. This will allow us to save and load (unserialize) diagrams as specified by requirement 10.

4.2.2.5 JGraphX

JGraphX is "the next generation of Java Swing Diagramming Library, factoring in 7 years of architectural improvements into a clean, concise design" (27). However it appears to lack documentation and is not sufficiently mature that we will consider it for this project, instead preferring the older and more established JGraph.

4.3 Development Methodology

Now that some justification has been given for the use of JGraph, a suitable development lifecycle model will be briefly discussed. Due to the fact that JGraph is an unfamiliar platform, a method of **evolutionary prototyping** will be adopted, where the application is constructed and constantly refined as familiarity is gained with the different parts of the framework (32 p. 119).

Throwaway prototyping, while likely to result in a more cleanly structured end product, will not be applied due to the time constraints of the project. This does not preclude re-implementing the application after the initial exploratory stage (which forms the main part of the project) is complete.

Traditional approaches such as the **waterfall model** (33 p. 66) are inappropriate for this project because the clear design needed for this model cannot be produced until a suitable understanding of the tools has been gained. Such an understanding relies on experimenting with JGraph, which given project time constraints must be carried as part of development.

4.4 Graph Model

In light of the selection of an augmented graph structure (i.e. with ports) as a suitable data structure, the mapping between Csound concepts and the graph model will be considered. Note that not every concept discussed in section 4.1 will be involved in the graph structure. The graph representation will store instrument structure only, and all other necessary functionality outside of that will be implemented with standard Java features.

It should be reasonably apparent that a **vertex** in a graph suitably models the role of an **opcode** in a diagram. That is, it may be connected to other opcodes to receive and pass values from and to them. At this point it is worth clarifying some terminology: hereafter let an **opcode** be the formal definition of the name and parameters of a Csound opcode, and let an **opcode vertex** be a particular instance of that opcode on a diagram.

The outputs and inputs must be differentiated from each other, however; as must the individual parameters in each of those lists. For this the **ports** extension to the vertex is appropriate which allows distinct and well defined terminations for the connections to another opcode. These ports will map directly to the idea of formal **parameters**. Specifically there will be one port per parameter on an opcode vertex.

Following on from this, the concept of an **edge** gives a way of representing the assignment of a variable to a parameter. Graphs in JGraph are always directed and so an edge has a direction. The direction will be implicit by the fact that outputs may only be connected to inputs and the direction of assignment is then obvious. Figure 9 shows an abstract diagram of three partially connected opcode vertices. The circles represent ports and the arrows represent connecting edges. It is shown that a single output may connect to multiple inputs, which would reflect the ability to make use of the result of an opcode vertex multiple times.

The problem of expressions identified earlier must now be discussed again. It was decided previously that an expression specified as an input to an opcode vertex may be entered as plain text and then parsed to form connections (edges) to the correct outputs. However, if this is the case, it means that the edges alone do not form an entire representation of the input to that port, and so the configuration of the instrument cannot be represented entirely by the graph structure.

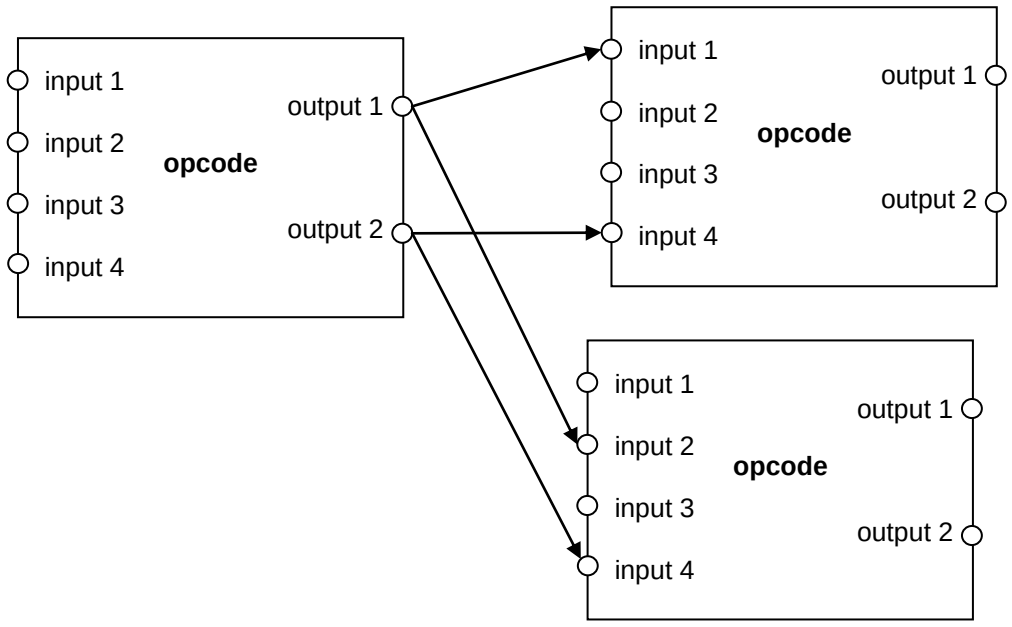


Figure 9: Abstract diagram showing opcodes as graph vertces with ports

One exception to this may be if an expression was itself parsed and expanded to a tree representation, which could then be represented on the diagram since a tree is a graph. However this would clutter the workspace with symbols not directly relevant to sound generation, without major benefit to the user. Figure 10 shows the result of expanding the example of `out (a1l+a2l+a3l)/3 * aenv` used earlier, with a simplified view of the opcode vertices involved. It can be seen that this does not allow the user to gain anything, except perhaps to access the intermediate result `a1l+a2l+a3l` for use elsewhere. This is only a small advantage compared to the cost of the clutter.

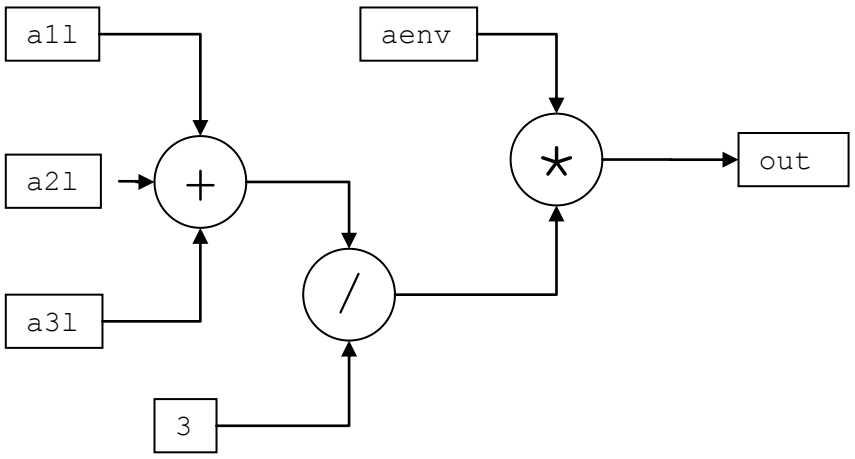


Figure 10: Expansion of expressions to diagram elements leads to clutter

The use of this technique will therefore be avoided, and so we confirm the compromise of storing plain text expressions. This means graph structure will now provide the *visualisation* of the instrument layout only, and a full representation and expansion to code will require access to the expression for each input. In other words, the presence of edges will be dictated by the input expressions alone – the former will not exist in their own right. Practically speaking, a function will be implemented which will, for a given input port and expression, make all the necessary edge connections to the referenced outputs (discussed later). An expression can be stored in the port object to which it relates.

This leaves the question of where interactive editing of the diagram now stands, since as discussed previously, JGraph facilitates drag and drop creation of edges. The answer (to be discussed fully later) is that in simple cases where a given input takes its value from only one output, it is still perfectly acceptable for the user to indicate this by dragging an edge between ports with the mouse (requirement 8). There is only a problem if an expression is already present on the input port where the edge is being targeted.

There need not actually be separate cases for expressions and single variables, since a variable is an expression itself. For the case of manual drag-and-drop connection of a single output to an input, we can fit it into the existing model by causing the drag-drop operation to set the expression on the target port to the name of the source output variable. Then calling the function for making the connections based on the expression (introduced above) can be called to actually make the edge. So, if the expression string for a port is made accessible and editable, connection between opcodes can appear to the user to be possible by either drag-and-drop or expression entry.

As a final note on the graph model, we will discuss requirement 7 which is that the user be able to specify the name of variables used for opcode vertex output. This is obviously necessary because the formal parameter given in the opcode definition is likely to conflict

with other instances of the same opcode, or even different opcodes (for example `ares` as shorthand for the-result-at-rate-a is hardly likely to be an output name unique). The formal parameter should, then, be given as a default but the user must be able to change it, and it is the user specified value to which expressions will refer. This user defined value would be stored in the output port object.

4.5 Opcode Acquisition

A means to make the hundreds of available Csound opcodes available for insertion into diagrams must now be discussed.

It may be argued that having a catalogue of every possible opcode in the application is unnecessary when a user could just enter the name of the opcode they wanted onto the symbols directly. However, this would not provide any information about what parameters a certain opcode was expecting, and would mean the user had to add the correct ports to the vertex themselves. It would be better to have all this information available automatically once an opcode has been selected, and therefore storage of this information is necessary.

4.5.1 Representation

Representation of an opcode *definition* is slightly different to that of an opcode vertex, in that it represents a name and some formal parameters rather than any particular input and output values. Recall from section 4.1.5 that the status of whether or not each parameter is optional will also be stored.

Since a categorised list is required, a group/container data structure for storing multiple opcodes will also need to be implemented. Such a container need only store its name, any opcodes contained, and any nested further containers. In other words, it is a tree node.

This catalogue of opcodes will need to be persisted between multiple runs of the diagramming tool, therefore it must be loaded from a file on disk. XML is recommended as an appropriate language for this file because the opcode definitions are structured data, and such a list of them may be reusable in other applications.

4.5.2 Acquisition

Manual entry of this catalogue information would be tedious and time consuming, so we instead present a novel method which is to parse these from the XHTML-based Csound manual. The Opcode Quick Reference page in the HTML version of (12) provides an ideal listing of the name and formal parameters of every opcode. What is more, it groups them into named hierarchical categories which are readily transferred to menus (see following section).

Parsing is relatively straightforward because the page is valid XHTML so an XML parser can be used. Certain formatting characteristics of the page make it easy to infer the semantics of portions of text, for example to detect when a new opcode is being introduced and which portion of the string is its name, and also which category heading it falls under.

This method of acquisition has the major advantage that when new opcodes are released for Csound and documented in the manual, we can reparse the page and bring the diagramming tool up to date with the latest version of Csound. If the catalogue of opcodes is stored on disk there is even no need to recompile. This fulfils requirement 18.

Recall that when WinXound.net was reviewed in 2.4.10 (p17) we noted a possibility of reusing the internal opcode list for that program, which was readily available. However this does not address the problem of updates so we have chosen not to reuse it at this time.

4.5.3 Presentation

Using the categories recovered from the quick reference manual page, a hierarchical menu structure can be formed, much like the one seen during the review of Crocodile Clips in section 2.6.5. This will satisfy requirement 4.

It is also straightforward to sort the entire list of opcodes by name and produce menus based on an A-Z ordering for use alongside the categories in the case where a particular opcode needs to be located rapidly (requirement 14).

4.6 Orchestras

As already discussed in the introduction of concepts, an orchestra is just a collection of instruments with a plain text header attached. The instruments are not necessarily ordered, but are uniquely numbered. So, to implement orchestras we simply need: a class with a collection of instruments; a way to obtain the next number in sequence for a new instrument added to the orchestra; a string field for the header; and a function to aggregate generated code for each instrument.

In terms of the UI, it is anticipated that a tabbed view be used to contain as many instrument graph workspaces as are required for the orchestra, a popular UI metaphor in multiple document applications.

4.7 Detailed Editing and Connection

Detailed consideration will now be given to how input expressions and output variable names will be entered by the user, and the effect this will have on the graph visualisation. Requirements 7 (renaming output variables), 8 (drag-and-drop connections), 9 (optional parameter entry and extension of the parameter lists), 13 (expressions and functions) and 16 (opcode level comments) relate to this.

Broadly speaking, editing can be divided into three main categories: inputs, outputs and comment. It is proposed that a separate user interface be implemented for editing these, since editing in place on the diagram itself is complicated to program in JGraph, and would also clutter the workspace. The interface would take the role of a **per-opcode-vertex properties window**, which could be opened and closed by the user as desired. This would allow the

graph workspace itself to be a read only view of data (aside from drag and drop moving and port connection), with all detailed editing taking place in the properties window.

The dialog box would have three sections, one for each of the above categories. Storing comments is trivial so will not be discussed here.

4.7.1 Variable Length Parameter Lists

In section 4.1.5 on parameters, a simpler approach to optional parameter lists was suggested, which involved giving the user most of the control of the optional part of the parameter list. This relies on them being able to specify optional parameters in a way that is valid for Csound, but removes some programming complexity. It also avoids unintentionally restricting flexibility through misinterpretation and then enforcement of the optional parameter structure when parsing the manual as per 4.5.2 .

We must therefore supply a means in the UI to carry out editing of the parameters list (and hence alteration of the number of ports available for connection). This is a separate matter than the actual specification of values for the inputs and names for the outputs, but need not be completely separated in the interface. A tabular representation for each of the two lists of parameters is proposed (e.g. Table 2 for inputs), where the first column holds the formal parameter name and the second holds the actual value/name. Parameter list editing could then be carried out by addition and deletion of rows in these tables.

Optional parameters are always at the end of the list and so it is acceptable to restrict such addition and deletion to the latter part of the table containing them. This avoids accidental reordering or shifting of the mandatory parameters, which would result in unexpected behaviour in Csound since the position of the argument dictates its interpretation.

Formal Parameter	**Actual Value/Expression**
xamp	
xcps	
ifn	
[iphs]	

Table 2: Example input parameters table

Modification of the table structure would result in real-time changes to the diagram (visible behind the properties window) – i.e. ports would appear and disappear to reflect the table.

4.7.2 Input Expression Parsing, Validation and Connection

Expressions were introduced at the start of this chapter and discussed again with the Graph Model (4.4). Design decisions so far have resulted in a need to be able to parse expressions to some extent in order to make connections to the relevant ports and so represent the use of outputs values as inputs diagrammatically. A mechanism for doing this will now be discussed, at a high level.

Given an expression string, it is possible to extract (using regular expressions) the names of variables, or at least candidates for being variables, because they are a simply the substrings which are not operator symbols or numbers.

This matching pattern will also, however, capture function names, for which is it is not desirable to attempt to make representative connections on the diagram. A simple technique for eliminating these is to maintain a list of registered functions. This can be added to the opcode acquisition phase since all functions are clearly defined in the manual. Then, if a candidate variable is in this list, it can be removed from consideration.

The remaining references to variables then fall in to one of three categories based on scope (recall that variable names define the type and scope in Csound, as per (12 p. 48) and discussed in 2.5.1):

- **Local** – variables with local instrument scope, which we can expect to find as outputs elsewhere on the diagram
- **Global** – variables with global scope which we can expect to find in either the instrument or the instrument header
- **Score parameters** – variables beginning `p` that are passed from the score and should not be expected to appear as either an output or in the header
- **Invalid name** – variables which do not have a valid name for Csound

It is therefore possible, using a recogniser for each of these types (based on further regular expressions and some ad-hoc parsing techniques), to implement a simple syntactic validation scheme to aid the user in production of valid code and satisfy requirement 19. This allows the user to be warned in the case that they have entered an invalid name, and avoids attempts to draw an edge/connection for it on the diagram. Also connections will not be attempted for score parameters, although there is no way to detect if they are defined and so no warning will be given.

For the other two cases, edge connection can be attempted. We will now consider how to implement this. More precisely, the problem to be solved is **for a given variable name in an expression on an input port, locate the output port which assigns that variable and create an edge between them**. At first sight this appears to be a simple search problem – and typically diagrams are small enough that this can be solved by a linear search of the output ports without detriment to performance. A problem emerges, however, when we wish to encode an instrument such as the following as a diagram (i.e. build a diagram which generates this code). This is a fragment only, so all variables used can be assumed to be defined.

. . .

```
a1r     oscil  10000, icps*0.999, 1
a1l     oscil  10000, icps, 1
a2r     oscil  10000, icps*0.996, 1
a2l     oscil  10000, icps*1.004, 1
a2r     vdelay a2r,5,5
a2l     vdelay a2l,5,5

outs (a1l+a2l)/2 * kenv, (a1r+a2r)/2 * kenv
...
```

Consider the case of forming a connection for the input expression `(a1l+a2l)/2 * kenv` to the output variable `a2l`. There are *two* assignments to `a2l` which must be decided between, since `a2l` is redefined in terms of itself when `vdelay` is used. In a code representation such as the above, it is obvious which of the two possibilities to use because of the ordering, but in a graph interpretation it is not as clear.

In the case of the user dragging the connection between ports this appears not to be a problem, since the source of the connection will be a single distinct output. However, it has already been decided that dragging a connection will just result in setting of the target expression to the source variable and then application of the algorithm currently being discussed to do the actual connection. While this offers consistency and convenience, it eliminates the ability to distinguish the intended output for connection to the input.

The most obvious method unfortunately appears to be one in which the graph representation is further abused. We propose selection of the *nearest* output port to the left of the input port in question as the source of the connection (since the diagrams flow left to right as discussed in 4.2.2.2). This is not strictly a good solution because it relies on a geometric/visual property of the diagram, rather than a logical property of the graph, but does have some significant merits in addition to solving the problem:

- It is more intuitive for the user to lay a diagram out in the approximate order of the code they are expecting to be generated

- For import purposes (requirement 15), layout of the diagram based on ordering of the source code will result in all the connections remaining valid, and as above it will be easy to see the relationship between the code and diagram

Note that the nearest output technique only need be used if an output is redefined in terms of itself (i.e. appears more than once as an output). There is no requirement that all outputs be to the left of an input they are used in.

Having decided this, it remains to note that if a variable which is not present as an output is specified in an expression, it will not be possible to make a connection. The user should be warned in this case. The exception to this is global variables, which should be checked for in

the orchestra header before warning. Note that we choose to warn rather than forbid the expression outright because of the possibility of the introduction of new variable types/scopes to Csound.

4.7.3 Output Naming and Validation

Requirement 7 is that the user be able to specify a name of their choice for actual output variables storing the result of an opcode vertex.

Validation of variable names has already been discussed in the previous section. The same validation technique can and in fact *must* (for consistency) be applied to user specified output variable names.

The other important point to consider is what happens when an output is renamed *after* it has been connected to one or more inputs. The obvious solution of performing "find and replace" on the expression for any opposite ports is sufficient to deal with this case, although it does have the problem of potentially capturing substrings of other unrelated variables in the expression.

4.7.4 User Editing of Connections

Finally, we must discuss what should happen when a user attempts to modify the connecting edges on the diagram when an expression has already been specified and the relevant outputs/inputs are connected.

First the case will be examined where an extra **edge is connected to an input port that already has an expression set**. The decision is between forbidding the operation outright or finding some way to modify the expression. The former seems overly restrictive in that the only edge which may be connected to an input port by drag and drop is the first one, then any further connections must be made by editing the expression string. Therefore an attempt to combine the output variable for the newly connected edge into the expression will be made. This need only be something as straightforward as appending the variable name separated by an operator such as +. More precise editing of the expression could then take place in the properties box.

Dealing with **deletion of one of a collection of edges terminating on an input** is more complicated. Our choices are between:

1. forbidding the operation (and forcing editing of the expression as the means to delete edges);

2. clearing the expression and disconnecting all ports;

3. attempting to modify the expression or

4. deleting the edge without changing the expression.

3 is the most difficult to implement well and 4 is the least useful, because code generation will be based on the expression and would be identical whether or not the edge was connected! Neither 1 nor 2 stand out as being particularly good choices alone, but a combination of the two is suggested where the user is warned that deleting the edge will clear the expression and so disconnect all other edges. They will be given the choice to cancel and edit the expression themselves.

This warning need not be displayed where the target input port has no other connections.

4.8 Code Generation

Requirement 1 (p28) states that it must be possible to generate consistent (i.e. repeatable) Csound code for the orchestra currently being edited. This is the main aim of the project so we must now discuss how it is to be achieved.

At first sight it would seem particularly simple: the overall application has a fixed number of levels of nested containers which make up the orchestra. We have now defined the data storage requirements for all of these, and Figure 11 below shows how the scalar data is arranged in a hierarchy of nested container objects.

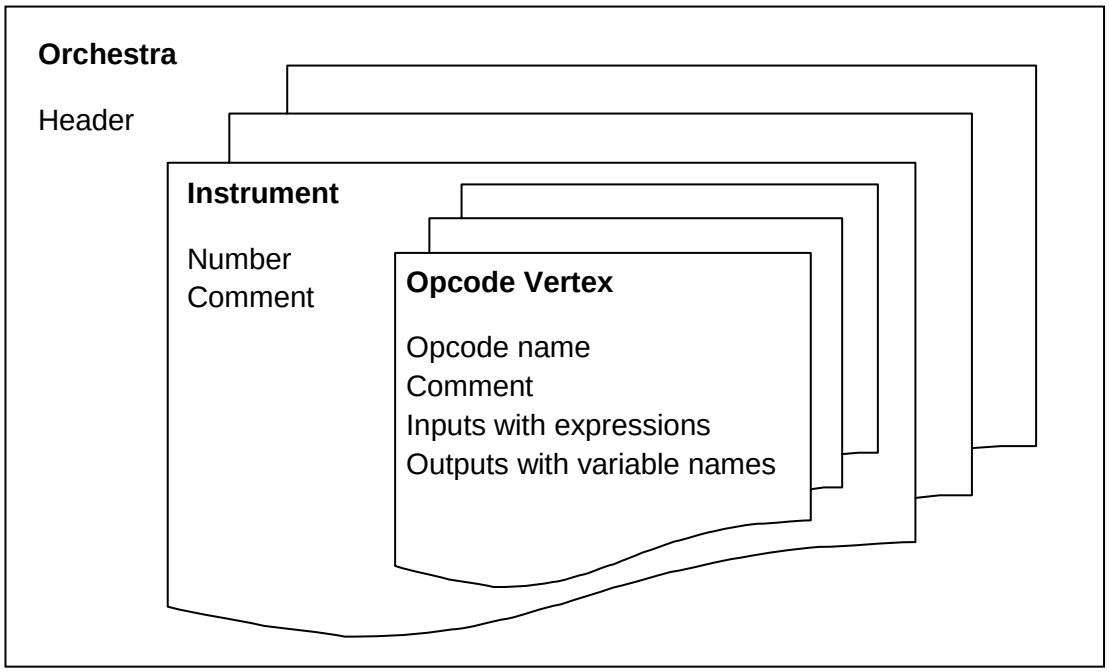


Figure 11: Orchestra structure as a hierarchy of nested containers

Code generation would then, seem to be a case of just flattening the structure according to the rules of the language, resulting in output according to the following pseudo-code:

```
print orchestra header
```

```
for each instrument
  print "instr" and its number
  print instrument comments
  for each opcode vertex
    print comma separated list of output variables
    print opcode name
    print comma separated list of input expressions
    print comment if any
  next
  print "endin"
next
```

The problem here is determining the correct order to output the code for the opcode vertices, which are stored unordered and related only by the graph structure and input expressions. The correct order would appear to be determined by *dependencies*. That is, a variable must have been output previously by an opcode before it can be used as an input (otherwise Csound will generate an error when it attempts to interpret the code).

However, there is a similar problem with code generation to that of deciding where to connect edges based on expressions, which was discussed above. That is, how to decide the correct order of code when a variable is redefined in terms of itself; or in other words which version of a variable to use. Fortunately this is simple to solve using the graph structure defined so far – it is not necessary to calculate dependencies separately; edges can just be traced back from any opcode with no output value (e.g. `outs`) to give the reverse order of the code. The correct version of a variable use will be given by the connecting edges.

Unfortunately this does not necessarily produce very logically structured code. When "backward chaining" in this way it is unclear which opcode should be generated next. A simple example is the opcode `outs`, which outputs to the digital-to-analogue converter (DAC) in stereo. The inputs are signals for the left and right channels. It is, then, unclear whether the value for the left or right channel should be obtained or generated first. When extending this to further opcodes with multiple inputs further up the chain it can be imagined that this would result in an ordering of code which is difficult to define. This would possibly violate the requirement that code generation be consistent.

Recall that the connection of edges due to expressions is based on the physical left to right order of the vertex symbols on the diagram: for an output variable named in an expression the instance nearest to the input is used. It follows that code generation could also be based on the physical layout of the diagram, and this is the method that will be adopted.

An important point to note is that this *will* allow the generation of invalid code if dependent vertices are positioned the wrong way around on the diagram. However if the user is aware of the need to position opcodes correctly it need not be a problem. Diagrams with this

problem are simple to spot because they will have an arrangement similar to Figure 12 which looks rather untidy.

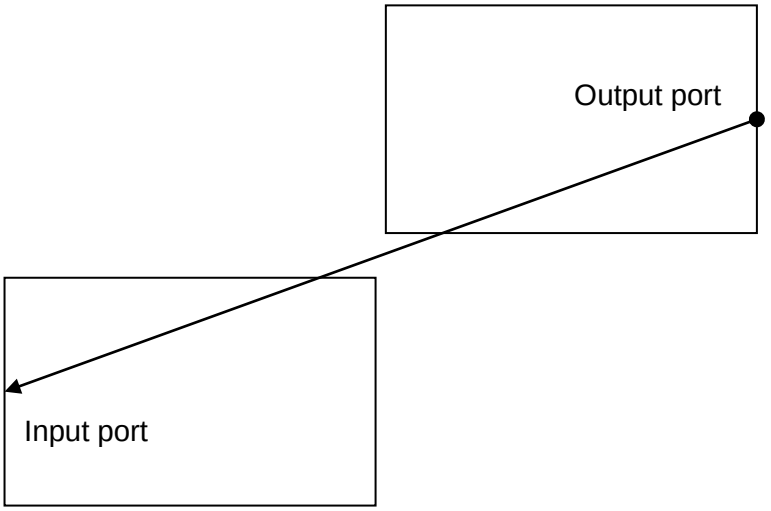


Figure 12: Left vertex generated first but depends on right hand vertex for a value, resulting in error or incorrect assignment in Csound

4.9 Saving/Loading

Java provides an object serialisation framework, and JGraph graph objects are serialisable. Therefore by ensuring the orchestra objects are serialisable, we can implement saving and loading of diagrams rapidly, albeit in a Java specific format. This also deals with versioning of the file format, ensuring that future versions of the tool can detect incompatible files.

The alternative would be to write a procedure to output a custom XML-based format representing the orchestra and contained instruments and opcodes. This would be appropriate because of the hierarchical nature of the orchestra as discussed earlier and shown in Figure 11. However, encoding all the positioning data and writing custom methods to restore it would be very time consuming and we will instead use the serialisation methods.

4.10 Parsing and Import

Requirement 15 is that the program should be able to import existing Csound orchestra code and create the representative diagram. There are two main stages involved in this: parsing the Csound code and laying out the diagram.

4.10.1 Parsing Orchestra Code

To parse orchestra code, there are two possible options. The first is using a formal, structured parser such as recursive descent or shift-reduce. The second is using an ad-hoc parser. The latter is favourable here because in order to create a diagram it is not necessary to extract the

full semantics from the code. We simply need to be able to distinguish the concepts outlined at the start of the chapter.

Therefore a simple algorithm such as the following would be sufficient as a beginning:

```
while not end of file read a line
if instr x seen, begin new instrument and advance line

if inside instrument
  if line not a comment
    parse opcode statement
    else
      store comment to apply to next opcode seen
else
  write line to orchestra header
end
loop
```

Parsing the opcode statement would be delegated to a separate function which would extract the inputs, name, output and any associated comment.

4.10.2 Automated Diagram Layout

In 4.8, code generation based on a physically ordered diagram was discussed. It is therefore reasonable that an imported instrument should also be laid out in a way based on the ordering of the code.

The complexity comes with arranging a layout in two dimensions. A one dimensional layout based on ordering of the opcode statements in the code would be trivial to implement, but adding a second dimension so that opcodes stack *down* the workspace in appropriate places requires some more thought. However, it need not be that difficult. Consider the workspace as a large grid, so that an opcode vertex may be placed in each grid cell. Then a suggested layout algorithm is as follows:

1. Starting at the top of the Csound source code file, create the vertex for the first statement encountered in the top left cell.

2. Inspect the next statement, and if it depends on the output from the previous opcode (i.e. that opcode's output variable appears in one of inputs on the new opcode) place it in the cell to the right and connect it. If it does not depend on any previous outputs, begin a new row of the table and insert at the beginning.

3. Inspect the next statement and check for dependencies again. If there are some, place the vertex for it at the next available cell on the row in which the dependent opcode vertex is located and make the connection. For multiple dependencies find

the right-most of the dependencies and place the new vertex in the next cell in *that* row, so that it is to the right of all required dependencies and consequently the code will regenerate correctly. If there are no dependencies, begin a new row and place at the beginning.

4. Repeat step 3 for all remaining statements.

This will result in the creation of non-overlapping vertices, and in the correct order so that if code is regenerated from the diagram any dependent variables are initialised before they are used. A disadvantage is that it may result in some very long connecting edges although this is purely a visual problem and does not impact the interpretation or functionality of the diagram. Consider the example in Figure 13: opcode vertex F depends on B, E and G. The algorithm positions it in the second row because E is the right-most dependency. However, G which is also a dependency is not dependent on any other vertices and so is to the far left and results in a very long connecting edge to F.

A possible way to resolve this is to apply a second algorithm after all vertices have been imported which minimises edge distances where possible. Such an algorithm would for example move G and B up to the third column. There is a potential problem with doing this, however. Suppose the output of A is stored in a variable, and the output of D then redefines this same variable name. Moving B to minimise the edge distance to F will place to the right of D which results in its input assuming the value of D's output variable, because it is identically named to the output of A and is closer. A must therefore be moved as well to maintain its position as closed output to B. This algorithm has the potential to get complicated quickly and is mainly for aesthetics, so it will not be implemented as a high priority.

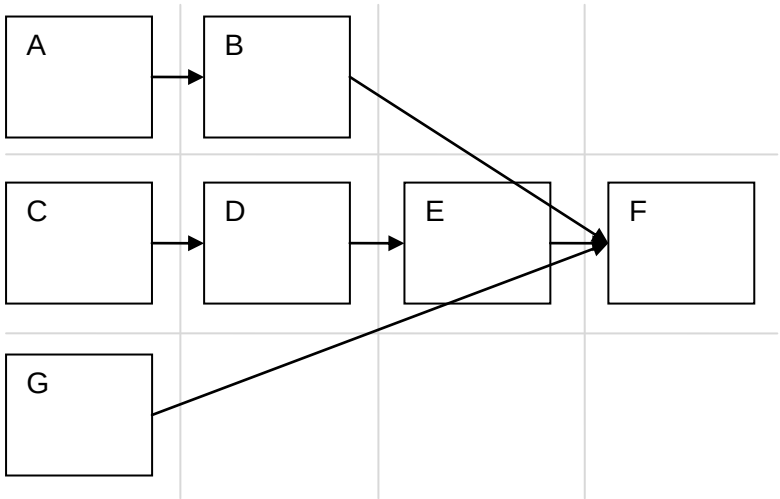


Figure 13: Example for diagram generation showing long edges

5 Detailed Design and Implementation

Here, a more detailed discussion will be given of important parts of the system and the relevant implementation details. Only particularly interesting or crucial functions will be treated in any depth, since the evolutionary prototyping methodology in use has resulted in little in the way of significant design documentation for more mundane parts of the system. The reader is invited to consult Appendix A4 for fully commented source code.

We shall take the approach of justifying and evaluating design decisions and algorithms as they are introduced. Alternative methods of implementing features will be suggested and commented upon where appropriate.

5.1 Language and Tools

Java has already been nominated as the language of choice; because it is cross-platform (satisfying requirement 27) and more importantly facilitates the use of the JGraph library which forms a central part of the application. The Eclipse IDE will be used, since the author has experience with its features and many of these can be used to speed up development. The most useful of these features is automated syntax checking which means the source need not be compiled to detect syntax errors resulting in a major time saving. Also available is automated formatting and indentation of the code ensuring standard layout for source files.

Eclipse also provides a "local history" feature which maintains copies of old versions of files, and allows reversion to an earlier copy if necessary. Because of this feature and the fact that the system will be developed by a single programmer, an additional version control system (such as the Concurrent Versioning System or Subversion) will not be used.

The use of Java promotes self-documenting code facilitated by the `javadoc` tool and structured comments. Separate JavaDoc documentation is not included in the appendices because the source code is supplied, which contains the comments. It can however be easily generated if required.

Good object oriented design principles are followed throughout, for example suitable encapsulation and variable naming.

5.2 High Level Overview

The annotated Figure 14 shows a high level overview of the system architecture. This is intended to summarise the context of the designs produced in the previous chapter in order to prepare the reader for the more detailed implementation discussion in this chapter.

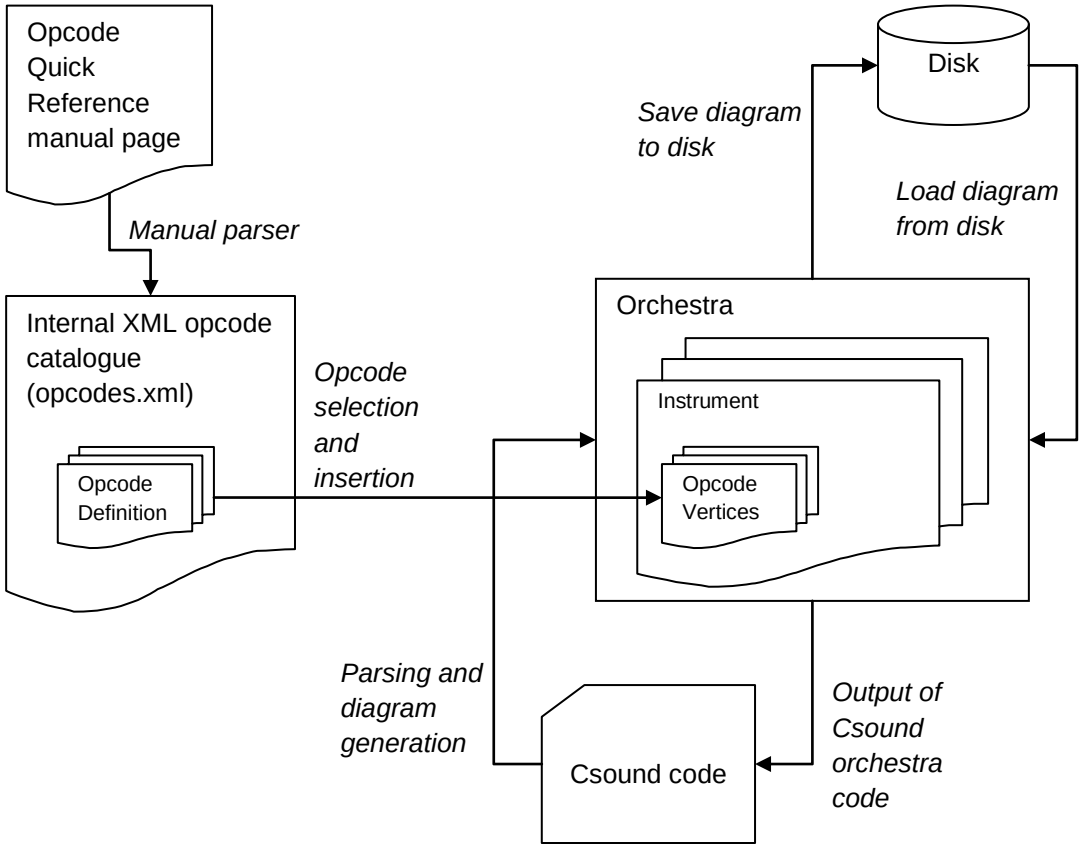


Figure 14: High level overview of system architecture

5.3 Opcode Loader and Format

In section 4.5, a high level design for acquiring and storing opcode definitions was proposed. The implementation of this part of the system will now be examined in more detail. The first consideration is the internal storage format for the opcode definitions, i.e. relevant class structure, since there must be some structure to parse *into*.

5.3.1 Structure and Storage

We implement four classes: `Opcode`, `OpcodeGroup`, `FunctionOpcode` and `Parameter`. An `OpcodeGroup` represents the container class mentioned in the designs, and contains `Opcode`s and/or `FunctionOpcode`s. `Opcode`s have a list of input and output `Parameter`s. `FunctionOpcode` is a subclass of `Opcode` representing a function, since functions can be interpreted as opcodes and so must conform to the same interface. `Parameter` is just a simple data structure but the other classes contain (in addition to their data structure) functionality to load objects of that type from and output to an XML representation. The full definitions of these classes are given in sources in Appendix A4.

For XML processing, the JDOM (34) library is used which allows XML documents to be interpreted as Java objects and handles their parsing and code generation. Note that the XML format referred to here is a format internal to this application – parsing of the manual is yet to be discussed in detail. An example of the storage format is given in Appendix A1 and it should be obvious from this how the groups contain other groups, opcodes, and functions in a hierarchical fashion.

This format is rather simple to read and write so there will be no detailed discussion of this. The file is read in when the application starts and used to create the opcode tree stored in the variable `octree` in `Editor`. One interesting point to note is that if a function is seen during the loading process, it is also added into the statically defined list `regFuncs` in the `FunctionOpcode` class. This list is used later in another part of the system to detect if a token from an expression is a valid function.

We will now progress to examine parsing of the manual to extract opcode definitions.

5.3.2 Parsing of Csound Manual

To extract the opcode definition information from the Csound manual opcode quick reference section, an ad-hoc parser is implemented. This is contained in the class `PageParser` in the `uk.ac.bath.cs.csdiag.opcodeloader` package. The opcode loader is in fact a complete program in itself that accepts two command line parameters. The first is a URI (Uniform Resource Identifier) of the manual page, which can for example be a local file specified with the `file://` scheme, the canonical version of the quick reference at `http://www.csounds.com/manual/html/MiscQuickref.html` or any other URL. The second is the name of a local file where the structured *internal* XML opcode catalogue discussed in the section above should be output. This allows on demand updating of the diagramming tool's opcode catalogue directly from the site, whilst still allowing the possibility of direct modification of the same by advanced users or the system developer.

Although not a recursive descent parser, the implemented parser does follow similar principles of handing off particular language constructs to be processed by separate functions

5.3.2.1 parse

The process begins at the `parse` method which loads the XHTML web page to be parsed into a JDOM `Document` object. A filter is then constructed which is used to extract an ordered list of all HTML `<b>` and `<pre>` elements from the document regardless of where they lie in the XML nesting. This is a useful heuristic that takes advantage of the fact that in the manual page `<b>` tags are only used to format **group headings** and `<pre>` elements are only used when the **definition of an opcode** is being given. This has the advantage that straight away other matter on the page (in which we are not interested) can be eliminated, without having to write code to ignore it explicitly. On the other hand this method is vulnerable to failure if the formatting of the manual is changed.

A more reliable technique might be to parse the DocBook XML sources of the Csound manual if these could be obtained, so that semantic information is not being inferred from formatting.

The parse function then loops through the collected elements and begins the process of extracting information. Lines up to the heading `Signal Generators` concern the orchestra structure and are not opcodes so processing does not start until that text is seen. After this point, the action taken depends on whether the element is a group heading or an opcode definition (which can be distinguished as described above).

If the element is a heading, it means we are beginning a new group for subsequent opcodes. This group has a position the overall hierarchy of groups, and this can be inferred because the full path to the group is given in the heading text. For example, the heading `Signal Modifiers:Standard Filters:Resonant` taken from about halfway down the manual page details a group that is nested two levels deep in the group hierarchy. By splitting the string where a colon occurs, the name of the group at each level can be extracted. The function keeps track of the current `OpcodeGroup` and updates this reference when a new heading is seen.

Hash tables are used to map group names to actual group objects at each level of the hierarchy, in order to allow lookup/matching of groups from the parsed strings in the source code. Full implementation details are clearly commented in the code itself, included in Appendix A4.

If the element is an opcode, the method `parseLine` is invoked on the element to create an `Opcode` object which can be added to the current group.

5.3.2.2 parseLine

This function converts a `<pre>` element representative of an opcode to an `Opcode` object. Rather than attempting to deduce which token in the string is the opcode name, another heuristic "trick" based on the manual formatting is used. This takes advantage of the fact that in the quick reference all opcode names are linked to their corresponding full manual page. Therefore the opcode name is surrounded by an HTML `<a>` tag which can be recognised. Once the opcode name is known, the current line can be split into two halves about that substring, resulting in strings known to contain the output and input parameters respectively. These parameter strings follow the same format and so both are passed off to the `parseParams` function for conversion to a list of `Parameter`s.

Functions are detected by the presence of parentheses, which causes a `FunctionOpcode` to be generated.

5.3.2.3 parseParams

This function interprets a string comma separated list of formal parameters with possible square brackets indicating optional parameters. Rather than attempting to parse the full

optional parameter structure properly (which is non-trivial as discussed in section 4.1.5) we use the regular expression `[A-Za-z0-9]+` to match parameter names whether or not they are optional. We also take note of where, if anywhere, the first occurrence of the opening square bracket `[` is seen. Then, parameter names which are seen after this location indicate optional parameters, and all other parameters are mandatory. With this information in hand it is straightforward to form up `Parameter` objects and return a list of them.

5.4 User Interface

The user interface is implemented using the `JGraph` component and the Swing windowing toolkit. The two important windows are the main `Editor` window, where the diagrammatic layout of instruments takes place, and the `DialogProperties` window where detailed editing of opcode parameters and comments is performed. Another key interface element is the `Instrument` class which provides the interactive diagram workspace.

Preliminary user interface designs can be found in Appendix A2.

5.4.1 Editor

The editor window is implemented as a standard `JFrame` utilising a `BorderLayout`. A screenshot is given in Figure 16. Use of this layout allows a toolbar at the top (in position `NORTH`) and leaves the remainder of the frame as an expanding area in which we place a set of tabs (`JTabbedPane`) representing the instruments in the orchestra. Inside each tab is a further `BorderLayout` with a text field accepting an instrument level comment in the `NORTH` position, and the remainder of the space taken by the "workspace" (an `Instrument` object). The `Instrument` is located inside a `JScrollPane` in order to allow scrolling when diagrams are larger than the screen size.

The menus and toolbars provide access to expected functionality such as adding/deleting instrument tabs, cut, copy paste, undo, redo and will not be discussed in detail apart from the **Insert** menu which is an important element of the project. It is based loosely on the design of the Crocodile Clips Add menu seen in the literature review Figure 8 (p25), in that it uses categories. These categories are obtained from the Csound manual as discussed above, and at the time of writing, result in the menu shown in Figure 15. Note the A-Z menu provided as the top submenu, and the option to show the properties window immediately for inserted items at the bottom.

Such a design provides the UI components to address the following requirements:

- Requirement 2: use of the resizable `JFrame` and `BorderLayout` allows the `Instrument` diagram editing area to expand to fill the space as the window is resized. The scroll bars allow scrolling as required.

- Requirements 3, 4, 14: The insert menu provides a categorised and alphabetical library of opcodes for insertion

- Requirement 16: The text field at the top of the instrument area (containing the text "This is a simple text instrument" in the example figure) allows comments to be entered for the instrument

- Requirement 12: Multiple tabs allows instruments to be collected as an orchestra

Other requirements may be seen to be satisfied in the screenshots, but the above list contains only those which directly concern UI components in the `Editor` frame.

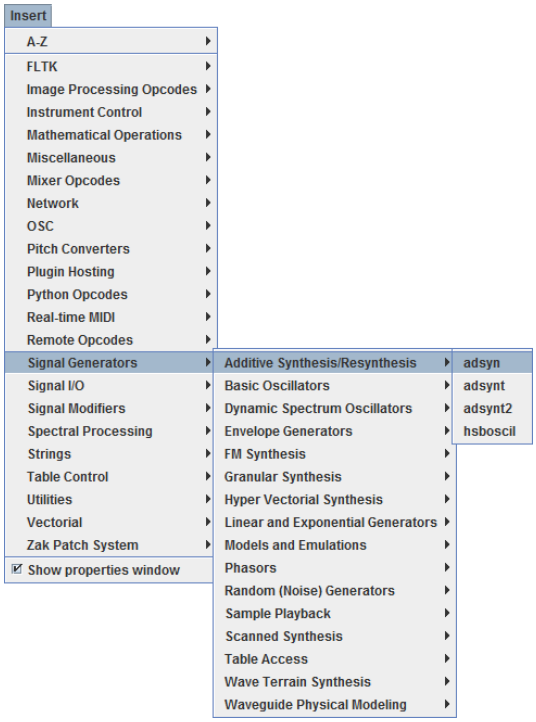


Figure 15: The insert menu

5.4.2 Instrument Workspace

The instrument workspace is the main area where opcode vertices can be placed, arranged, and connected. It is coded in the `Instrument` class, which extends `JGraph`. Using the functionality inherited from `JGraph`, it draws `OpcodeVertexViews`, `DefaultPortViews` and `DefaultEdgeViews` to represent the cells in the graph. The latter two are built into JGraph, but the former is written to incorporate some custom features which will be seen in a following section. Figure 16 shows three `OpcodeVertexViews` and some edges (lines) between ports (the small squares terminating each line and labelled with a name).

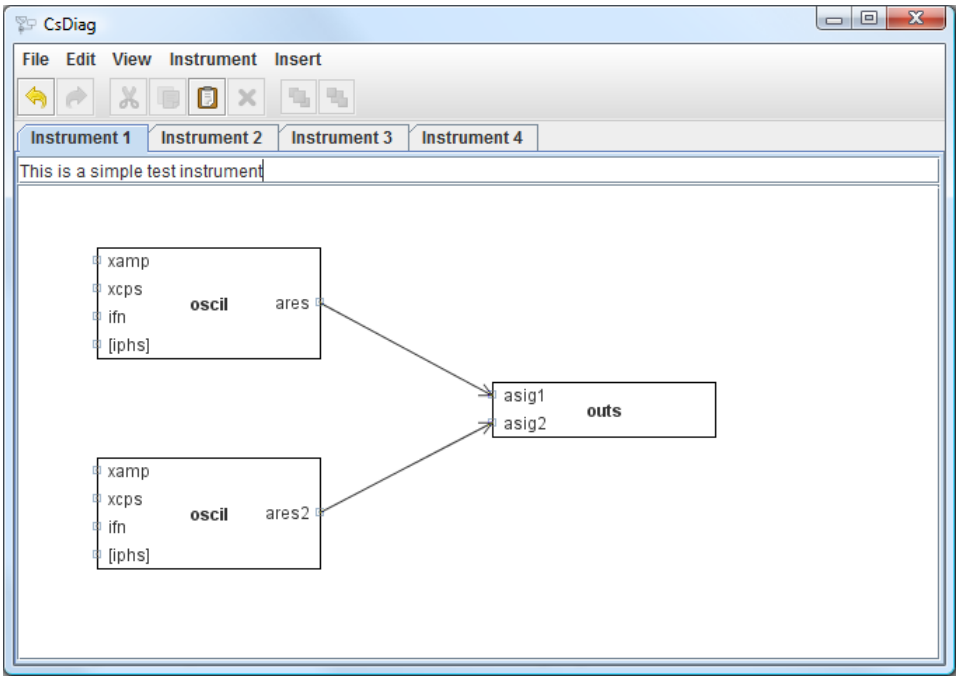


Figure 16: Appearance of the main Editor window

Multiple edges and vertices can be selected and deleted using the delete key or menu option. Vertices can be moved by dragging and any connected edges will also move to ensure they still terminate at the correct place to connect the input and output ports. Edges can be created by dragging from an output to an input.

The detail of what happens when edges are created and deleted, and the text that appears on the vertices, will be discussed later.

The following requirements are satisfied by these features:

- Requirement 2: The `Instrument` class provides a workspace area for constructing instruments
- Requirement 5: JGraph's built in functionality allows vertices to be selected and moved by drag and drop
- Requirement 8: Edges can be created by drag and drop

5.4.3 DialogProperties

The properties dialog box (the Americanized spelling is deliberately used for consistency with Java) allows editing of detailed properties of a given opcode vertex. It is defined in the

class `DialogProperties`. One instance of the dialog may be displayed for each opcode vertex in the orchestra and any number of them may be onscreen at any time.

Figure 17 below shows an example of the properties dialog for an `oscil` opcode in Instrument 1 in the orchestra (the window title reports this).

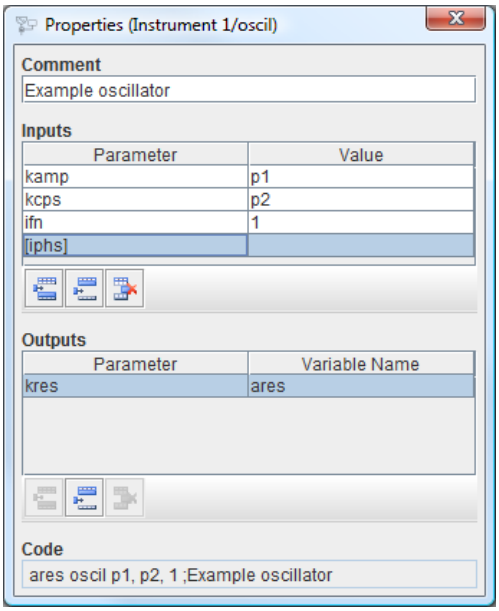


Figure 17: Example instance of the DialogProperties box

A `Hashtable` which maps `OpcodeVertex` objects to `DialogProperties` objects is maintained in `Editor`. The purpose of this is to record which `OpcodeVertex` objects on the diagram currently have their corresponding properties window open. When the menu option to show the properties window for an opcode vertex is invoked, an entry is made into the map. When the window is closed, it is removed. Maintaining such a map is useful for:

- Avoiding opening the properties window twice if the menu command is selected twice – in the second instance the correct window can just be given focus and brought to the foreground

- Ensuring that the corresponding properties window is closed if the opcode vertex is deleted

In terms of components actually present in the window (which is a `JDialog`-derived class), it is clear that there is a **comment** text field, to allow modification of the comment on the opcode. The two tables proposed in section 4.7 for editing the **input and output details** are present as `JTable` objects. These tables are based on custom table models (extending `AbstractTableModel`) which update the values on the opcode vertex directly when the

table is modified. The toolbar underneath each table allows addition and removal of optional parameter rows.

`DialogProperties` also registers itself as a Java `ChangeListener` on the `OpcodeVertex` it is concerned with. This allows it to update the text in the UI when the `OpcodeVertex` is changed as a result of another action outside of the dialog. For example, when the name of a connected output is changed in the properties for *that* vertex, the input expression on *this* vertex will update. The listener ensures that change is immediately visible in the UI.

Also note we have decided to include the generated code for the opcode vertex at the bottom of the dialog. This is a read only text field, and by the means discussed above (and listeners on all fields on the dialog itself) it automatically updates as input/output values are modified and connections are made/unmade on the diagram. This functionality was originally intended to display in a separate dialog box on demand, but code generation is sufficiently straightforward and fast that an always-visible field updated in real time is feasible.

The properties dialog addresses the following requirements:

- Requirement 7: Names of output variables can be changed by modifying the cells in the Variable Name column of the output table
- Requirement 9: Optional parameters can be added and removed from either of the tables using the supplied buttons to add/remove rows
- Requirement 11: Csound code for the current opcode vertex is displayed and updated in real time
- Requirement 13: Expressions can be entered into the Value column of the inputs table to assign them as the input value for that port (this to be further discussed in the following section)
- Requirement 16: Commentary on opcodes can be recorded using this dialog
- Requirement 19: Variables names entered into the table are validated (to be further discussed in the following section)

5.5 Vertices, Ports and their Views

As already seen, the class `OpcodeVertex` represents a vertex in the graph and an opcode statement in the instrument. It extends from JGraph's `DefaultGraphCell` which is the default implementation of a vertex. The `DefaultGraphCell` contains a list of *children*, which are the ports defined on that vertex. For this project, `OpcodeVertex` extends this implementation by storing two further lists of references to the same ports, in order to separate them into inputs and outputs. It also defines a field for a comment.

Once instantiated, an `OpcodeVertex` does not keep a reference to the `Opcode` it was based on – the name and parameters are copied to the vertex itself. This is necessary to allow user adjustment of the parameter list and facilitate features such as the input/output tables in `DialogProperties`. This also means that if opcodes are ever deprecated in future

versions of Csound and the internal catalogue described earlier is updated to remove them, old diagrams will still be editable since there will be no "broken" references.

The `OpcodeVertex` provides the logical function of an instance of an opcode. To visualise this, we use the `OpcodeVertexView` class. This contains, as an inner class, the `OpcodeVertexRenderer` which can draw the vertex onscreen. This uses the parent class to paint the vertex using the default implementation, then paints on labels for the ports in the correct location using the method `paintPortLabels()`. This results in vertices which appear similarly to the three shown in the earlier example in Figure 16.

The labels for inputs reflect the names of the formal parameters, the labels for outputs show the names of the actual output variables. Actual input values are not shown because this would clutter the interface – instead these can be inferred from the connecting lines, or viewed in the properties dialog.

The fact that vertices are visually rendered as described above satisfies requirement 6.

5.6 Expression Parsing and Connection

Let us now consider the implementation of the algorithms discussed in 4.7.2 concerning connection of edges based on an entered expression. Recall that this algorithm is in fact the *only* way that edges are created, since in reality the drag and drop method of drawing an edge only modifies the expression on the target port and then generates the edge based on that expression. Generation of edges from expressions will be discussed shortly but first the actions performed by `setValue` on `OpcodeInputPort` will be reviewed.

The `setValue` method on `OpcodeInputPort` is used to modify the expression or value for that input port. It is called by either the table model in `DialogProperties` for when the value is changed there by the user, or by the `connect` and `deleteCells` methods in `Editor` for when an edge has just been created by drag and drop or deleted from the workspace. `setValue` validates the variables in the expression, and then sets the new expression on the object.

5.6.1 extractVars

To extract the individual variables from an expression, the method `extractVars` is implemented. This applies the regular expression `[A-Za-z0-9_]+` to match possible candidates for being variables, then removes duplicate matches, matches which are function names (according to `FunctionOpcode.isFunction` which checks against the list of registered functions it gathers when loading `opcodes.xml` at startup) and matches which are just numbers. These variables can then be validated and used for connection, which will now be described.

5.6.2 Variable Validation

Validation of variables was designed in section 4.7.2. It is implemented in the `Variable` class which contains static methods for testing if variables are of particular types. These methods work by examining the variable name and matching it to various regular expressions similar to the one in the previous section. `setValue` extracts the variables from the supplied expression, and validates each one by calling `Variable.isValidLocal` and `Variable.isValidGlobal` to test validity of each scope. Variables which are not valid under one of these are added to an exception which is thrown at the end and results in a warning in the UI.

Variable validation for `setValue` can be skipped by specifying the second parameter of that method as false. This is for performance reasons when the program is calling `setValue` internally and all variables in the expression are known to be valid.

5.6.3 refreshConnections

A call to `setValue` is usually followed by a call to `refreshConnections` (which is also defined on `OpcodeInputPort`). `refreshConnections` first deletes all the current edges to the input port, then extracts the variables from the expression and attempts to locate output ports where these variables are set.

Deletion of connected edges is trivial and variable extraction has already been discussed. This leaves the location and connection of relevant outputs to be covered. This functionality is contained within the `makeConnections` method.

For each extracted variable, `makeConnections` calls `getNearestOutput` to obtain the nearest output port that sets the required variable. This method iterates through all ports which have the correct output variable, using the `portDistance` function to calculate the Euclidean distance to the port. The closest port is then returned and an edge created between it and the `OpcodeInputPort` whose connections are being refreshed.

A visual enhancement is also implemented in `makeConnections`, which is to use solid lines for the connecting edge when only one variable features in the expression and dashed lines when there are multiple edges. This makes it easier for the user to see when an output value is only forming part of an input and is being supplemented in some way.

This implementation of `refreshConnections`, and the fact that drag-and-drop connection creates edges by setting the input expression then calling this method produces an interesting feature. This is that if a user attempts to drag-connect an output to an input, and there is an intervening vertex closer to the input with the same output variable defined, the connection will be made to the output on this vertex instead. This is useful and not a bug because with a left to right code generation (discussed in 4.8) an intervening vertex which shadows the output name of another will result in an unexpected value at the input if the input is intended to be connected to a vertex that is further away. Having a drag-and-dropped

edge automatically moved to the nearer port will indicate the assignment which will actually be made, and the user can then rearrange the diagram if desired.

In the case where an output port for the requested variable name cannot be found anywhere on the diagram by `makeConnections`, its scope is checked using the validation methods discussed earlier. If it is a p-value from the score, it is ignored and a connection is not attempted. If it is global, it is checked for in the orchestra header (using a very primitive search which will unfortunately result in a positive result for any matching substring, including where it has been commented out); if found, it is ignored and not connected. In the case where it is missing from the header, or for missing local variables, an error is produced and displayed as a message box by the UI.

5.6.4 Edge Deletion

Manual deletion of edges by users is interesting in the case that the deleted edge forms part of an expression on the target input port. In 4.7.4 it was concluded that the best solution to this was to clear the entire expression (and hence delete all other edges terminating on the target) after first warning the user that this would happen. This functionality is implemented in the `deleteCells` method on `Editor` (which also deals with deletion of vertices, resulting in the deletion of any connecting edges which will be subject to the same treatment as discussed).

A consideration is that multiple cells may be deleted at once using a selection and so it is possible that even though multiple edges are connecting into a port, they have all been selected for deletion. In this case it is obvious to the user that the port will be left empty and so the expression is cleared without warning.

The code is sufficiently well commented that it should be easy to follow the deletion algorithm in an inspection of the listing for `Editor.deleteCells` in Appendix A4, therefore further details are not discussed here.

5.6.5 Edge Connection

Connection of edges presents a similar problem to be solved as in the case of deletion. That is, if there is already an expression present on the port that is the target of a connection, how should it be modified to incorporate the new variable being connected? This implementation uses the very simple method of appending the name of the connected variable to the end of the expression, separated by a "+" so that it is added to the existing value. The user could then rearrange the expression as they wished using the properties window.

A future improvement may be to present a small dialog box offering a choice of operator to use to join the new value with.

The mechanisms given in this section satisfy the following requirements:

- Requirement 8: Expression parsing backs the UI features introduced earlier to allow interactive connection of ports

- Requirement 13: Handling of expressions and the use of functions in those expressions is provided

- Requirement 19: Rudimentary validation of variable names is performed, and checks for presence of variables are made as part of the process of formatting connections. Appropriate warnings are given

5.7 Code Generation

Code generation was discussed in section 4.8, with the outcome that code would be generated based strictly on the left-to-right ordering of the vertices on screen. The straightforward part of code generation is based on the nested structure shown earlier in Figure 11 (p51). Each of the following classes implements a method `getCode` which operates as described here:

- `Orchestra` – gives the orchestra header followed by the result of calling `getCode` on each member instrument.

- `Instrument` – gives the `instr` opening statement followed by the instrument level comment, then the results of calling `getCode` on all member opcodes in order (to be discussed below) before finishing with `endin`.

- `OpcodeVertex` – gives the comma separated list of outputs, followed by the opcode name and the comma separate list of inputs. The `getCode` method for opcode allows a separator to be specified as an argument, with the default as the tab character. This is to allow a shorter separator for when the code is shown in `DialogProperties` and space is limited, but proper tab separation for outputted ORC files.

In order to obtain the correct ordering for the opcode level generations, the `CellViewComparator` class is implemented (as an inner class of `Instrument`). This compares the x position of two `CellView`s (`CellView` is a class from JGraph representing physical cells on the graph and encoding data such as location and size). By using a comparator, the standard Java sorting algorithm for sorting collections can be used to efficiently sort the `OpcodeVertex`es by x position, ready for generation in `Instrument.getCode`. This recognises non-functional requirement 26.

Implementation of code generation satisfies Requirement 1 (there is a method `saveOrcFile` in `Editor` which implements the final trivial step of writing the generated

code out to a file). It also underlies the UI features which satisfy Requirement 11, allowing real-time viewing of generated code at the opcode vertex level.

5.8 Serialisation

Serialisation and de-serialisation of the orchestra was intended to be implemented in the `serializeOrchestra` and `unserializeOrchestra` methods of the Editor (with the appropriate save and open dialog boxes presented beforehand by `loadDiagFile` and `saveDiagFile` to obtain a filename). However we will pre-empt the test results and comment here that implementation of this part of the system has so far been unsuccessful. When attempting to serialize some test orchestras, some very obscure errors are seen which are almost impossible to track down to a certain component of the system. Certainly it is not as simple as a named object not implementing the `Serializable` interface (necessary for an object to serialize in Java), or this would be easily remedied. It would appear that the way in which the JGraph API has been used has resulted in an arrangement of object references somewhere which causes the serialization to fail.

Due to project time constraints there was insufficient time to debug this fully and so the methods have been commented out. This means that requirement 10 has not currently been satisfied, with the consequence that this system has not yet met all its mandatory requirements and is missing a crucial part of the functionality in the code version created so far. Given sufficient time and JGraph expertise, it should be possible to either debug this or write a separate file format out manually without using serialization. The latter could use the JDOM XML library already linked to the application. This is left as further work to be undertaken.

5.9 Code Parsing and Import

This portion of the system (as designed in section 4.10) was not implemented due to time constraints imposed by the need to first implement the full diagramming tool with editing and code generation. Had there been a basic system already available (for example if Dia had proved more suitable for the task), more project resources could have been allocated to this part of the implementation.

It was decided therefore that this part would be excluded (since requirement 15 to which it relates is not mandatory) and indicated in section 6 as a future extension. Such future work would be relatively straightforward to carry out since we have:

1. Provided the previously missing foundation of a Csound diagramming tool so this work need not be repeated

2. Written it in a structured, object oriented style that facilitates easy extension and modification

3. Provided full API documentation

4. Suggested algorithms for crucial stages involved in this part of the system

5.10 Image Rendering

Requirement 22 was optional and a suggestion that rasterization of diagrams may be performed to allow users to save pictures of their diagrams for inclusion in documents etc.

This was found to be very straightforward to implement due to the inclusion of the `getImage` function in the `JGraph` class which performs this task. Therefore despite the optional nature of the requirement this feature has been added and the current diagram can be saved as Portable Network Graphics (PNG) using a menu option on the Instrument menu.

6 Testing and Evaluation

This section discusses the testing stage of the project and gives an evaluation of the implemented solution based on the results of the tests. The Introduction and Requirements sections present this as a mainly implementation-based project intended to produce a piece of software to solve the problem of diagrammatic Csound instrument construction. Therefore, testing is approached mainly as a requirements validation exercise.

Throughout its course, the project has introduced several new ideas surrounding diagram-to-code generation and so in a sense is also investigative. Focus on these investigative areas has led to the reprioritization of other aspects, with the consequence that some requirements have intentionally been left unsatisfied (for example Csound code import and some optional requirements). These requirements are therefore not tested, but instead have been discussed in terms of their effect on the value of the software product as a whole. It should, then, be noted that the program is not intended to be "perfect" finally developed product but instead is presented as a proof-of-concept foundation for future work in this area.

6.1 Testing Strategy and Plan

As mentioned above, the testing of the system is approached from a requirements validation perspective. That is, all tests are conducted in relation to a particular requirement, similarly to the way that the implementation documentation referenced the requirements that were satisfied at each stage. As a side-effect of performing the validation, verification of the functionality involved will occur and this will expose possible errors in the implementation.

These tests are **black box** tests in that they are conducted on the whole system in its completed state. This is considered appropriate because the system is intended as a single application and is not especially modular in design. Admittedly, high level black box testing cannot detect every error - in fact testing in general cannot prove the absence of errors because of its non-exhaustive nature in complex systems (35). What such testing does supply is a breadth-not-depth validation of the system requirements from a user perspective. As a consequence many parts of the system are tested simultaneously which increases the coverage of the tests for the same investment of time. Positive results in these tests indicate that the approach used is, in general, fit for purpose, which in this case where the solution is a prototype, is sufficient.

Informal **white box** testing of individual methods was carried out at implementation time to ensure expected results on a limited range of data. In the case of failure, the relevant code was debugged until a positive result was seen and so these tests have not been documented. In a production implementation, unit testing using a framework such as JUnit may be implemented to provide more formal verification of low levels. Knowledge of the internal

structure of the system was, however, used to inform the design of the black box tests in order to ensure that these tests rely on as much internal functionality as possible.

The test plan used can be found (completed with results) in Appendix A3. It covers each requirement identified in section 3, specifying a small number of tests for each that cover most possible situations relevant to the requirement. It can be seen that due to the overlap between requirements, some tests are dependent on a positive result from others and a positive result for these tests implies that other parts of the system are working. An example of this is where a value must be internally generated and then displayed in the UI. Black box verification of the internal generation must take place through the UI and so as a result both the internal function and the UI are tested together. Therefore, where requirements overlap, tests of the same feature may be conducted from the perspective of both requirements, explicitly in one and implicitly in the others.

Tests are grouped by specific parts of the system, using the groupings/work units identified in the summary of the requirements section:

- **Workspace and generic diagramming functionality** (Requirements 2, 5) Tests involve placement and movement of opcode vertices around the workspace, ensuring the window can be sized and scroll etc.

- **Opcode acquisition and selection** (Requirements 3, 4, 14, 18) Tests involve import of the manual to the internal catalogue, checking of the menus for accurate reproduction and categorisation, insertion of opcodes

- **Opcode details and connection** (Requirements 6, 7, 8, 9, 16) Tests involve editing of details using the properties dialog, and ensuring that connecting or disconnecting ports updates expressions correctly. Also ensuring that data displays on the actual diagram correctly.

- **Orchestra-level features** (Requirements 12, 17) Tests involve adding and removing instruments from the orchestra and modifying the orchestra header text

- **Expression entry and validation** (Requirements 13, 19) Tests address various aspects of variable name validation and will also cover some automated edge connection issues again to ensure that connection is attempted and warnings are generated for the correct types of variable

- **Code generation and export** (Requirements 1, 11) Tests address the correct transfer of data from the representation in the UI to outputted code, including correct ordering of the orchestra code

- **Image rendering** (Requirement 22) Tests will ensure that both diagrams which fit on the screen and those which do not are rendered correctly to bitmaps

6.2 Known Shortcomings of Prototype Implementation

The following requirements were known not to be satisfied fully, prior to testing (many of these were optional features in any case):

- Requirement 10, Saving and restoring of diagrams was not compliant, due to obscure bugs in the serialization methods. Debugging was attempted by temporarily restricting the serialization to just one Instrument object and then using the `DebuggingObjectOutputStream` class described in an article on http://crazybob.org/2007_02_01_crazyboblee_archive.html. There was initially limited progress, allowing blank diagrams to be serialized error-free, but work on this was deferred due to time constraints. The relevant implementation section has already suggested possible ways to remedy the problem.

- Requirement 15, importing of Csound code and automated layout of a diagram based on it. This was removed from scope due to project time constraints based on the fact that a general Csound diagramming foundation was not in place at the start of the project and therefore had to be implemented as extra work. However, this aspect was given detailed consideration in the design and recommended as future work.

- Requirement 20, opcode search, was not implemented as it was optional – the A-Z menu provides equivalent functionality requiring only a few more mouse clicks.

- Requirement 21, different geometric shapes for different opcodes was only an option and not implemented – this was considered more difficult to implement than it was valuable, although it should be noted that the opcode catalogue format developed is sufficiently extensible to allow storage of data about the type of shape that should be used for a particular opcode. Only the JGraph details of how to use an arbitrary shape for a vertex and ensure the ports line up remain to be understood before this can be implemented.

- Requirement 23 (again an optional feature), connection with Csound executable to verify code, not implemented, suggested as future work

- Requirement 24 (optional), user defined opcodes, not implemented – suggested as future work

6.3 Analysis of Results

The results of executing the test plan as documented in Appendix A2 will now be discussed. As expected, the majority of tests completed without issue, however there were a small

number of bugs detected. The focus will be on these negative results since positive results require no further action to satisfy requirements.

In addition to considering individual results, the opportunity will also be taken to reflect on possible implementation issues as a result of the design of algorithms etc. which are not detected by testing. Finally, non-functional aspects will be considered.

6.3.1 Port Display and Refresh

One particular area of concern surrounds update of the diagram display when values are changed in the properties window. This resulted in the failure of tests 3.3, 3.4 and 3.5, all of which reported that after changing an output variable in the properties dialog, no change was seen on the diagram until the respective opcode vertices were moved with the mouse. This would suggest that the issue is caused by the opcode vertex views not being repainted when a value changes. The line `cell.graph.repaint()` was added to the methods in `DialogProperties` that should change the diagram but this results in no change to behaviour. Therefore further investigation into the correct way to update the display in JGraph must be conducted to remedy this problem.

Similar problems are experienced when adding and removing ports for optional parameters. It would appear that the correct ports are added and removed as expected, but that the ports are not spaced correctly with the labels. Tests 3.4, 3.5, 3.6 concern this. Figure 18 below shows incorrect spacing of input ports relative to the labels after deletion of optional ports on `linseg`.

Figure 18: Incorrect spacing of ports relative to labels after modification

This is again likely to be the result of an incomplete understanding of JGraph's operation. The labels are drawn with custom code, whereas the ports themselves are created and assigned x and y locations before being passed to some JGraph code to draw (since JGraph must handle mouse clicks at the correct location). The function `distributePorts` on `OpcodeVertex` was implemented to attempt to address this, and does work to distribute the ports correctly when the opcode is *created*. However subsequent calls appear to have no effect so it is possible that JGraph ignores further changes to co-ordinates after object creation. A thorough investigation into the methods used by JGraph to render the ports would need to be conducted.

6.3.2 Opcode Catalogue Import

This section refers to the functionality tested in section 2 of the test plan where the manual is parsed to acquire opcode definitions and these are displayed on the menus. All tests were passed, however there were also some observations.

The first of these is that occasionally two categories of the same name appear to be created, for example "Table Control". A closer investigation of which opcodes and subcategories were being placed into which of the identically named categories revealed that the two category names are not exactly identical, one differing from the other by the presence of a trailing space character. This resulted in them being entered into the hash table separately at manual parse time. Attempts were made to modify `PageParser` to remove this space (using various trimming methods) but these failed to match the space. Further debugging revealed that the space was an HTML non-breaking space in the manual (the ` ` entity) and this appears not to be matched by the regular expression whitespace class. The JDOM XML parser has an option that prevents it expanding entities which should have solved the problem by allowing detection of the text ` ` but when this was activated no effect was observed and the space continued to appear. This problem can be rectified by manual editing of the generated `opcodes.xml` file after it has been generated; however, the problem of removing the spaces at parse-time remains a minor problem to be solved.

The second point of note was that in some places opcodes appear twice on the menu. This was quickly found to be because they appear twice in the manual, once for each combination of parameter variable *types* (for example for some opcodes the definition is given once for an a-rate result and again for a k-rate result). This is obviously redundant in our application since output variables can be renamed to whichever type the user wishes but other than that there are no ill-effects. The parsing method could be modified to remove duplicates easily enough in a future update.

Finally, it was noted that some of the A-Z opcode menus ran off the bottom of the screen, resulting in those opcodes later in the ordering being inaccessible. At which point a menu runs off the screen is obviously dependent on screen resolution and window position which is what makes this problem difficult. It is surprising that Java does not implement a scrollable menu automatically, but there seems no simple way to enable this. Therefore we propose setting an arbitrary limit on the number of items that may show in an A-Z list and then creating further submenus chained from this (entitled for example "More...") to contain any overflow. An alternative would be to abandon A-Z menus and implement the opcode text search feature mentioned in Requirement 20, as this would fulfil the same user requirement of being able to find a known opcode rapidly.

6.3.3 Deletion of Instruments

The failure highlighted by test 4.3 is a trivial bug and could be easily resolved by calling `Editor.closePropertiesFor()` on each vertex in the deleted instrument.

6.3.4 Graph Model and Code Generation Improvements

The current method of code generation and edge connection by physical vertex positioning works correctly and indeed has several advantages, including a clear way for the user to manipulate the generated code order and a visual guide to ensure variables are not inadvertently shadowing others of the same name.

However, it does not seem a particularly "clean" or scientific technique; not least because it relegates the graph structure to a purely visual function when it could in fact be used to supply variable dependency information or used to infer the order. Given further time to redesign this tool and the benefit of hindsight, it seems possible that a solution based more on the graph structure could be made to work.

Let us now consider alternative graph representations by examine the example of QuteCsound (19), which was not available during the design phase of this project. Figure 19 shows the graph view of the following code:

```
instr 1
  ivel     veloc
  kenv   madsr    0.001, 0.4, 0, 0
  anoise noise    20000*kenv*(ivel/127), 0
  anoise   butterhp anoise,4000
      outs     anoise, anoise
endin
```

It would appear then that the problem of graphically representing an instrument has been elegantly solved by QuteCsound.

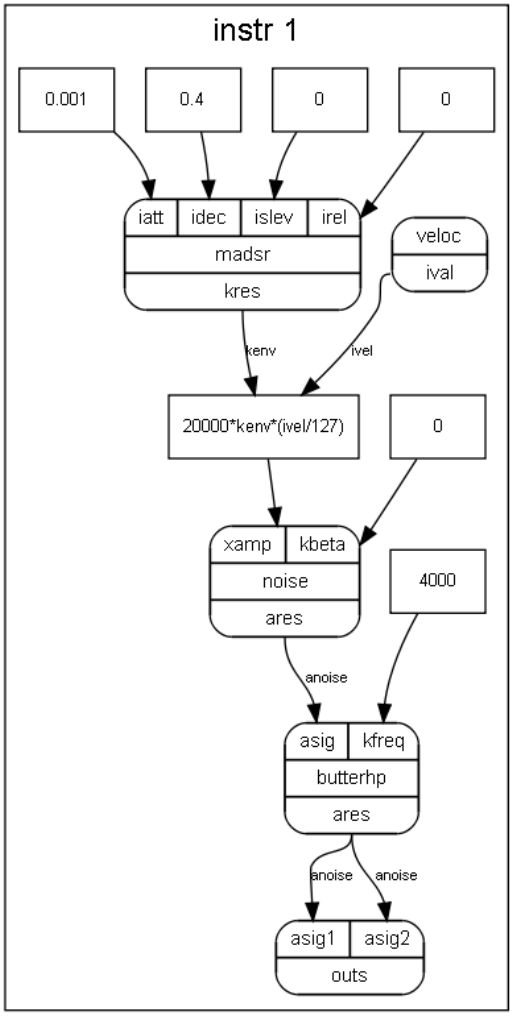


Figure 19: QuteCsound output showing a more complex graph with expressions and redeclarations

Expressions and values are recorded in vertices of their own before being connected into inputs, rather than being specified directly on the input as with our implementation. While this has the advantage that it allows all information to be displayed on the diagram rather than requiring a dialog to access it, it introduces an additional complication that opcodes-to-vertices is no longer a 1:1 mapping and there are now different types of vertex to deal with. It also suffers from the same potential problem that the set of variables used in an expression is encoded twice – once by way of the connected edges, and again as text in the expression string itself. It is possible in theory for these to become unsynchronised, and for there to be ambiguity in determining the correct edges from the expression, as seen. Note that these are not problems in QuteCsound because the graph is not intended to be interactively editable, but would become issues in trying to adapt QuteCsound's approach to interactive editing.

It can also be seen that the names of actual parameters are shown on the connecting edges themselves rather than on the source port. Although this does no harm, it is not strictly correct because it implies that two edges from the same port could have different variable names, and results in redundancy of information because this is not possible. This can be seen on Figure 19 with the variable `anoise` into `outs`.

It can be concluded from this additional analysis of a separate implementation that expressions are the source of weakness in the graph representation. They allow a many-to-many mapping between outputs and inputs where the rules to combine the edges terminating on an input are not defined in graph theory. The theory dictates that the only way to address this is to expand expressions fully to trees and make them part of the graph, which would result in clutter. This could perhaps be addressed with hiding or collapsing inner parts of the expression but is certainly complicated. Therefore, our method of parsing expressions and connecting the nearest available value is justified for practical purposes.

One final comment is that we have approached the overall problem from the perspective that the code which is output should be easily understandable and editable by a user. If we remove this requirement and permit the output of semantically correct but difficult to read code, the possibility of assigning **unique variables** for every output becomes apparent. This would remove the need to match expressions to the "nearest instance" of a given variable, which in turn would allow a code generation algorithm based on backward chaining/formal tree traversal.

6.3.5 Non-Functional Considerations

There were no specific tests designed to validate the non-functional requirements given in section 3.3. Instead we will summarise the system with respect to these here:

25. *The system must be delivered, documented, and tested for compliance with the requirements by the project hand in date of **27 April 2009**.*

    - This is an administrative requirement and at the time of writing work is on schedule to meet this deadline.

26. *Code generation must complete quickly to allow rapid adjustment of the diagram and regeneration of the code by the user. A reasonable average time would one second for each instrument. For individual opcodes there should be no discernible delay in generation and display of code.*

    - During execution of the functional test plan, no noticeable delays were experienced in the generation of code. The only delay seen was when inserting the first opcode vertex after the application starts – this causes a freeze of approximately 1 second while presumably the `OpcodeVertex`

and respective view classes are loaded into memory for the first time. This is not seen for subsequent vertex additions.

27. *The program must be able to operate in a cross-platform way, so that it is able to support the same operating systems as Csound itself.*

    - The program is written for Java which is available for all major operating systems. The compiled JAR files supplied are compatible only with Java SE 1.6 , but the source code will also compile on Java 1.5.

6.3.6 Development Model

In hindsight the use of an evolutionary prototype model was entirely satisfactory for development of software within the bounds of this project. It resulted in software which satisfied the majority of the requirements and so proved the intended concept, incorporating novel designs and algorithms devised during the development process as a result of previous work on the prototype. Such improvisation would not have been possible with a stricter model such as waterfall or an iterative process.

The software is however as a result, not directly suitable for production release and as commented earlier it would benefit from the prototype being thrown away and re-implemented. Such re-implementation should be done by someone with experience of JGraph, such as the author now that such experience has been acquired as part of the project.

6.4 Future Extensions

There has already been much discussion of improvements that would address most of the requirements which were not satisfied in this first prototype, and a detailed review of possible changes to the graph structure. These are obviously areas for future work but will not all be discussed again. Instead, descriptions are given here of other potential improvements for the application.

6.4.1 Online Help/Manual Pages

Many other Csound editing tools, including some of those identified in the literature incorporate the manual as a form of online help system. That is, the correct manual page can be called up for the opcode under the cursor in the event that the user is unsure of its function or usage. This would be an obvious addition to our software for similar reasons, and could be accessed by an extra item on the context menu for the selected opcode.

6.4.2 Saveable Groups/User Defined Opcodes (UDO)

User Defined Opcodes are groups of built-in opcodes that are connected together in some way and can be used as a single opcode, much like a function definition in procedural programming. They are defined in the orchestra header and can be called inside instrument blocks (12 p. 53). This is another feature that could be readily implemented in this diagramming tool, for example by allowing a selected set of opcodes in the instrument to be

saved as a UDO. At the very least, existing UDOs could be parsed from the header and presented for insertion into the diagram.

Arguably this is possible in the current implementation, since the user can access the orchestra header to add the UDO definition, and can modify the opcodes.xml file to put it on the menus. However this is obviously an inelegant approach and could be improved.

6.4.3 Control Widgets

When Blue was reviewed in section 2.4.4, it was seen how UI "widgets" such as knobs and sliders could be attached to control various parameters of the instrument. Also, Csound contains the FLTK category of opcodes for the same purpose. Although such controls to adjust parameter values is mainly intended for use in real-time performances, there may be some value in providing widgets for adjusting hardcoded instrument parameters in the diagram design view. On the whole, however, this does not seem an area of major interest, especially since it is likely to result in the incorrect impression that the widgets might be available in a real-time performance.

6.4.4 Code Verification/Auditioning with Csound

Something the current implementation does not contain at all is any communication with the Csound executable itself. When suitably extended, our solution could make use of such in a number of ways:

- Attempting compilation of the orchestra file at the user's request to detect semantic errors or missing variables
- Use of a preset score to audition the instruments on demand
- Use of MIDI to allow playing the instrument currently being constructed, for example with a controller keyboard

6.4.5 Writing into CSD files

Our current solution to code output is to write out the slightly outdated ORC file format. Such a file would traditionally be used alongside a SCO file containing the score to render a piece. However, the more recent trend is to use Csound Unified Files (CSDs) which contain the score and orchestra in the same file, in an XML-like container structure. It is simple to paste the code from an ORC file into as CSD, but an extension to our application could be automated writing of the orchestra portion of a CSD file. A point to bear in mind here is that it will often be desirable to export the orchestra section into an existing CSD file, which requires the format to be interpreted to some extent (to find the beginning and end of the `CsInstruments` section) and also the user to be warned if existing orchestra content will be overwritten.

6.4.6 SVG Output

We implemented the optional requirement concerning export of diagrams as images. However these are bitmap/raster graphics when diagrams are generally best represented as a vector format such as SVG. Therefore implementation of a means to export to this format is desirable. The JGraph manual (28 p. 99) discusses this in more detail.

7 Conclusions

The aim of this project was to implement a GUI facilitating interactive diagrammatic design and editing of Csound instruments. We will now consider the extent to which that problem has been addressed, and in what state this leaves the field of Csound front-ends in general.

The literature review identified a gap in the current provision of interactive front-ends, namely that there is no solution which allows a user to import the code for an existing Csound instrument, edit it using a diagram, and re-export it back to Csound code. In attempting to fill this gap, we have successfully implemented a modern, flexible and extensible GUI allowing interactive graphical design of instruments and code generation for the full set of Csound 5 opcodes.

The ability to generate diagrams from previously created orchestra code remains to be added, and this is suggested as a future project, now that a suitable foundation is in place. A very recent solution, QuteCsound, can generate diagrams from code – a feature not seen elsewhere in the literature review. However these diagrams are not interactive, so a convergence of these two functions still remains to be achieved. It is hoped that basis provided by this project will allow rapid implementation of such functionality.

We have covered novel diagram-to-code generation techniques and addressed the details and inherent problems of a diagrammatic representation that have resulted in limited progress elsewhere in the domain. The suitability of such methods has been critically evaluated, and as such this project has now provided some much needed modern research in the area. Although far from exhaustive, this is expected to be useful to those undertaking further work in this field.

Another original technique introduced is the parser for extracting opcode definitions from the manual, and it is due to this that we claim the solution is readily extensible for new opcodes, which need not even involve recompilation of the program. This is a major improvement over older solutions such as Patchwork which, had they been able to keep up with new opcodes in Csound, may have had a longer lifespan.

In terms of missing functionality, in addition to code import, the key feature of being able to save and load diagrams is absent. This limits the practical use of the application at this stage, which is not entirely unexpected for a prototype. However, the prototype serves as a good proof-of-concept for a modern interactive Csound diagram editing program and so has been successful and useful in most respects. For these and the remaining items of functionality intended as future work, we have offered suggestions or discussions which will provide a basis for either an immediate implementation or further research on the area.

The discussions in section 6 have given a critical review of the shortcomings of the prototype implementation, and this forms a strong foundation for future improvements. We reiterate the point made that use of an evolutionary prototype model was entirely satisfactory for this project, and resulted in software which proved the intended concept and incorporated novel designs and algorithms. However, as a result, the software is not directly suitable for production release and would benefit from the prototype being re-implemented.

To conclude, the project was successful in the sense that it has provided a significant and useful advance to the field of interactive diagrammatic instrument editors, which could be readily built upon to work towards a full solution suitable for use as production software.

8 Bibliography

1. Boulanger, Richard. Introduction to Sound Design in Csound. The Csound Book. Cambridge, Massachusetts : MIT Press, 2000, p. 5.

2. Frontends. Csound Website. [Online] [Cited: 13 October 2008.] http://www.csounds.com/frontends/.

3. Zmoelnig, Iohannes M. Pure Data. Pure Data. [Online] [Cited: 13 October 2008.] http://puredata.info/.

4. Vercoe, Barry L and Scheirer, Eric D. SAOL: The MPEG-4 Structured Audio Orchestra Language. Computer Music Journal. 1999, Vol. 23, 2.

5. Boulanger, Richard (Ed). The Csound Book. Cambridge, Massachusetts : MIT Press, 2000. 0262522616.

6. Kernighan, Brian D and Ritchie, Dennis M. The C Programming Language. Englewood Cliffs : Prentice Hall, 1978. 0-13-110163-3.

7. Wikipedia. Modular synthesizer. Wikipedia. [Online] 15 November 2008. [Cited: 16 November 2008.] http://en.wikipedia.org/w/index.php?title=Modular_synthesizer&oldid=251945911.

8. Visualising 1,051 Visual Programs - Module Choice and Layout in the Nord Modular Patch Language. Noble, James and Biddle, Robert. Sydney : Australian Computer Society, Inc., 2001.

9. Tutorials. Csound website. [Online] [Cited: 16 November 2008.] http://www.csounds.com/tutorials.

10. Gather, John-Philipp. Amsterdam Catalog of Csound Computer Instruments. Buffalo : University at Buffalo, 1995.

11. University of Florida. Courses. Florida Electroacoustic Music Studio/Computer Aided Music Instruction Laboratory. [Online] University of Florida. [Cited: 24 November 2008.] http://emu.music.ufl.edu/courses/#6445.

12. Vercoe, Barry et al. The Canonical Csound Reference Manual. Cambridge, Massachusetts : MIT, 2008.

13. Downloads. Csound Website. [Online] [Cited: 27 November 2008.] http://www.csounds.com/downloads/.

14. Csound News Archive. Csounds.com. [Online] [Cited: 27 November 2008.] http://csounds.com/news/archive/news06.html.

15. Yi, Steven. blue: a music composition environment for csound. 2008.

16. UTEMS. Patchwork. UTEMS. [Online] University of Texas at Austin. [Cited: 6 December 2008.] http://ems.music.utexas.edu/dwnld/.

17. Perry, Dave. Visual Orchestra. The Sonic Spot. [Online] [Cited: 6 12 2008.] http://www.sonicspot.com/visualorchestra/visualorchestra.html.

18. Gutsfeld, Sebastian. Cabel. Sourceforge. [Online] [Cited: 6 December 2008.] http://cabel.sourceforge.net/.

19. Cabrera, Andrés. QuteCsound. Sourceforge. [Online] [Cited: 11 April 2009.] http://qutecsound.sourceforge.net/.

20. Pinkston, Russell. An Introduction to Csound. [Online] [Cited: 5 December 2008.] http://ems.music.utexas.edu/program/mus329j/CSPrimer.pdf.

21. CsoundXML: A meta-language in XML for sound synthesis. Kröger, Pedro. Barcelona, Spain : s.n., 2004. International Symposium on Music Information Retrieval.

22. Kröger, Pedro. Desenvolvendo uma meta-linguagem para síntese sonora. Bahia : Universidade Federal da Bahia, 2004.

23. Dia. GNOME Live. [Online] GNOME Project. [Cited: 6 December 2008.] http://live.gnome.org/Dia.

24. Various. Using Dia for drawing out instrument diagrams. Nabble. [Online] Nabble. [Cited: 6 December 2008.] http://www.nabble.com/Using-Dia-for-drawing-out-instrument-diagrams-td19589773.html.

25. Henstridge, James. Dia. GNOME Live. [Online] [Cited: 6 December 2008.] http://projects.gnome.org/dia/custom-shapes.

26. Breuer, Hans. Re: python plugin and object properties. [Online] 8 July 2001. [Cited: 9 April 2009.] http://mail.gnome.org/archives/dia-list/2001-July/msg00054.html.

27. JGraph Ltd. JGraph - The Java Open Source Graph Drawing Component. JGraph. [Online] [Cited: 14 March 2009.] http://www.jgraph.com/.

28. Benson, David. JGraph and JGraph Layout Pro User Manual. Northampton : JGraph Ltd., 2008.

29. Graphviz. [Online] Graphviz. [Cited: 14 March 2009.] http://www.graphviz.org/.

30. Spitzak et al. Fast Light Toolkit (FLTK). [Online] 2008. [Cited: 6 December 2008.] http://www.fltk.org/.

31. Diestel, Reinhard. Graph Theory. s.l. : Birkhäuser, 2006. 3540261834.

32. Dawson, Christian. Projects in Computing and Information Systems. s.l. : Addison Wesley, 2005. 0321263553.

33. Sommerville, Ian. Software Engineering. 7th Edition. Harlow : Pearson Education, 2004. 0321210263.

34. Hunter, Jason. JDOM. [Online] [Cited: 13 April 2009.] http://www.jdom.org/.

35. Formal approaches to software testing. Petrenko, Alexandre and Ulrich, Andreas. Montréal : Springer, 2003. Third International Workshop on Formal Approaches to Testing of Software. 3540208941.

Appendices

A1 Extract of opcodes.xml File

The full file is ~10,000 lines so only an extract is included here. The first portion shows the header for the file, nested groups, opcodes and parameters. The second portion shows a further nested group. The final portion shown gives examples of functions.

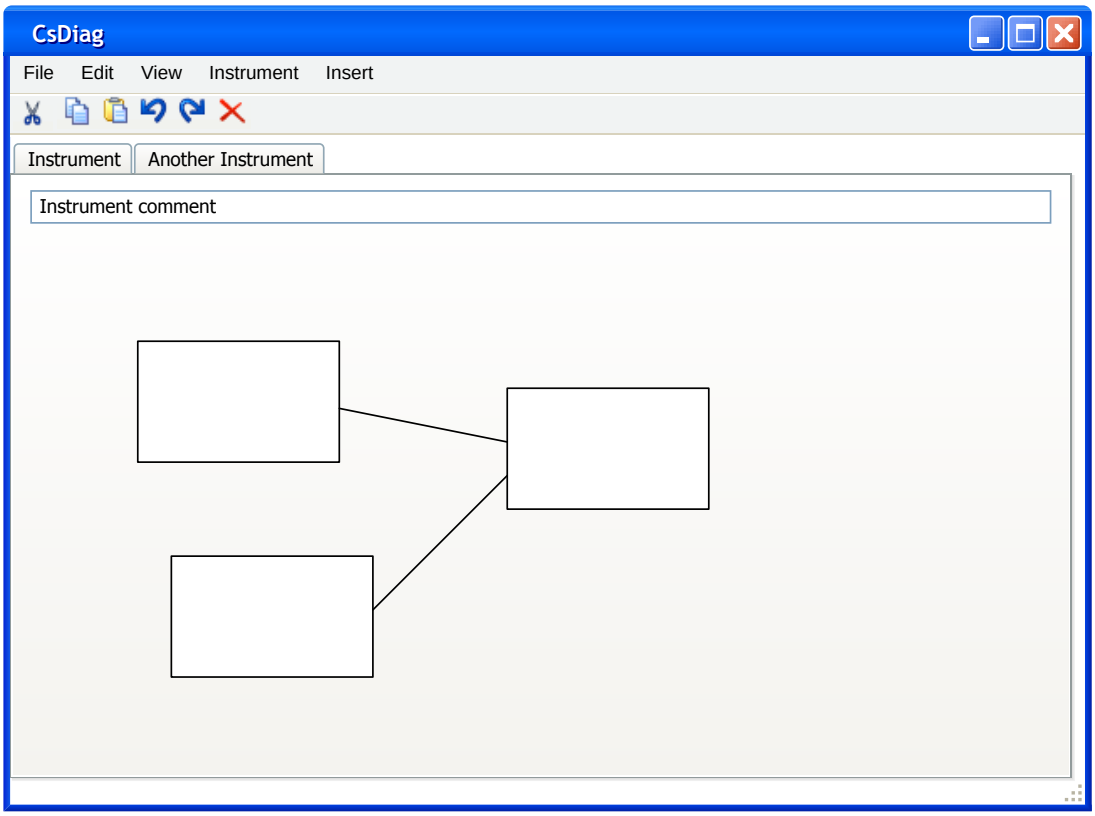
```
<?xml version="1.0" encoding="UTF-8"?>
<opcodes xmlns="http://people.bath.ac.uk/cjw26/csdiag/opcodes"
version="5.09">
  <group>
    <name>Plugin Hosting</name>
    <group>
      <name>VST</name>
      <opcode>
        <name>vstaudio</name>
        <input>instance</input>
        <input optional="yes">ain1</input>
        <input optional="yes">ain2</input>
        <output>aout1</output>
        <output>aout2</output>
      </opcode>
      <opcode>
        <name>vstaudiog</name>
        <input>instance</input>
        <input optional="yes">ain1</input>
        <input optional="yes">ain2</input>
        <output>aout1</output>
        <output>aout2</output>
      </opcode>

...

  <group>
    <name>Real-time MIDI</name>
    <group>
      <name>Note Output </name>
      <opcode>
        <name>midion</name>
        <input>kchn</input>
```

```
        <input>knum</input>
        <input>kvel</input>
      </opcode>
      <opcode>
        <name>midion2</name>
        <input>kchn</input>
        <input>knum</input>
        <input>kvel</input>
        <input>ktrig</input>
      </opcode>
      <opcode>
        <name>moscil</name>
        <input>kchn</input>
        <input>knum</input>
        <input>kvel</input>
        <input>kdur</input>
        <input>kpause</input>
      </opcode>

...

    <group>
      <name>Functions</name>
      <function>
        <name>cent</name>
      </function>
      <function>
        <name>cpsmidinn</name>
      </function>
      <function>
        <name>cpsoct</name>
      </function>
      <function>
        <name>cpspch</name>
      </function>
```

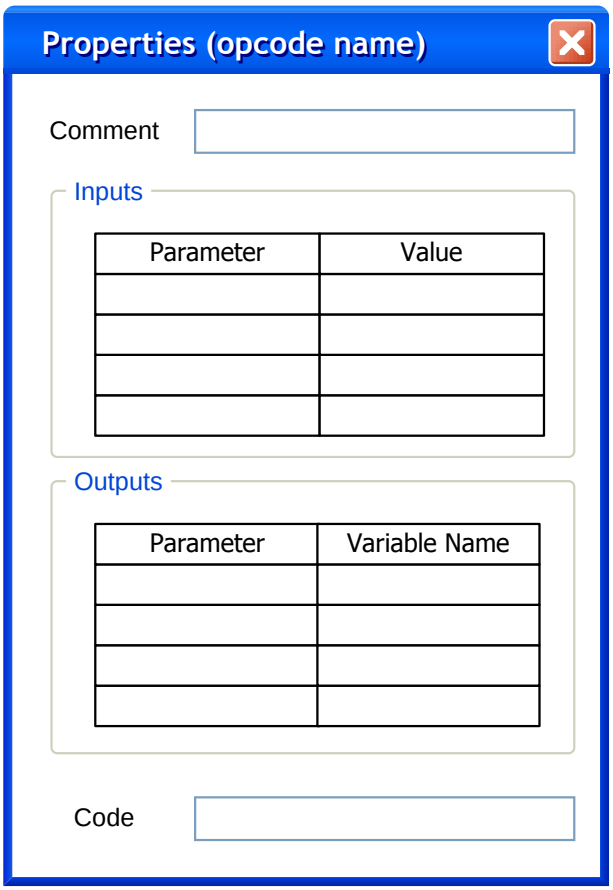
A2 User Interface Designs

These are preliminary drawings produced to design the user interfaces. As such there may be features missing from the drawings which are present in the finished prototype.

Main user interface mock-up showing menus, toolbar, tabs for instruments and the diagramming area:



Properties dialog box for editing individual opcodes, showing variable tables, space for comment and space for generated code:



A3 Test Plan and Results

The following test plan was created and then used to verify key parts of the system to ensure compliance with the requirements. Due to time constraints, exhaustive or low level testing was not feasible and the focus was on a higher level of testing which nonetheless covers the full breadth of the system.

Ref no	Related Requirement	Description/Expected Functionality	Result	Comments
1 Workspace and generic diagramming functionality				
1.1	5	Insert an opcode from the menu and click to select, opcode should highlight	Pass	
1.2	5	Perform test 1.1 and then click and drag the opcode, it should move with the mouse	Pass	
1.3	5	Perform test 1.1 action then click and drag one of the resize handles on the opcode vertex, this should resize it	Pass	
1.4	2	Perform test 1.2 and move the vertex offscreen to the bottom right, both scrollbars should appear and when dragged move the opcode back into view	Pass	
1.5	2	With the window in its restored state, drag the border or maximise it: the window should be resizable and the white workspace area should fill the available window space	Pass	
1.6	2	Insert a very large number of opcode vertices (~100): the diagram should retain them all	N/T	Requires automated means to test quickly
2 Opcode acquisition and selection				
2.1	3, 4, 18	Use the PageParser program to create an opcodes.xml file from the canonical Csound Manual which should be placed in the main CsDiag working directory. Start the main program and select the insert menu. Visually check for at least 20 top level categories. Check that the first and last categories from the web page have been imported. Check that category entries are submenus and contain items with opcode names.	Pass	Certain categories and opcodes appear twice, e.g. Table Control at the top level categories and oscil at the opcode level.
2.2	3	Randomly select 5 opcodes from 5 separate categories and click the menu item to insert. Opcodes should appear in the workspace.	Pass	
2.3	14	Use the A-Z submenu to insert 5 random opcodes from 5 different “letters”. Opcodes should appear in the workspace.	Pass	Some letters have very long menus resulting in some opcode choices being off the screen and thus not selectable.

3 Opcode details and connection				
3.1	-	Insert the oscil opcode and display the properties window for it using the menu. The window should display with the correct instrument and opcode in the title and with the tables populated with the correct input and output parameters.	Pass	Input and output parameters shown are consistent with the manual's definition
3.2	6	Perform test 3.1 and verify that the opcode name, formal input parameters and actual output parameters are shown on the diagram, and that they match with those showing in the properties dialog.	Pass	
3.3	6, 7	Perform test 3.2 then change the name of the output variable using the outputs table and inspect the diagram to ensure the label on the output port has changed.	FAIL	The text on the diagram does not update until a repaint is forced by moving the affected vertex.
3.4	9	Perform test 3.2 then use the table to add an extra optional input parameter at the end of the list. Check the diagram to ensure a port has been added.	FAIL	Port added but not spaced correctly. Label does not display until repaint as with 3.3
3.5	9	Perform test 3.2 then use the table to add an extra optional output parameter at the end of the list. Check the diagram to ensure a port has been added.	FAIL	Port added but not spaced correctly. Label does not display until repaint as with 3.3
3.6	9	Perform test 3.2 then delete the optional parameter *iphs*. Check that the diagram updates.	Pass	Labels are spaced correctly, but ports are not, after delete.
3.7	8, 13	Create two opcode vertices and move them apart from each other. For the right-hand one display the properties window and enter the name of the output of the left hand one into one of the inputs. A solid line should connect the two ports.	Pass	
3.8	8, 13	Perform test 3.7 and then add another vertex. Rename the output if it is the same as any existing output then modify the same input expression as used in 3.7 to include the new output port via some connecting operator. A connecting edge should be drawn to it and both lines should now be dashed.	Pass	
3.9	8	Create two opcode vertices and move them apart from each other. Display both property windows, and drag the output of the left hand one to the input on the right. A line should be drawn and the target input value that the line connects to should be set to the source output port.	Pass	

3.10	8	Perform test 3.9 and then insert and connect a further opcode by drag and drop. New opcode output should be appended to target expression with a +	Pass	
3.11	8	Connect the output of two opcodes to a third, then delete one of the connecting edges. A dialog should display with a warning, select no and no change should occur.	Pass	
3.12	8	Connect the output of two opcodes to a third, then delete one of the connecting edges. A dialog should display with a warning, select yes and the expression should clear, deleting both edges.	Pass	
3.13	8	Connect the output of two opcodes to a third, then select both connecting edges and delete them. No message should be displayed, the expression should clear and both edges should be deleted.	Pass	
3.14	8	Connect the output of two opcodes to a third, then delete the third opcode. Both connecting edges should also be deleted.	Pass	
3.15	8	Connect the output of two opcodes to a third, then delete one of the source opcodes. The warning dialog about edge deletion resulting in clearing of the expression should display.	Pass	
3.16	16	Insert an opcode and display the properties window. Enter a comment and close the window. Reopen the window and the comment should be shown again.	Pass	
3.17	7	Insert two opcode and connect one to the other. Modify the output variable name on the source opcode and ensure it updates the name on the input port of the other.	Pass	
3.18	-	Insert an opcode and open the properties window, then delete the opcode. The window should close.	Pass	
3.19	7	Test nearest-variable capture: Insert three opcodes, and set two to have identically named output variables. Arrange the opcodes in a row left to right with the two with identical outputs together on the left. Enter the name of those outputs into the input of the third opcode in the properties window. Should connect to the nearest output with the correct name.	Pass	

3.20	7	Test nearest-variable capture: Perform setup as per 3.19 then attempt to connect the output of the leftmost opcode to the input on the rightmost. The connection should actually be made between the second and third opcodes because the second opcode has the same output name and is closer.	Pass	
4 Orchestra-level features				
4.1	12	Use the instrument menu to add another instrument to the orchestra. Test that it can be selected and Instrument 1 can also still be restored by selecting the tab. Ensure opcodes can be added to both instruments	Pass	
4.2	12	Perform 4.1 and then delete each instrument in turn. The instrument tabs should disappear.	Pass	
4.3	12	Delete an instrument containing opcodes with open property windows. The windows should close.	FAIL	The window remains open
4.4	17	Use the menu to display the instrument header and edit the default that should display. Close the window then redisplay it and ensure the changes have been retained.	Pass	
5 Expression entry and validation				
5.1	13, 19	Create an opcode and attempt to set the output variable name to a valid name from each class in Table 1 (p19 of dissertation document) in turn. No errors should be seen.	Pass	
5.2	13, 19	Repeat 5.1 with another opcode present on the diagram. With each new output introduced, attempt to create a connection by entering it as an expression in an input on the other opcode. No errors should be seen if all names are valid.	Pass	
5.3	13, 19	Create three opcodes (referred to as A, B, C), with different output variable names. Arrange so that they do not overlap. This is a setup for subsequent tests.	Pass	
5.4	13, 19	Set up as per 5.3 and enter an expression A+B into C (where A and B are the relevant output variables). Connections should be created with dashed lines and no error.	Pass	
5.5	13, 19	Set up as per 5.3 and enter a more complex expression (A*A+B)/7 into C (where A and B are the relevant output variables). Connections should be created with no error.	Pass	

5.6	13, 19	Set up as per 5.3 and enter an expression involving a score parameter such as A+B+p5 into C (where A and B are the relevant output variables). Connections should be created for A and B with no error about p5 being nonexistent.	Pass	
5.7	13, 19	Set up as per 5.3 and enter an expression involving a function such as A+cpspch(B+p5) into C (where A and B are the relevant output variables). Connections should be created for A and B and there should be no error about cpspch being invalid.	Pass	
5.8	13, 19	Set up as per 5.3 and enter an expression involving a variable with an invalid name such as A+B+dsajlk into C (where A and B are the relevant output variables). Connections should be created for A and B and there should be two error messages – one about dsajlk being an invalid name and another about not being able to connect it.	Pass	
5.9	13, 19	Set up as per 5.3 and enter an expression involving a variable that is not defined but has a valid name that is not a function, such as A+B+kstuff into C (where A and B are the relevant output variables). Connections should be created for A and B and there should be one error message about not being able to connect kstuff.	Pass	
5.10	13, 19	Set up as per 5.3 and enter an expression involving only a score parameter such as p5 into C. No connections or error messages should be seen.	Pass	
5.11	13, 19	Set up as per 5.3 and enter an expression containing a valid but undefined global variable name such as gkstuff into C. An error message should be seen reporting that the variable is not defined.	Pass	
5.12	13, 19	Repeat 5.11 but before entering the expression, define gkstuff = 5 in the orchestra header. No error should then be seen on entry of the expression. No connections should be made either	Pass	
5.13	13, 19	Repeat 5.11 but before entering the expression, set the name of one of the outputs A or B to gkstuff. No error should then be seen on connection.	Pass	
6 Code generation and export				
6.1	1, 11	Test single opcode code generation: place one opcode and fill all inputs with constant expressions. Display properties and check the generation for correct reproduction and order of inputs	Pass	

6.2	1, 11	Test single opcode code generation: place one opcode and fill only **mandatory** inputs with constant expressions. Display properties and check the generation to ensure no excess separating commas	Pass	
6.3	1, 11	Test opcode comments: Enter comments in the Comment box of the properties window and check for appearance in the dynamic code generation	Pass	
6.4	1, 11	Test instrument code generation: For the default instrument 1, enter an instrument-level comment, place a single opcode and use the View > Instrument Source menu item to generate code for the instrument	Pass	
6.5	1, 11	Test code ordering: Place two different opcodes in a clear left-right order on the workspace. Generate instrument level code and check ordering correct. Reverse the order, regenerate, and check for a reversal in the code.	Pass	
6.6	1, 11	Test removal of statements: After executing test 4, delete one of the opcodes and regenerate code. Check that relevant statement has been deleted from code.	Pass	
6.7	1	Test multi-instrument orchestra code gen: Insert another instrument and ensure there are at least two opcodes on each instrument. Modify the default orchestra header, and generate **orchestra** code (to a file on disk). Check that both instruments and the header are included correctly.	Pass	Unix line endings are used for the orchestra output so the file does not appear correctly formatted in Windows notepad.
6.8	1	Test connected opcodes: In a new single instrument orchestra, place two different opcode and connect ones output to the other's input (with the correct left-right ordering of the opcode). Generate instrument code and check the actual output parameter has been substituted for the formal input parameter as expected.	Pass	
7 Image Rendering				
7.1	22	Construct a diagram contained entirely within the window, including multiple opcode vertices and connections between them. Render to an image and check for presence of all elements.	Pass	
7.2	22	Construct a diagram larger than the window (such that scrolling is required to edit it), including multiple opcode vertices and connections between them. Render to an image and check for presence of all elements.	Pass	

A4 Source Code Listings

This section lists the source code for the applications developed. Package and import statements have been omitted to save space. All classes belong to the `uk.ac.bath.cs.csdiag` class, except the last which is the `PageParser` and belongs to `uk.ac.bath.cs.csdiag.opcodeloader`. Classes are listed in alphabetical order within their package.

Only source files including interesting parts of the system or key algorithms are included. In particular the UI classes have been left out because they are long and present only routine user interface code. Full sources can be found on the CD.

FunctionOpcode

Opcode representation of single-input, single-output function

```
public class FunctionOpcode extends Opcode {
   /**
    * Registered functions
    */
   protected static Hashtable<String, FunctionOpcode> regfuncs = new
Hashtable<String, FunctionOpcode>();

   /**
    * Create a new FunctionOpcode
    *
    * @param symbol
    *            The name of the function
    */
   public FunctionOpcode(String symbol) {
      this.name = symbol;
      this.inputs.add(new Parameter("in", false));
      this.outputs.add(new Parameter("aout", false));
   }

   /**
    * Is the specified string a registered function?
    *
    * @param s
    *            String to test for functionness
    * @return Is it a function?
    */
   public static boolean isFunction(String s) {
      return regfuncs.containsKey(s);
   }

   public Element getXML() {
      Element el = new Element("function");

      // Set name element
      Element name = new Element("name");
      name.setText(this.getName());
      el.addContent(name);

      return el;
   }

   /**
    * Load a FunctionOpcode from an XML function element
    *
    * @param e
    *            The function element
    * @return The reconstructed function opcode
    */
   public static FunctionOpcode loadFromXML(Element e) {
      FunctionOpcode o = new FunctionOpcode(e.getChildText("name"));
      return o;
   }
}
```

Instrument

Graph representation of a Csound instrument.

```
public class Instrument extends JGraph {

   private static final long serialVersionUID = 5356019698991919162L;
   protected int instrNumber;
   protected String description = "";
   protected transient Orchestra orchestra;

   /**
    * Construct a new instrument graph
    *
    * @param instrNumber
    *            The unique instrument number
    * @param orchestra
    *            The orchestra to which the instrument belongs
    */
   public Instrument(int instrNumber, Orchestra orchestra) {
      this(new InstrumentGraphModel(), null);
      this.instrNumber = instrNumber;
      this.orchestra = orchestra;
   }

   /**
    * Construct the Graph using the Model as its Data Source
    *
    * @param model
    *            Model for graph data
    * @param cache
    *            Layout cache for storing layout information
    */
   public Instrument(GraphModel model, GraphLayoutCache cache) {
      super(model, cache);
      // Make Ports Visible by Default
      setPortsVisible(true);
      // Use the Grid (but don't make it Visible)
      setGridEnabled(true);
      // Set the Grid Size to 10 Pixel
      setGridSize(6);
      // Set the Tolerance to 2 Pixel
      setTolerance(2);
      // Accept edits if click on background
      setInvokesStopCellEditing(true);
      // Allows control-drag
      setCloneable(true);
      // Jump to default port on connect
      setJumpToDefaultPort(true);
      // Prevent JGraph's inline changes to cell names
      setEditable(false);
      // Anti-aliasing on
      setAntiAliased(true);
   }

   /**
    * @return Unique instrument number
    */
   public int getInstrNumber() {
      return instrNumber;
   }

   /**
    * @param instrNumber
    *            Unique instrument number
    */
   public void setInstrNumber(int instrNumber) {
      this.instrNumber = instrNumber;
   }

   /**
    * @return Descriptive comment about the instrument
    */
   public String getDescription() {
      return description;
   }

   /**
    * @param description
    *            Descriptive comment about the instrument
    */
   public void setDescription(String description) {
      this.description = description;
   }

   /**
    * Do code generation, strictly left to right across the visualised graph
    *
```

```
     * @return Generated code for this instrument
     */
    public String getCode() {
        // Begin instrument block
        String out = "instr " + this.getInstrNumber() + "\n";

        // If there's a comment add that, else don't leave just a ;
        if (!this.getDescription().equals(""))
            out += "; " + this.getDescription() + "\n";

        // Sort the cells by X position
        List<CellView> views = Arrays.asList(this.getGraphLayoutCache()
                .getCellViews());
        Collections.sort(views, new CellViewComparator());

        // For each cell
        for (CellView view : views) {
            if (DefaultGraphModel.isVertex(this.getModel(), view.getCell())) {
                // If it's a vertex, generate code and add to the overall
                // instrument code
                OpcodeVertex v = (OpcodeVertex) view.getCell();
                out += v.getCode() + "\n";
            }
        }

        // Terminate instrument block
        return out + "endin\n";
    }

    /**
     * Compare the X location of two cells
     *
     * @author Chris Ware
     */
    public class CellViewComparator implements Comparator<CellView> {
        public int compare(CellView arg0, CellView arg1) {
            double a = arg0.getBounds().getMinX();
            double b = arg1.getBounds().getMinX();
            return Double.compare(a, b);
        }
    }
}
```

Opcode

Definition of an opcode.

```
public class Opcode implements Comparable<Opcode> {
    protected String name;
    protected List<Parameter> inputs = new ArrayList<Parameter>();
    protected List<Parameter> outputs = new ArrayList<Parameter>();

    public Opcode() {
        name = "";
    }

    public Opcode(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String toString() {
        return outputs + " " + name + " " + inputs;
    }

    public List<Parameter> getInputs() {
        return inputs;
    }

    public void setInputs(List<Parameter> inputs) {
        this.inputs = inputs;
    }

    public List<Parameter> getOutputs() {
        return outputs;
    }

    public void setOutputs(List<Parameter> outputs) {
        this.outputs = outputs;
    }
```

```
/**
 * Load an opcode from an XML representation
 *
 * @param e
 *            XML Element to reconstruct to an opcode
 * @return Opcode extracted from the XML
 */
public static Opcode loadFromXML(Element e) {
   Opcode o = new Opcode(e.getChildText("name"));

   Iterator<?> it = e.getContent().iterator();
   while (it.hasNext()) {
      Element sub = (Element) it.next();

      // Try to check if a parameter is optional
      boolean optional;
      try {
         optional = sub.getAttribute("optional").getBooleanValue();
      } catch (Exception ex) {
         optional = false;
      }

      if (sub.getName() == "input") {
         o.getInputs().add(new Parameter(sub.getText(), optional));
      } else if (sub.getName() == "output") {
         o.getOutputs().add(new Parameter(sub.getText(), optional));
      }
      // Else it's probably the name - we don't care about that
   }

   return o;
}

/**
 * @return XML element representation of this opcode
 */
public Element getXML() {
   Element el = new Element("opcode");

   // Set name element
   Element name = new Element("name");
   name.setText(this.getName());
   el.addContent(name);

   // Add input parameters
   Iterator<Parameter> it = this.getInputs().iterator();
   while (it.hasNext()) {
      Parameter p = it.next();
      Element input = new Element("input");
      input.setText(p.getName());
      if (p.isOptional())
         input.setAttribute("optional", "yes");
      el.addContent(input);
   }

   // Add output parameters
   it = this.getOutputs().iterator();
   while (it.hasNext()) {
      Parameter p = it.next();
      Element output = new Element("output");
      output.setText(p.getName());
      if (p.isOptional())
         output.setAttribute("optional", "yes");
      el.addContent(output);
   }

   return el;
}

public int compareTo(Opcode o) {
   return (this.getName().compareTo(o.getName()));
}
}
```

OpcodeGroup

Possibly nested category of opcodes.

```
public class OpcodeGroup implements Comparable<OpcodeGroup> {
   private String name;
   private ArrayList<Opcode> members;
   private Hashtable<String, OpcodeGroup> subgroups;

   /**
    * Create an opcode group with the specified name
    *
    * @param name
    *            Context sensitive name for the opcode group
    */
   public OpcodeGroup(String name) {
      this.name = name;
      this.members = new ArrayList<Opcode>();
      this.subgroups = new Hashtable<String, OpcodeGroup>();
   }

   /**
    * @return Context sensitive name for this opcode group
    */
   public String getName() {
      return name;
   }

   /**
    * @param name
    *            Context sensitive name for this opcode group
    */
   public void setName(String name) {
      this.name = name;
   }

   /**
    * @return Member opcodes of this OpcodeGroup
    */
   public ArrayList<Opcode> getMembers() {
      return members;
   }

   /**
    * @return Subgroups of this OpcodeGroup
    */
   public Collection<OpcodeGroup> getSubgroups() {
      return subgroups.values();
   }

   /**
    * Add subgroup necessary for the supplied path
    *
    * @param path
    *            Array of descendant group names, including this group
    */
   public OpcodeGroup addSubgroup(String[] path) {
      if (path.length == 1)
         return this;

      OpcodeGroup child;
      String name = path[1].trim();
      if (!this.subgroups.containsKey(name)) {
         child = new OpcodeGroup(name);
         this.subgroups.put(name, child);
      } else {
         child = this.subgroups.get(name);
      }

      String[] nextpath = new String[path.length - 1];
      System.arraycopy(path, 1, nextpath, 0, path.length - 1);
      return child.addSubgroup(nextpath);
   }

   /**
    * Add a subgroup to this group
    *
    * @param o
    *            Subgroup to add
    */
   public void addSubgroup(OpcodeGroup o) {
      this.subgroups.put(o.getName(), o);
   }

   /**
    * Load a hierarchy of OpcodeGroups from an XML document
    *
    * @param file
```

```
     *            File to parse for groups
     * @return List of OpcodeGroups forming the roots of the hierarchy
     */
    public static List<OpcodeGroup> loadTreeFromXML(File file)
          throws JDOMException, IOException, FileNotFoundException {
       SAXBuilder parser = new SAXBuilder();
       parser.setIgnoringBoundaryWhitespace(true);
       Document doc = parser.build(file);
       return loadTreeFromXML(doc);
    }

    /**
     * Load a hierarchy of OpcodeGroups from an XML document
     *
     * @param d
     *            Document to load the groups from
     * @return List of OpcodeGroups forming the roots of the hierarchy
     */
    public static List<OpcodeGroup> loadTreeFromXML(Document d) {
       List<Content> groups = d.getRootElement().getContent();
       Iterator<Content> groupit = groups.iterator();
       ArrayList<OpcodeGroup> tree = new ArrayList<OpcodeGroup>();

       // Foreach group
       while (groupit.hasNext()) {
          Element group = (Element) groupit.next();
          OpcodeGroup o = loadFromXML(group);
          tree.add(o);
       }
       return tree;
    }

    /**
     * Load an OpcodeGroup and all children and subgroups from an XML element
     *
     * @param e
     *            XML element to load from - should be an opcodegroup element
     * @return The opcode group reconstructed from the XML
     */
    private static OpcodeGroup loadFromXML(Element e) {
       OpcodeGroup ocg = new OpcodeGroup(e.getChildText("name"));

       Iterator<Element> it = e.getContent().iterator();
       while (it.hasNext()) {
          Element sub = it.next();
          if (sub.getName() == "opcode") {
             // Opcode
             ocg.getMembers().add(Opcode.loadFromXML(sub));
          } else if (sub.getName() == "function") {
             // Function
             FunctionOpcode f = FunctionOpcode.loadFromXML(sub);
             // Put into the list of registered functions
             FunctionOpcode.regfuncs.put(f.getName(), f);
             ocg.getMembers().add(f);
          } else if (sub.getName() == "group") {
             // Nested group
             OpcodeGroup g = loadFromXML(sub);
             ocg.addSubgroup(g);
          }
          // Else it's probably the group name - we don't care about that
       }

       return ocg;
    }

    /**
     * @return The XML representation of this opcode group
     */
    public Element getXML() {
       Element el = new Element("group");

       Element name = new Element("name");
       name.setText(this.getName());
       el.addContent(name);

       Iterator<OpcodeGroup> itg = this.getSubgroups().iterator();
       while (itg.hasNext()) {
          el.addContent(itg.next().getXML());
       }

       Iterator<Opcode> ito = this.getMembers().iterator();
       while (ito.hasNext()) {
          el.addContent(ito.next().getXML());
       }

       return el;
    }
```

```
   public int compareTo(OpcodeGroup g) {
      return this.name.compareTo(g.getName());
   }

   /**
    * Flatten the tree structure and return all descendant member opcodes
    *
    * @return A list of descendant opcodes
    */
   public ArrayList<Opcode> flatten() {
      ArrayList<Opcode> ops = new ArrayList<Opcode>();
      ops.addAll(this.members);
      Iterator<OpcodeGroup> it = this.getSubgroups().iterator();
      while (it.hasNext()) {
         ops.addAll(it.next().flatten());
      }
      return ops;
   }
}
```

OpcodeInputPort

A logical input port on an opcode vertex.

```
public class OpcodeInputPort extends OpcodePort {
   private static final long serialVersionUID = -8066552230485897724L;
   protected String value = "";

   public OpcodeInputPort(Parameter p) {
      super(p);
   }

   public OpcodeInputPort(String name) {
      super(name);
   }

   public String getValue() {
      return value;

   }

   /**
    * Set the value of this input parameter
    *
    * @param value
    *            Value expression
    * @param checkValid
    *            Should this be checked for validity
    * @throws InvalidVariablesException
    *             Thrown if checking is on and variables are misnamed or
    *             globals are missing - value is always set
    */
   public void setValue(String value, boolean checkValid)
         throws InvalidVariablesException {
      this.value = value;
      this.getVertex().fireChange();
      if (checkValid) {
         InvalidVariablesException e = new InvalidVariablesException();
         // For each attempted variable check validity
         for (String v : extractVars(value)) {
            // Validate
            if (!Variable.isValidLocal(v) && !Variable.isValidGlobal(v))
               // If it's not global and not a valid local, add to invalid
               // list
```

```
            e.invalid.add(v);
      }
      if (e.hasInvalid())
         // If the exception accumulated any bad things, throw it
         throw e;
   }
}

/**
 * Force connections to all outputs mentioned in value
 */
protected void makeConnections() throws RuntimeException {
   List<String> vars = extractVars(this.value);
   List<String> missingVars = new ArrayList<String>();

   // For each variable
   for (String var : vars) {
      // If it's not a function
      if (shouldConnect(var)) {

         // Find the nearest source
         OpcodeOutputPort source = getNearestOutput(var);

         if (source != null) {
            // If it's multiple inputs, use dashed lines
            boolean dashed = (vars.size() > 1);

            // Connect source
            source.connectTo(this, dashed);

            // It's missing but if it's global check for it in the
            // header
         } else if (!Variable.isValidGlobal(var)
               || (Variable.isValidGlobal(var) && !this.getVertex()
                     .getInstrument().orchestra.getHeader()
                     .contains(var))) {
            missingVars.add(var);
         }
      }
   }

   // Were there any missing variables?
   if (missingVars.size() > 0) {
      String s = "";

         for (String mv : missingVars)
            s += (mv + "\n");

         throw new RuntimeException(
               "The following connectable variables were not found as an
output:\n"
                     + s
                     + "\nNo connections for these have been added to the
diagram.");
      }
   }

   /**
    * Get the nearest output port to this one with a certain user variable
    * string
    *
    * @param userVar
    *            Variable name to search for
    * @return The nearest instance of that variable on an output port, or
null
    *         if there are none
    */
   protected OpcodeOutputPort getNearestOutput(String userVar) {
      PortView[] ports = this.getVertex().getInstrument()
            .getGraphLayoutCache().getPorts();
      PortView thisPort = (PortView) this.getVertex().getInstrument()
            .getGraphLayoutCache().getMapping(this, false);

      PortView closest = null;
      double closestDist = 0;

      for (int i = 0; i < ports.length; i++) {
         // For all opcode output ports
         if (ports[i].getCell() instanceof OpcodeOutputPort) {
            OpcodeOutputPort port = (OpcodeOutputPort) ports[i].getCell();
            // That aren't on the same vertex as this one
            if (port.getVertex() != this.getVertex()) {
               // And that have the right output variable name...
               if (((OpcodeOutputPort) ports[i].getCell()).getUserVar()
                     .equals(userVar)) {
                  // Decide it it's closer
                  double distance = portDistance(thisPort, ports[i]);
                  if (closest == null || distance < closestDist) {
                     closest = ports[i];
```

```
                    closestDist = distance;
                }
            }
        }
    }
    if (closest != null)
        return (OpcodeOutputPort) closest.getCell();
    else
        return null;
}

/**
 * Work out the distance between two port views
 *
 * @param a
 *            First port view
 * @param b
 *            Second port view
 * @return Euclidean distance between ports
 */
public static double portDistance(PortView a, PortView b) {
    return Point2D.distance(a.getLocation().getX(),
a.getLocation().getY(),
            b.getLocation().getX(), b.getLocation().getY());
}

/**
 * Destroy all connections into this port
 */
protected void destroyConnections() {
    this.getVertex().graph.getGraphLayoutCache().remove(
            this.getEdges().toArray());
}

/**
 * Redraw the connections to reflect the current expression
 *
 * @throws RuntimeException
 *             if there are variables that cannot be located for a
 *             connection
 */
public void refreshConnections() throws RuntimeException {
    this.destroyConnections();
    this.makeConnections();
}

/**
 * Extract all variables mentioned in the value expression to a list
 *
 * @return List of variable names
 */
protected List<String> extractVars(String expr) {
    ArrayList<String> vars = new ArrayList<String>();

    // Iterate through the matches to this regex
    Pattern param = Pattern.compile("[A-Za-z0-9_]+");
    Matcher m = param.matcher(expr);
    while (m.find()) {
        // It's a new variable if unique, not a function and not just a
        // number
        if (!vars.contains(m.group()) && !m.group().matches("[0-9]+")
                && !FunctionOpcode.isFunction(m.group()))
            vars.add(m.group());
    }
    return vars;
}

/**
 * Check if connection to an output should be attempted for a particular
 * variable
 *
 * @param var
 *            Variable to check eligibility for connection for
 * @return Should connection be attempted?
 */
protected static boolean shouldConnect(String var) {
    // Connect variables that are not functions or p variables
    boolean attempt = !FunctionOpcode.isFunction(var);
    attempt = attempt && !Variable.isP(var);
    return attempt;
}
}
```

OpcodeOutputPort

A logical output port on an opcode vertex.

```
public class OpcodeOutputPort extends OpcodePort {
   private static final long serialVersionUID = -1147611910721457965L;
   protected String userVar = "";

   /**
    * Create a new output port
    *
    * @param p
    *            The parameter from the model that it represents
    */
   public OpcodeOutputPort(Parameter p) {
      super(p);
      userVar = p.getName();
   }

   public OpcodeOutputPort(String name) {
      super(name);
      userVar = name;
   }

   /**
    * Get the user's actual output variable
    *
    * @return User specified output variable name
    */
   public String getUserVar() {
      return userVar;
   }

   /**
    * Set the user's actual output variable name
    *
    * @param newUserVar
    *            User defined output variable name
    * @throws RuntimeException
    *             if variable name is invalid
    */
   public void setUserVar(String newUserVar) throws RuntimeException {
      // Validate
      if (!Variable.isValidLocal(newUserVar)
            && !Variable.isValidGlobal(newUserVar)) {
         throw new RuntimeException(newUserVar
               + " is not a valid Csound variable name");
      } else {
         // Rename on opposite ports - already validated so don't check
again
         OpcodePort[] opposites = this.getOppositePorts();
         for (int i = 0; i < opposites.length; i++) {
            OpcodeInputPort input = (OpcodeInputPort) opposites[i];
            input.setValue(input.getValue().replaceAll(this.userVar,
                  newUserVar), false);
         }

         this.userVar = newUserVar;
         this.getVertex().fireChange();
      }
   }

   /**
    * Get the output name
    */
   public String getValueName() {
      return userVar;
   }

   public void connectTo(OpcodePort target, boolean dashed) {
      super.connectTo(target, dashed);

      // If it's connecting to an input port (which it should do
      // according to model), update the value string
      /*
       * if (target instanceof OpcodeInputPort) { OpcodeInputPort in =
       * (OpcodeInputPort)target; if (in.getValue().equals(""))
       * in.setValue(this.getUserVar()); else in.setValue(in.getValue() +
"+" +
       * this.getUserVar()); }
       */
   }
}

/**
 * Port on an OpcodeVertex to allow connection of parameters to other ports
 *
 * @author Chris Ware
```

```
 */
```

OpcodePort

Superclass for ports used to connect opcodes together.

```
public abstract class OpcodePort extends DefaultPort {
   // The parameter it represents - no need to serialize
   protected transient Parameter parameter = null;

   // Details about the parameter - may match the parameter above or not, if
   // user specified
   protected String formal = "";
   protected boolean optional = false;

   /**
    * Create an opcode port based on a formal parameter from an opcode
    * definition
    *
    * @param parameter
    *            The parameter it represents
    */
   public OpcodePort(Parameter parameter) {
      super(parameter.getName());
      this.parameter = parameter;
      this.formal = parameter.getName();
      this.optional = parameter.isOptional();
   }

   /**
    * Create a custom opcode port without an associated formal parameter
    *
    * @param name
    *            The name of the formal parameter
    */
   public OpcodePort(String name) {
      super(name);
      this.parameter = null;
      this.formal = name;
      this.optional = true;
   }

   /**
    * @return Any ports connected to this one
    */
   public OpcodePort[] getOppositePorts() {
      GraphModel m = getVertex().getInstrument().getModel();
      Object[] edges = DefaultGraphModel.getEdges(m, new Object[] { this })
            .toArray();
      OpcodePort[] opp = new OpcodePort[edges.length];

      for (int i = 0; i < edges.length; i++) {
         opp[i] = (OpcodePort) DefaultGraphModel.getOpposite(m, edges[i],
               this);
      }

      return opp;
   }

   /**
    * @return The vertex this port belongs to
    */
   public OpcodeVertex getVertex() {
      return (OpcodeVertex) this.getParent();
   }

   /**
    * @return The Parameter this port represents or null if it was a user
    *         defined parameter
    */
   public Parameter getParameter() {
      return parameter;
   }

   /**
    * Make a connection to the target port
    *
    * @param target
    *            Target port to connect to
    */
   public void connectTo(OpcodePort target) {
      connectTo(target, false);
   }

   /**
    * Make a connection to the target port, with a possibly dashed line
    *
    * @param target
```

```
     * @param dashed
     */
    public void connectTo(OpcodePort target, boolean dashed) {
       DefaultEdge edge = new DefaultEdge();
       GraphModel model = this.getVertex().getInstrument().getModel();

       if (model.acceptsSource(edge, this)
             && model.acceptsTarget(edge, target)) {
          GraphConstants.setLineEnd(edge.getAttributes(),
                GraphConstants.ARROW_SIMPLE);
          if (dashed)
             GraphConstants.setDashPattern(edge.getAttributes(),
                   new float[] { 5, 5 });
          getVertex().getInstrument().getGraphLayoutCache().insertEdge(edge,
                this, target);
       }
    }

    /**
     * @return Name of the formal parameter
     */
    public String getFormal() {
       return formal;
    }

    /**
     * @return Is this an optional parameter
     */
    public boolean isOptional() {
       return optional;
    }
}
```

OpcodeVertex

Graph vertex to represent an instantiation of a Csound opcode.

```
public class OpcodeVertex extends DefaultGraphCell {

   private static final long serialVersionUID = -6677403453502084141L;
   protected transient Instrument graph;
   protected ArrayList<OpcodeInputPort> inputs = new
ArrayList<OpcodeInputPort>();
   protected ArrayList<OpcodeOutputPort> outputs = new
ArrayList<OpcodeOutputPort>();
   protected String comment = "";
   EventListenerList changeListeners = new EventListenerList();

   /**
    * Create an opcodecell with specified name, inputs, outputs
    *
    * @param name
    * @param inputs
    * @param outputs
    */
   public OpcodeVertex(String name, Instrument graph, List<Parameter>
inputs,
         List<Parameter> outputs) {
      super(name);

      this.graph = graph;

      Iterator<Parameter> it = inputs.iterator();
      while (it.hasNext()) {
         this.inputs.add(new OpcodeInputPort(it.next()));
      }
      it = outputs.iterator();
      while (it.hasNext()) {
         this.outputs.add(new OpcodeOutputPort(it.next()));
      }

      addInputPorts();
      addOutputPorts();
      distributePorts();

      double height = Math.max(inputs.size(), outputs.size()) * 20;
```

```
   AttributeMap map = new AttributeMap();

   GraphConstants.setBorderColor(map, Color.black);
   GraphConstants.setBackground(map, Color.white);
   GraphConstants.setBounds(map, new Rectangle2D.Double(20, 20, 160,
         height));

   this.getAttributes().applyMap(map);
}

/**
 * Create a new opcode vertex from the specified opcode
 *
 * @param o
 *            The opcode template to create it from
 * @param graph
 *            The graph the vertex belongs to
 */
public OpcodeVertex(Opcode o, Instrument graph) {
   this(o.getName(), graph, o.getInputs(), o.getOutputs());
}

/**
 * Add ports for the Opcode input parameters
 */
private void addInputPorts() {
   for (OpcodeInputPort p : inputs)
      this.add(p);
}

/**
 * Add ports for the Opcode output parameters
 */
private void addOutputPorts() {
   for (OpcodeOutputPort p : outputs)
      this.add(p);
}

/**
 * Distribute the ports evenly along the sides of the vertex
 */
public void distributePorts() {
   int count = 0;
   float space = 0;
   if (inputs.size() > 0)
      space = GraphConstants.PERMILLE / inputs.size();
   for (OpcodeInputPort p : inputs) {
      Point2D point = new Point2D.Double(0, count * space + space / 2);
      GraphConstants.setOffset(p.getAttributes(), point);
      count++;
   }

   count = 0;
   if (outputs.size() > 0)
      space = GraphConstants.PERMILLE / outputs.size();
   for (OpcodeOutputPort p : outputs) {
      Point2D point = new Point2D.Double(GraphConstants.PERMILLE, count
            * space + space / 2);
      GraphConstants.setOffset(p.getAttributes(), point);
      count++;
   }

}

/**
 * Get the opcode lvalues in Csound format
 *
 * @return Full outputs string based on user defined variables
 */
public String getUserOutputString() {
   String out = "";
   for (OpcodeOutputPort p : outputs) {
      out = out + ", " + p.getUserVar();
   }
   // Cut the last comma off, if there is one
   return (out.length() > 2) ? out.substring(2) : "";
}

/**
 * Get a list of input ports on this opcode
 *
 * @return Ordered list of input ports
 */
public ArrayList<OpcodeInputPort> getInputs() {
   return inputs;
}
```

```
/**
 * Get a list of output ports on this opcode
 *
 * @return Ordered list of output ports
 */
public ArrayList<OpcodeOutputPort> getOutputs() {
   return outputs;
}

/**
 * Get the instrument this vertex is in
 *
 * @return The parent instrument
 */
public Instrument getInstrument() {
   return graph;
}

/**
 * Generate source for this opcode using the default separator (tab)
 *
 * @return Csound source code for the opcode
 */
public String getCode() {
   return getCode("\t");
}

/**
 * Generate source for this opcode
 *
 * @param sep
 *            Separator for inputs, opcode name and outputs
 * @return Csound source code for the opcode
 */
public String getCode(String sep) {
   String inputValues = "";

   for (int i = 0; i < this.getChildCount(); i++) {
      Object port = this.getChildAt(i);
      if (port instanceof OpcodeInputPort) {
         String val = ((OpcodeInputPort) port).getValue();
         if (val.length() > 0)
            inputValues = inputValues + ", " + val;
      }
   }
   // Lose the additional comma-space from before the first item
   if (inputValues.length() > 2)
      inputValues = inputValues.substring(2);

   String outputs = this.getUserOutputString();
   if (outputs.length() > 0)
      outputs = outputs + sep;

   String comment = this.comment;
   if (comment.length() > 0)
      comment = sep + ";" + comment;

   return sep + outputs + this.toString() + sep + inputValues + comment;
}

/**
 * Add a custom input at the end
 *
 * @param name
 */
public void addInput(String name) {
   OpcodeInputPort n = new OpcodeInputPort(name);
   inputs.add(n);
   this.add(n);
   distributePorts();
}

/**
 * Add a custom input before the specified item
 *
 * @param name
 * @param position
 */
public void addInput(String name, int position) {
   OpcodeInputPort n = new OpcodeInputPort(name);
   inputs.add(position, n);
   this.add(n);
   distributePorts();
}

/**
 * Add a custom output at the end
 *
```

```
     * @param name
     */
    public void addOutput(String name) {
       OpcodeOutputPort n = new OpcodeOutputPort(name);
       outputs.add(n); // Add to vertex output list
       this.add(n); // Add to ports
       distributePorts();
    }

    /**
     * Add a custom output before the specified item
     *
     * @param name
     * @param position
     */
    public void addOutput(String name, int position) {
       OpcodeOutputPort n = new OpcodeOutputPort(name);
       outputs.add(position, n);
       this.add(n); // It's not adding properly
       distributePorts();
    }

    /**
     * Remove an optional input
     *
     * @param index
     */
    public void removeInput(int index) {
       // Remove from children
       this.remove(inputs.get(index));
       // Remove from typed port list
       inputs.remove(index);
       // Redistribute remaining ports
       distributePorts();
    }

    /**
     * Remove an optional output
     *
     * @param index
     */
    public void removeOutput(int index) {
       // Remove from children
       this.remove(outputs.get(index));
       // Remove from typed port list
       outputs.remove(index);
       // Redistribute remaining ports
       distributePorts();
    }

    /**
     * Get the comment for the opcode
     *
     * @return The comment
     */
    public String getComment() {
       return comment;
    }

    /**
     * Set the comment for the opcode
     *
     * @param comment
     */
    public void setComment(String comment) {
       this.comment = comment;
       fireChange();
    }

    public void addChangeListener(ChangeListener listener) {
       changeListeners.add(ChangeListener.class, listener);
    }

    public void removeChangeListener(ChangeListener listener) {
       changeListeners.remove(ChangeListener.class, listener);
    }

    protected void fireChange() {
       Object[] listeners = changeListeners.getListenerList();
       int numListeners = listeners.length;
       for (int i = 0; i < numListeners; i += 2) {
          if (listeners[i] == ChangeListener.class)
             ((ChangeListener) listeners[i + 1])
                   .stateChanged(new ChangeEvent(this));
       }
    }
 }
```

OpcodeVertexView

View and renderer to visualise an opcode vertex on the diagram.

```
public class OpcodeVertexView extends VertexView {
   protected transient static WrapperRenderer renderer = new
WrapperRenderer();

   /**
    * Creates new VertexView for the specified cell
    *
    * @param arg0
    *            a graph cell to create view for
    */
   public OpcodeVertexView(Object arg0) {
      super(arg0);
   }

   public CellViewRenderer getRenderer() {
      return renderer;
   }

   public static class WrapperRenderer extends JPanel implements
         CellViewRenderer {
      private transient static JLabel label = new JLabel();
      private transient static OpcodeVertexRenderer portLabelRenderer = new
OpcodeVertexRenderer();

      /**
       * Cache the current graph for drawing
       */
      transient protected JGraph graph = null;

      /**
       * Cached hasFocus and selected value.
       */
      transient protected boolean hasFocus, selected, preview;

      /** Cached default foreground and default background. */
      transient protected Color defaultForeground, defaultBackground;

      /**
       * Constructs a renderer that may be used to render vertices.
       */
      public WrapperRenderer() {
         super(new BorderLayout());
         this.add(portLabelRenderer, BorderLayout.CENTER);
         defaultForeground = UIManager.getColor("Tree.textForeground");
         defaultBackground = UIManager.getColor("Tree.textBackground");
      }

      public Component getRendererComponent(JGraph graph, CellView view,
            boolean sel, boolean focus, boolean preview) {
         portLabelRenderer.getRendererComponent(graph, view, sel, focus,
               preview);

         // Format
         label.setOpaque(false);
         label.setText(view.getCell().toString());
         label.setVerticalTextPosition(JLabel.CENTER);
         label.setHorizontalTextPosition(JLabel.CENTER);
         setBackground((graph != null) ? graph.getBackground()
               : defaultBackground);

         this.graph = graph;
         this.selected = sel;
         this.preview = preview;
         this.hasFocus = focus;
         return this;
      }

      public Point2D getPerimeterPoint(VertexView view, Point2D source,
            Point2D p) {
         return portLabelRenderer.getPerimeterPoint(view, source, p);
      }
   }

   /**
    * The renderer class for instance view.
    */
   public static class OpcodeVertexRenderer extends VertexRenderer {

      protected CellView vertexView = null;
      protected OpcodeVertex vertex = null;

      protected FontMetrics fontMetrics = null;

      // Margin between port labels and the edge of the vertex
```

```
      public static transient int PORTLABELSPACING = 8;

      /**
       * The maximum width of a label, any label more than this value in
width
       */
      public static transient int MAXLABELWIDTH = 150;

      public void paint(Graphics g) {
         super.paint(g);
         g.setColor(getForeground());
         paintPortLabels(g);
      }

      public Component getRendererComponent(JGraph graph, CellView view,
            boolean sel, boolean focus, boolean preview) {

         // Finds the ports and install them into the renderer
         Graphics2D g = (Graphics2D) graph.getGraphics();
         fontMetrics = g.getFontMetrics();

         vertex = (OpcodeVertex) view.getCell();
         vertexView = view;

         Component c = super.getRendererComponent(graph, view, sel, focus,
               preview);
         return c;
      }

      /**
       * Draws a String. Its horizontal position
       * <code>x</code> is given and its vertical position is centred on
       * given y
       *
       * @param g
       *            a Graphics2D to draw with
       * @param label
       *            a String to draw
       * @param x
       *            an offset to left edge of the bounding box of vertex
       * @param y
       *            an offset to centre of the string
       * @param right -
       *            should the co-ordinates refer to the right of the label
       *            instead of the left?
       */
      public static void drawPortLabel(Graphics g, String label, double x,
            double y, boolean right) {
         FontMetrics metrics = g.getFontMetrics();
         int sw = metrics.stringWidth(label);
         int sh = metrics.getHeight();

         // Offset to account for the text size
         int offsetX = 0;
         if (!right) {
            // Just space a little from the edge
            offsetX = PORTLABELSPACING;
         } else {
            // Bring the left hand edge back to simulate right align, with
            // the spacing
            offsetX = -sw - PORTLABELSPACING;
         }

         // -4 to centre it vertically about the port
         g.drawString(label, (int) x + offsetX, (int) (y + sh / 2 - 4));
      }

      /**
       * Paints port labels in the view.
       *
       * @param g
       *            Graphics2D to draw with
       */
      public void paintPortLabels(Graphics g) {
         // Ports should be drawn in a lighter font
         g.setFont(new Font("sans-serif", Font.PLAIN, 12));

         paintPortLabels(g, vertex.getInputs().toArray(
               new OpcodePort[vertex.getInputs().size()]), false);
         paintPortLabels(g, vertex.getOutputs().toArray(
               new OpcodePort[vertex.getOutputs().size()]), true);
      }

      /**
       * Paint a particular array of ports
       *
       * @param g
       *            The graphics subsystem
```

```
 * @param ports
 *            The ports to paint
 * @param right
 *            Should these be on the right of the vertex?
 */
public void paintPortLabels(Graphics g, OpcodePort[] ports,
      boolean right) {
   if (ports.length > 0) {
      Rectangle2D bounds = GraphConstants.getBounds(vertexView
            .getAllAttributes());
      // Get the bounds of the vertex and deduct twice the cell label
      // height plus the vertical buffer distance from it.
      double height = bounds.getHeight();

      // Counter to calculate offsets
      int count = 0;
      float space = GraphConstants.PERMILLE / ports.length;

      // For each port
      for (OpcodePort p : ports) {
         String labelValue;

         if (p instanceof OpcodeOutputPort)
            labelValue = ((OpcodeOutputPort) p).getUserVar();
         else
            labelValue = p.getFormal();

         // Options values show with square brackets
         if (p.isOptional())
            labelValue = "[" + labelValue + "]";

         Point2D offset = new Point2D.Double(
               (right ? GraphConstants.PERMILLE : 0), count
                     * space + space / 2);

         double x = 0;
         double y = 0;
         if (offset != null) {
            if (bounds != null) {
               // By x position should be 0 or the width of the
               // vertex
               x = offset.getX() * bounds.getWidth()
                     / GraphConstants.PERMILLE;
               // The y position is the proportion of the vertex
               // height available for port label. Remember a bit
               // is reserved either end for the vertex label.
               y = offset.getY() * height
                     / GraphConstants.PERMILLE;

               drawPortLabel(g, labelValue, x, y, right);
               count++;
            }
         }
      }
   }
}
```

Parameter

Represents a formal parameter on an opcode definition.

```
public class Parameter {
   /**
    * Parameter name
    */
   private String name;

   /**
    * Is it optional?
    */
   private boolean optional;

   /**
    * Create a new parameter
    *
    * @param name
    *            Formal name of the parameter
    * @param optional
    *            Is this parameter optional?
    */
   public Parameter(String name, boolean optional) {
      this.name = name;
      this.optional = optional;
   }

   /**
    * @return Formal name of the parameter
    */
   public String getName() {
      return name;
   }

   /**
    * @return Optional status of the parameter
    */
   public boolean isOptional() {
      return optional;
   }
}
```

Variable

Contains static validation methods for variables.

```
public class Variable {
   // Variable validation as per
   // http://www.csounds.com/manual/html/OrchKvar.html

   static boolean isP(String v) {
      return v.matches("p[0-9]+");
   }

   static boolean isI(String v) {
      return v.matches("i[a-zA-Z0-9_]+");
   }

   static boolean isK(String v) {
      return v.matches("k[a-zA-Z0-9_]+");
   }

   static boolean isA(String v) {
      return v.matches("a[a-zA-Z0-9_]+");
   }

   static boolean isW(String v) {
      return v.matches("w[a-zA-Z0-9_]+");
   }

   static boolean isF(String v) {
      return v.matches("f[a-zA-Z0-9_]+");
   }

   static boolean isS(String v) {
      return v.matches("S[a-zA-Z0-9_]+");
   }

   /**
    * Is this variable name valid in a local context
    *
    * @param v
    *            The variable to check
    * @return Whether or not it is valid as a local variable
    */
   static boolean isValidLocal(String v) {
```

```
      return isP(v) || isI(v) || isK(v) || isA(v) || isW(v) || isF(v)
            || isS(v);
   }

   /**
    * Is this variable name valid in a global context
    *
    * @param v
    *            The variable to check
    * @return Whether or not it is valid as a global variable
    */
   static boolean isValidGlobal(String v) {
      boolean valid = v.startsWith("g");

      if (valid) {
         String sub = v.substring(1);
         valid = isValidLocal(sub) && !isP(sub) && !isW(sub);
      }

      return valid;
   }
}
```

PageParser

Ad-hoc parser to extract opcodes from the page nominally located at http://www.csounds.com/manual/html/MiscQuickref.html

```
public class PageParser {

   /**
    * Entry point for the Page Parser program to extract an opcode catalogue
    * from the Csound manual
    *
    * @param args
    *            Input URI for opcode quick reference page and output
filename,
    *            separated by a space
    */
   public static void main(String[] args) {

      if (args.length == 2) {
         String in = args[0];
         String out = args[1];

         PageParser p = new PageParser();

         try {
            System.out.println("Downloading and parsing " + in + "...");
            Hashtable<String, OpcodeGroup> tree = p.parse(in);
            System.out.println("Writing to " + out + "...");
            p.writeXML(tree, out);
            System.out.println("Done");
         } catch (JDOMException e) {
            e.printStackTrace();
         } catch (IOException e) {
            e.printStackTrace();
         }
      } else {
         System.out.println("PageParser\n\nUsage:");
         System.out.println("java " + PageParser.class.getCanonicalName()
               + " <inuri> <outfile>\n");
         System.out
               .println("<inuri>    URI of the Opcode Quick Reference manual
page to parse");
         System.out.println("<outfile>  Filename for the output XML file");
      }
```

```
   }

   /**
    * Write out an XML file from the tree-structured hashtable
    *
    * @param tree
    *            Hashtable with nested opcode groups that form the tree
    * @param outpath
    *            Path of local file to write to
    */
   public void writeXML(Hashtable<String, OpcodeGroup> tree, String outpath)
{
      XMLOutputter output = new XMLOutputter();
      Element root = new Element("opcodes");
      Document doc = new Document(root);
      root.setNamespace(Namespace
            .getNamespace("http://people.bath.ac.uk/cjw26/csdiag/opcodes"));
      root.setAttribute("version", "5.09");

      for (OpcodeGroup g : tree.values()) {
         root.addContent(g.getXML());
      }

      output.setFormat(Format.getPrettyFormat());

      File f = new File(outpath);
      try {
         output.output(doc, new FileOutputStream(f));
      } catch (FileNotFoundException e) {
         System.err.println(e.getMessage());
      } catch (IOException e) {
         System.err.println(e.getMessage());
      }
   }

   /**
    * Parse the Opcode quick reference page at the specified URI into a set
of
    * Opcode groups
    *
    * @param uri
    *            Source of "opcode quick reference" page
    * @param lookup
    *            A reference to a 1 dimensional lookup table for use for A-Z
    *            listing
    * @return Hashtable of top level groups keyed by group name
    * @throws JDOMException
    *             If the XML does not parse correctly
    * @throws IOException
    *             If the file is inaccessible
    */
   public Hashtable<String, OpcodeGroup> parse(String uri,
         Hashtable<String, Opcode> lookup) throws JDOMException, IOException
{
      // Load the XML parser
      SAXBuilder parser = new SAXBuilder();
      // Don't expand entities - there's some pesky nbsps
      // This still doesn't work for some reason
      // parser.setExpandEntities(false);

      Document doc = parser.build(uri);

      // Each opcode is conveniently in a <pre>
      ElementFilter pre = new ElementFilter("pre");
      // And the headers are in <b>
      Filter f = pre.or(new ElementFilter("b"));

      Iterator<Element> it = doc.getDescendants(f);

      boolean opcodesStarted = false;

      // The master array of grouped parsed opcodes
      Hashtable<String, OpcodeGroup> ocgs = new Hashtable<String,
OpcodeGroup>();

      OpcodeGroup currentGroup = null;

      // Iterating over document elements
      while (it.hasNext()) {
         Element e = it.next();

         // Don't start until we see the text Signal Generators
         if (e.getText().contains("Signal Generators")) {
            opcodesStarted = true;
         }

         // Seen a new opcode group
         if (opcodesStarted && e.getName().equals("b")) {
```

```
            String groupName = e.getTextNormalize().replace(".", "");
            String[] path = groupName.split(":");

            OpcodeGroup tlg;
            // Trim doesn't want to work, so regex replace whitespace
            String name = path[0].replaceAll("\\s$", "");

            if (!ocgs.containsKey(name)) {
               // New top level group
               tlg = new OpcodeGroup(name);
               ocgs.put(name, tlg);
            } else {
               tlg = ocgs.get(name);
            }

            // If we just added the conditional values or math ops, delete
            // them they're not real opcodes
            if (currentGroup != null
                  && (currentGroup.getName().contains(
                        "Conditional Values") || currentGroup.getName()
                        .contains("Arithmetic and Logic Operations")))
               tlg.getSubgroups().remove(currentGroup);

            currentGroup = tlg.addSubgroup(path);
         }

         // Seen an opcode
         if (opcodesStarted && e.getName().equals("pre")) {
            Opcode o = parseLine(e);

            if (o != null) {
               System.out.println(o.getName());
               currentGroup.getMembers().add(o);
               if (lookup != null)
                  lookup.put(o.getName(), o);
            }
         }
      }
      return ocgs;
   }

   /**
    * Parse the Opcode quick reference page at the specified URI into a set
of
    * Opcode groups
    *
    * @param uri
    *            Source of "opcode quick reference" page
    * @return Hashtable of top level groups keyed by group name
    * @throws JDOMException
    *             If the XML does not parse correctly
    * @throws IOException
    *             If the file is inaccessible
    */
   public Hashtable<String, OpcodeGroup> parse(String uri)
         throws JDOMException, IOException {
      return parse(uri, null);
   }

   /**
    * Parse one opcode line
    *
    * @param e
    *            The HTML element representing the opcode
    * @return An opcode object with parameters set from the line
    */
   protected Opcode parseLine(Element e) {

      Opcode o = new Opcode();
      String outputs = "";
      String inputs = "";

      // Split contents to output, opcode and input
      Iterator<Content> itc = e.getContent().iterator();
      while (itc.hasNext()) {
         Object c = itc.next();

         // First element seen is the <a> with the opcode name
         if (o.getName() == "" && c instanceof Element)
            o.setName(((Element) c).getText().trim());

         // If we see text before the opcode name, it's outputs
         if (o.getName().length() == 0 && c instanceof Text)
            outputs = ((Text) c).getText();

         // If we see text after the opcode name, it's inputs
         if (o.getName().length() > 0 && c instanceof Text)
            inputs = ((Text) c).getText();
```

```
      }

      inputs = inputs.replace("\\\n", "").trim();
      outputs = outputs.replace("\\\n", "").trim();

      if (inputs.contains("(")) {
         // It's a function
         o = new FunctionOpcode(o.getName());
      } else {
         o.setInputs(parseParams(inputs));
         o.setOutputs(parseParams(outputs));
      }

      return o;
   }

   /**
    * Parse a parameter specification string to parameter list
    *
    * @param s
    *            String of parameters from the left or right side of the
opcode
    *            name
    * @return A representative List of Parameter objects
    */
   protected List<Parameter> parseParams(String s) {
      ArrayList<Parameter> params = new ArrayList<Parameter>();

      int bracketpos = s.indexOf("[");
      boolean optional = false;

      Pattern param = Pattern.compile("[A-Za-z0-9]+");
      Matcher m = param.matcher(s);

      // Matches
      while (m.find()) {
         // If the optional params have started
         if (m.start() > bracketpos && bracketpos > -1)
            optional = true;

         params.add(new Parameter(m.group().trim(), optional));
      }

      // Pattern to match optional parameters (things in square brackets)
      // Superseded by above adhoc method since all optionals are together
in
      // the list
      /* Pattern optp = Pattern.compile("\\[([^\\]]+)\\]"); */

      return params;
   }
}
```

A5 Usage Instructions

This section provides usage instructions for the main **CsDiag** program and the **PageParser**.

The supplied disc contains, along with the full source code in the `src` directory, four Java Archive (JAR) files in the `bin` directory. These are required to run precompiled versions of the tools and must be present together in the same directory:

- `csdiag.jar` – Main diagrams program, executable JAR
- `pageparser.jar` – Csound manual to opcodes.xml parser
- `jdom.jar` – JDOM XML parsing library
- `jgraph.jar` – JGraph graph drawing library

With the files in place, CsDiag can be started by running:

```
java –jar csdiag.jar
```

The `opcodes.xml` file from the CD (or an equivalent generated with PageParser) should be located in the working directory to avoid an error.

The command to run PageParser has the following syntax:

```
java –jar pageparser.jar <input url> <output path>
```

The input URL should normally be:

http://www.csounds.com/manual/html/MiscQuickref.html

which is the location of the canonical manual.

The output path can be any local path (although paths with spaces are not accepted). Recall, though that it is necessary for the output file to be named opcodes.xml in the `CsDiag` working directory to be usable.

A6 Project Poster

UNIVERSITY OF BATH
Department of Computer Science

Diagrammatic Construction of Csound Instruments

Chris Ware
Supervisor: Prof. John Fitch

Abstract

Csound is a powerful music programming language, capable of emulating any commercial synthesizer. However it is also considered difficult for musicians without programming experience to use. Here, we specify and implement a graphical front end enabling instruments to be constructed as diagrams. Usable Csound code can then be generated from these diagrams. We also lay the foundations for the reverse process: generation of diagrams from existing code.

1. Introduction

An established practice among users and manufacturers of hardware synthesizers is the drawing of **diagrams** to record design and configuration. This is especially true of *modular synthesizers* where the musician connects up the components themselves. Csound is a software implementation of a modular synthesizer [1], yet natively it is programmed using an assembler-like language. This presents an opportunity for software which converts between the two representations, improving the accessibility of Csound.

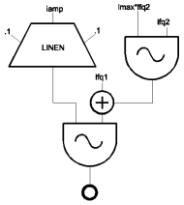


Figure 1: Example of a simple synthesizer diagram

This project aims to work incrementally towards a full solution where instruments can be constructed using both diagrams and code simultaneously. It will bridge the gap between these similarly to tools in other domains such as database and web design.

2. Background

We investigated the de facto **diagramming conventions** for Csound and also **language structure and coding style**. Well known publications in the field such as [1], [2] have established a trend of representing Csound **opcodes as blocks** in a flow diagram. Lines connect input and output terminals on the blocks to show assignment of variables to parameters. This directly influences our interface design.

The **current coverage** of this area in related programs (such as *Patchwork* and *Blue*) was reviewed, and of 11 critiqued none satisfied *all* the desirable properties:

- Capable of editing instruments
- Diagram-to-code generation
- Code-to-diagram generation
- Being cross platform, open source and up to date with Csound developments

3. Interactive Diagramming

Our program is based on the typical insert, drag, drop and resize paradigm used in almost every drawing package (making use of the JGraph library). Users add blocks representing opcodes to the workspace and connect them together by dragging between output and input "ports". The latter represents the assignment of outputs to variables to inputs (Fig 2).

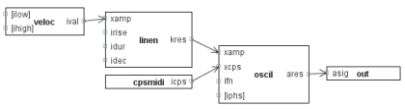


Figure 2: Incomplete diagram with five opcodes. Arrows connect some ports, indicating use of an output variable as a parameter value.

Opcode Acquisition

A consideration with the above mechanism is how the user selects the opcode they wish to add to the diagram, and indeed how we actually make every possible opcode (of which there are hundreds) available in the UI. We have a novel solution: **parsing the Csound manual**, specifically the "Opcode Quick Reference" [3]. This obtains not only the **formal parameters** for each opcode (from the canonical definition) but also a hierarchical **categorisation** of opcodes based on the headings in the manual. This transfers directly to the menus (Fig 3). It is also therefore easy to extend the program to handle new opcodes as they are created.

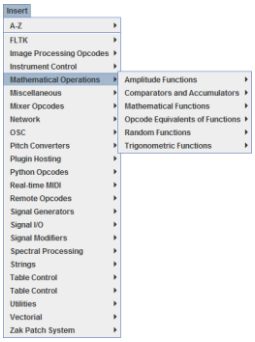


Figure 3: Opcode categories from the manual parsed to give the menu structure

4. Code Generation

Once a diagram has been created, orchestra code can be generated automatically by interrogating the graph structure. This seems trivial—for each opcode we have its name, the name of any output variables, and can trace edges to find the value "connected" to any input port. However, there are some complicating factors to consider:

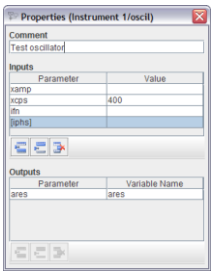


Figure 4: Properties box for more complex editing of opcode blocks

Expressions

Often the values assigned to the inputs of an opcode are more complex than just the output of another opcode. Values may be combined by **arithmetic operations** or transformed by **functions**. It is cumbersome to build such expressions using only diagram components so we implement the *properties box* (Fig 4) where an expression can be entered as text. The program will perform rudimentary parsing and automatically connect the relevant blocks on the diagram together to reflect the variables used in the expression.

Code Order

Ordering the generated statements requires care to ensure variables are defined with the correct values before they are referenced. Originally a backwards chaining solution was planned, working backward from opcodes without output variables and filling in dependencies. However this is problematic if variables are ever redefined in terms of themselves, so we **order the code based on physical layout** (leftmost blocks generate first).

5. Conclusions and Further Work

We have implemented a modern, flexible and extensible GUI allowing interactive graphical design of instruments and code generation for the full set of Csound 5 opcodes. The ability to generate diagrams from existing orchestra code remains to be added, however. This is suggested as a future project, now that a suitable foundation is in place. Recent solutions such as *QuteCsound* [4] can generate diagrams but these are not interactive, so a convergence of these two functions remains to be achieved.

References

1. Boulanger, Richard (Ed). The Csound Book. Cambridge, Massachusetts : MIT Press, 2000. 0262522616.
2. Gather, John-Philipp. Amsterdam Catalog of Csound Computer Instruments. Buffalo : University at Buffalo, 1995.
3. Vercoe, Barry et al. The Canonical Csound Reference Manual. Cambridge: MIT, 2008.
4. Cabrera, Andrés. QuteCsound. http://qutecsound.sourceforge.net/