Iconic programming for HMSL

Phil Burk, Robert Marsanyi
Center for Contemporary Music, Mills College, Oakland, CA 94613
phil@mill.s.edu, mm@i24.n125.z1.lidonet.org

ABSTRACT

This paper introduces a new iconic programming tool, Wire, and describes the integration of Wire with the existing Hierarchical Music Specification Language (HMSL). Compositions may now be created using a combination of the Wire Patch Editor, the Hierarchy Editor, and the Shape Editor. As an option, traditional text-based programming can also be brought into play.

1. WHAT IS WIRE?

Wire is a dataflow-based real-time programming environment for music experiment, written in HMSL. HMSL is a text-based object-oriented composition language, itself based on Forth and the Object Development Environment (ODE). Wire is designed to operate as a stand-alone system, or as an extension to HMSL. Its primary interface is an iconic circuit design screen of the sort used in other dataflow CASE and music environments.

A prototype of Wire was implemented in 1986 under the auspices of an Apple Computer Fellowship, intended for software synthesis. It was taken up again in 1991 in response to requests from HMSL users. As the project progressed, it became apparent that the power of the system lay in the integration of the dataflow/iconic language mechanism with the existing text-based programming mechanism of HMSL, which is the focus of this paper.

Additional features and grammatical attributes provided include a low "a-lot" threshold for beginning HMSL users in 1986. There was considerable work being done with iconic interfaces for a variety of programming environments, among them Hookup (David Levitin's system at MIT) and Kyma. In fact, as the conference hosted by Apple at the conclusion of the fellowship, there were at least three icon-based composition/performance environments presented.

Today the paradigm is widespread; environments exist for laboratory data acquisition and analysis tools, CASE tools, and of course music tools including "Max" and "Bars& Pipes". The usefulness of this sort of environment for composition is further born out by other works-in-progress.

1.1. THE WIRE ENVIRONMENT

The Wire environment shares the same basic features of all these systems. The user interacts with a Patch Editor by creating icons representing Unit objects, each of which may have plug-in or connection points for connecting to other objects. Patches can be collapsed to form new basic unit types, and can be embedded in other Patches to form hierarchies.

Information is passed into and out of a Patch using Units called Inbays and Outbays. A basic repertoire of Units exists for designing simple algorithms and for controlling devices such as MIDI, graphics, and DSP.

Wire objects can be edited while they are running, with the exception of user interface objects (faders, buttons, etc.). Each input plug in a Wire object can have one input and one pass-through connection defined; values appearing on the input are passed through the output, analogously to a MIDI "Thru" connection. This effectively allows a fan-in and fan-out of 1 for plugs. In Wire, the interface and execution are separate components of the system. Wire Patches can be run without the editor environment. Wire objects have a default representation, but their look can be customized by the user.

1.2. WIRE INTERFACES

The base class in Wire is the Unit. Units can be connected, disconnected, executed, kicked off by an input (which forces execution), queried by an output, and saved and loaded. Associated with each Unit is another class, Graphic,Unit, which provides the interface functionality of the Patch Editing screen. Patches, a subclass of Units, provide a way to hierarchically nest groups of interconnected Units.

The values that circulate through the plugs in a Patch are simple 32-bit numbers. Interpretation of the numbers is left to the execution of the Units themselves; for example, some Units expect the numbers to represent pointers to other objects, or MIDI values, or sample lengths. Those Units may be defined to validate their input data. Wire itself does not impose any data typing.
Wire supports both a data-driven and demand-driven dataflow mechanism, through the methods KICK and QUERY. When an input plug is KICKed, the Unit generally executes itself and passes output plug values "downstream", kicking any connected Units; thus data feeding into a Patch creates a rippling execution from inputs to outputs. By default, kicking the first input plug triggers execution. Any input plug can be made the trigger by holding down the Shift key when connecting to that plug. Through the use of a custom "kick function", other sorts of behavior can be defined. QUERY provides a way for a Unit to interrogate Units "upstream" for input values when required; execution proceeds backward from output back through inputs until all the required data is provided. The QUERY mechanism provides a way to obtain data values without having to explicitly send KICK messages through the preceding Unit structure. Use of both data-driven and demand-driven mechanisms simultaneously can lead to unexpected results; for example, it is possible to set up Patches that, once KICKed, oscillate continuously between QUERYing and KICKing without finishing. For this reason, QUERY can be optionally disabled.

1.3. Adding new types of units

ForthUnits are a subclass of Units, and provide a way to encapsulate definitions from the underlying language into the Wire environment. A ForthUnit contains a pointer to a Forth word (or subroutine), whose inputs match the input plugs to the Unit and whose outputs match the output plugs. Executing one of these Units simply passes the inputs to the word, executes it, and sends the outputs to the output plugs. New types of ForthUnits are easily constructed; in fact, it is relatively simple to define them "on the fly", while the Wire environment is operating and even while a Patch is executing.

For example, say we want to define a Unit that updates the frequency of a DSP circuit. We define a Forth word that, when given a frequency in Hertz, translates the value to a phase increment and sends it to the circuit:

```forth
: SET.FREQ [ Hertz Circuit -- ]
  hertz Hz>PhaseInc
  put.freq: circuit
```

Next we must define a word that will be called by the system when it is time to create such a Unit:

```forth
: BUILD.FREQ.UNIT ( -- unit )
  inputs[ "Circ" "Hz" ]inputs
  'c set.freq
  new.forth.unit
```

To make this new type of Unit available from the Patch Editor, we associate a name with a pointer to the function that creates it:

```
'c build.freq.unit "Freq"
add.unit.def
```

These definitions and comments can be entered in the Forth window while editing and running Patches, or included with other source code from a text file.

2. WIRE AND HMSL

While Wire as a standalone tool is useful, it is as a part of HMSL that it becomes interesting. In this environment, it becomes possible to use different development tools for different parts of the same piece. In this context, another new tool, the Hierarchy Editor (described at the end of this section), becomes the mechanism for integrating different ways of working interactively.

Wire and HMSL interact in two basic ways: HMSL objects can be embedded in Patches, and Wire objects can be used in HMSL hierarchies.

2.1. HMSL objects in Wire

HMSL and other ODE objects can be embedded in a Wire Patch using Object and Method Units.

Object Units

An Object Unit has two plugins: one input, and one output. When kicked on the input, the object is sent to the output. Most of the HMSL classes have a corresponding unit type; when a Unit of type SHAPE is created, for example, a corresponding HMSL Shape is also created and made available to the Shape Editor.

Method Units

A Method Unit has a variable number of input and output plugs. The first input plug is by convention
used to pass in the object to which the message is sent, the remaining inputs are the arguments for the method. A MARY: Uni, for example, has one input (the receiver object) and one output, as shown in figure 2. When kicked with the object, MARY: outputs the number of elements in the object to its output port.

In addition to the mechanism described, we considered two other ways of handling. One technique is to pass messages and their arguments as a parameter list to Method Units, as in Max. The other technique is to have one plug for each possible method and argument, which obviously becomes unmanageable. We chose to pass objects to Method units to take advantage of the polymorphism inherent in the underlying object-oriented system.

The inputs to MethodUnits are just numbers, as with other Wire objects. MethodUnits can check data on inputs that are supposed to receive objects to ensure that they are indeed valid objects. This validation can be disabled dynamically to decrease runtime overhead.

2.2. Wire objects in HMSL

Wire Units and Patches can be embedded inside HMSL in two ways: as Morphs or as replacements for functions.

Units are Morphs

The parent class of all Wire executable objects, OB.UNIT, is a subclass of OB.MORPH, the parent of HMSL objects. This means that Units and Patches can be embedded in an HMSL hierarchy, as with other HMSL classes such as Players. Collections and Sequencing. For example, a parallel Collection may contain two Players, a subCollection and a Patch, all to be executed in parallel when the parent Collection is executed.

HMSL Morphs invoke their component Morphs by sending them an EXECUTE: message. When a Patch is sent the EXECUTE: message, it passes its parent Morph to its InBay, making that object available for manipulation inside the Patch. When the Patch considers itself finished, it kicks its OutBay, which sends a DONE: message back to the incoming Morph. In the example below, the sequential Collection would wait for a DONE: messages from the Shape before sending an EXECUTE: message to the Patch.

Units can replace functions

Most HMSL objects’ behaviors are heavily customized by embedding forth functions to be executed at specific times in the object’s execution.

For example, all Morphs can optionally have functions to be executed on starting, stopping, or replacing (if they are replaced). Players may have functions for calculating time values; Collections may have functions to determine which component to execute next. Instruments may have interpreter functions which translate Shape data into output.

When integrated with Wire, all these functions can be specified as Wire Patches, where the function arguments are passed in through an InBay and the results returned through an OutBay.

2.3. The Hierarchy Editor

When HMSL was originally designed, one of the tools envisaged was a Hierarchy Editor that would let the user build and manipulate HMSL hierarchies dynamically. A prototype has been built based around the familiar notion of an outline editor. In addition to standard operations, it allows the user to invoke appropriate editors for component objects (the shape editor for Shapes, the Patch Editor for Wire Patches, and so on) and play or record all or part of a hierarchy. The authors are finding the editor is becoming the “basic base” for the making of pieces in HMSL, allowing us to look over all the elements of a piece, hear what’s going on at any points in the structure, and edit things on the fly while we work, a distinctly different experience to the edit-compile-audition cycle that the text environment engenders.

3. A SIMPLE EXAMPLE

The resulting HMSL environment is probably best understood with the aid of a simple example. Consider a piece with two parts, a short melodic phrase that plays twice followed by a sequence of notes that follow a random walk. The two parts repeat in an AABAAB pattern. While the piece is playing, any notes entered on a MIDI keyboard will replace notes in the melody. To generate this piece, we first use the Hierarchy Editor to set up a Sequential Collection containing the phrase Shape and a Patch, as shown below. The numbers in parentheses are the repeat counts.

(100) PCOL = OB.COLLECTION
(2) MELODY = OB.SHAPE
(1) RANKMARK = OB.PATCH

The melody contained in the Shape can be described graphically, or as an event list, or by reading a MIDI File, etc. The random walk Patch is created using the iconic Patch Editor and can be seen in figure 3. It fires when the Collection sends its address through the lobby. The Patch uses a Loop Unit to generate 16 notes. Each time the loop fires it calculates a random number, adds it to the previous number, clips it to the minimum and maximum values, outputs the
the Patch is executed, the Shape address is passed in the InBay. The number of notes in the Shape is extracted by the Many Unit and used to set the upper bounds for the Random Unit. When a MIDI note is received, it will kick the random unit which generates a random index into the shape. The note then passes through to the ED TO Unit which passes the notes into the Shape at the given Element and Dimension. Note that the Vel plug on the ED TO is larger to indicate that the Unit is triggered when the Note value is received instead of when the object is received.

4. RESULTS

All these disparate paradigms have distinct advantages and disadvantages. Wire, for example, is more forgiving to the user than the HMLs text environment and is often easier to understand because of the visual metaphor used. The HMLs text environment is more efficient, better at handling complete data structures, and ultimately more flexible. The Hierarchy Editor is good at ad-hoc manipulation of the large-scale structure of a piece. Sometimes, though, it is easier to create a hierarchy or a patch algorithmically by defining a Forth word. By integrating these paradigms into one environment, the composer can take advantage of the strengths of each.

REFERENCES

1HMSL for the Apple Macintosh and the Commodore Amiga is distributed by Frog Peak Music, P.O Box 151051, San Rafael, CA 94915-1051 (415) 461-1442
2Carla Santi, "Kyma", Symbolic Sound Corporation, Champaign, Illinois.
3Kyma(Dev.), Keith Lott, "A general purpose flowchart-based sound synthesis language for the Apple Mac II"; Robert Marsanyi, "Units under HMSL".
4LabVIEW 2, National Instruments Corp., Austin, TX.
5VEE-Engine, Hewlett-Packard Co., Palo Alto, CA.
6M. Puckett and D. Ziaorelli, "Max", Opcode Systems, Menlo Park, CA.
7Bar&Pipes", Blue Ribbon Bakery, Inc.