# The Run-Time Components of Sonnet

David H. Jameson
Computer Music Centre
IBM Research Division

**Abstract**

Sonnet is an audio-enhanced monitoring and debugging system whose aim is to investigate how sound can be used in a program development environment. Running under AIX, it consists of a visual programming language to design run-time actions that can subsequently be attached to running programs. Run-time actions are built from visual components that can be connected together. Components have the job of interacting with the running program, transforming data in useful ways, and ultimately arranging for sounds to be generated. In this chapter, I introduce the main visual components that were provided to make Sonnet a viable environment for program sonification and give examples of how to use them in practice.

## 1 Introduction

As interest in sonification has been growing, several research groups have been working on the use of sound for program monitoring and/or debugging over the last few years [1, 4, 5, 6, 7, 12]. Although an important concern has been to determine useful mappings from program execution space into sound space, there are other issues that need to be better understood. In particular, once you have decided to support sonification of program behavior, a variety of mechanisms are required to retrieve the desired information, to massage or transform it appropriately, and to inject the results to the audio generators.

## 2 Sonnet Overview

In a previous paper, Sonnet was introduced as a proposal for a program sonification system [7]. Sonnet was subsequently implemented on an IBM RS/6000 running AIX. As with many system tools, Sonnet has continued to evolve over time with more components[1] being added to support operations that were either necessary for basic operation of the system or for enhancing the system with new features.

Sonnet includes an integrated visual programming language called SVPL (Sonnet Visual Programming Language) that is used to construct run-time actions that can be associated visually with statements or data in running programs. For example, in Figure 1, the MIDI note number 64 is sounded when line 34 of the program is reached. The note stops when line 39 is executed. The three rectangles, which are examples of Sonnet components, form a single run-time action.

Each component in Sonnet contains at least one input or output port allowing connections to be made to other components. Input ports can be clocked or nonclocked, visually distinguishable by their colors (red or green respectively) and in most cases the clock property can be toggled by the user as desired. Nonclocked ports store their inputs. Clocked ports store their inputs and

[1] In earlier papers, components were called swidgets, a shorthand for sound **widgets**.

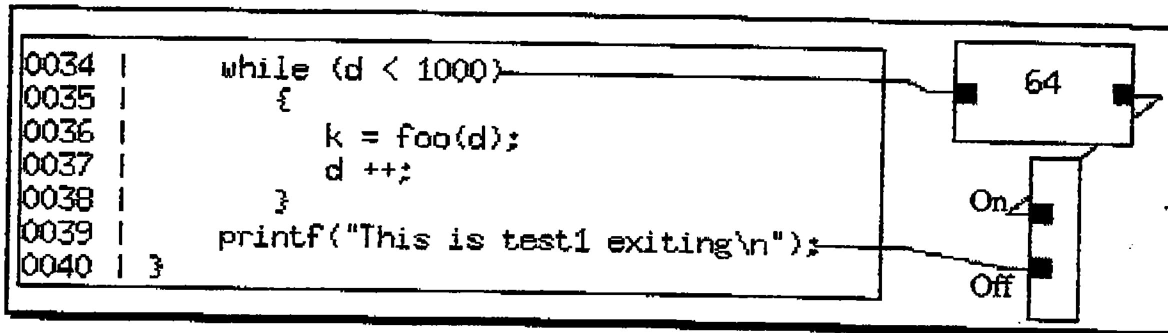


Figure 1: Triggering actions at run-time.

then trigger the component. Execution order is top to bottom, first for nonclocked ports and then again for clocked ports. Normally, newly instantiated components have all input ports except the bottom one set to nonclocked. This is then analgous to the right-to-left ordering used by MAX, the visual programming language developed by Miller Puckette [9], [10], [11], but the ability to toggle the clock property makes it easier to trigger components in some cases without the need for extra objects. For example, setting both input ports of a plus component allow the addition operation to occur when a packet arrives at either input port, not just the bottom one.

Output ports transmit packets of information to input ports. Packets contain type information and data. Valid packet types are integer, floating point, or boolean. Automatic conversion is applied where necessary.

Sonnet components are either primitive (implemented as C++ objects) or black-boxed (Figure 2). Black boxes are created simply by selecting a set of components (including other black boxes) which can then be collapsed into a single component and made available for subsequent usage as first class components. The system determines which ports should remain visible in the resulting blackbox based on interconnections between ports. Unconnected ports or ports that have connections to components not included in the blackbox remain visible. The rest are hidden.

Source files are simply a another subclass of components and each line of a source file is considered to be an output port. The packet produced by these output ports contains the instruction address of the beginning of the line.

## 3 Basic Components

There are no menus in Sonnet. Everything is controlled by SVPL, including operations such as running, halting, or continuing execution. This is extremely attractive because it allows a high degree of programmability as well as unattended operation for continuous monitoring, the latter being very important in a sonification system where sounds such as alarms are used to attract your attention from other tasks. In the descriptions below, a bold font is used for components and an italic font represents a port within a component.

## 4 Overlord

The **overlord** controls the overall execution of the program (Figure 3(a)). It has three input ports: *run*, *break*, and *continue*. To run a program once, simply send a packet to its *run* port. To run a program 10 times (say), trigger the **overlord** from an appropriately initialized **for** component. (Figure 3(b)). Stopping a program conditionally, perhaps based on values of variables

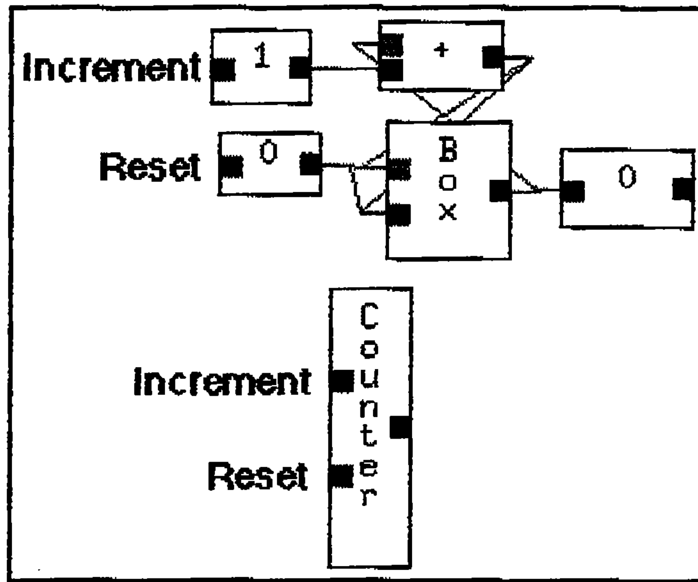


Figure 2: Encapsulating a counter into a blackbox.

at certain places in the program can easily be done. Of course, one normally does not want to simply stop the program, particularly when we are interested in the dynamic behavior of the system but it is useful to be able to do so. Incidentally, the **bang** component, which simply sends a content-free packet for triggering purposes, is named after the similar object in MAX. SVPL does have some similarities to MAX, particularly in the underlying procedural mechanism for communicating packets from one component to another. However, as was briefly mentioned above, there are several semantic differences as a consequence of the intended usage of SVPL. SVPL is documented in detail elsewhere [8].

# 5 Controlling Sound

Currently, Sonnet only supports sound generation via MIDI.[2] Several components are available for this purpose (Figure 4). The names have been added to the figures for reference only. Most of the ports in these components provide access to standard MIDI channel events. The port named Port exists to support multi-port MIDI interfaces.

# 6 Using the Overlord Component

I found it convenient to augment the **NoteOn** component with a *monophonic* port which accepts boolean packets. When *monophonic* is false (the default), **NoteOn** behaves in the obvious manner, transparently constructing MIDI events. When *monophonic* is true, **NoteOn** automatically transmits a MIDI note off message for the previous note whenever a new note is sent to it. This allows a sequence of note values to be played without the user having to to worry about stopping previous notes explicitly.

[2]I assume that the reader is familiar with MIDI fundamentals.

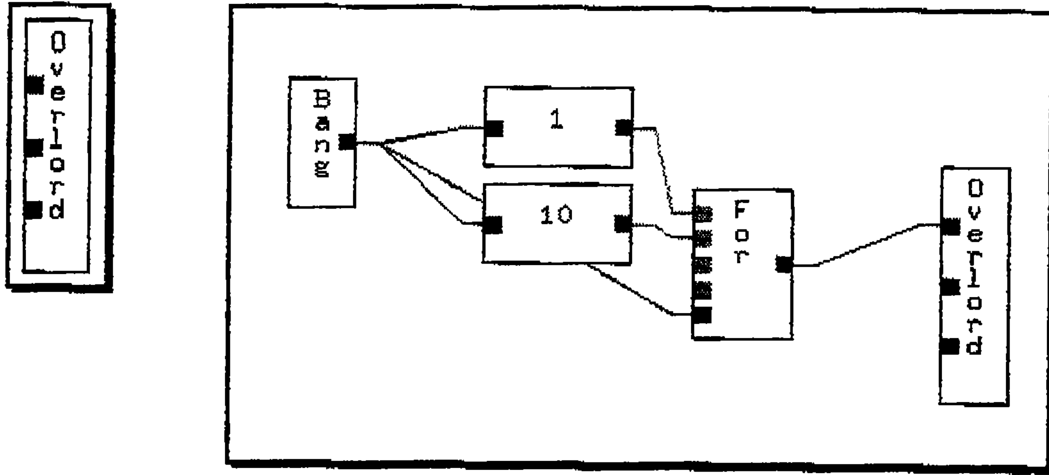


Figure 3: The overlord component with its three ports. (b) Running a program 10 times.

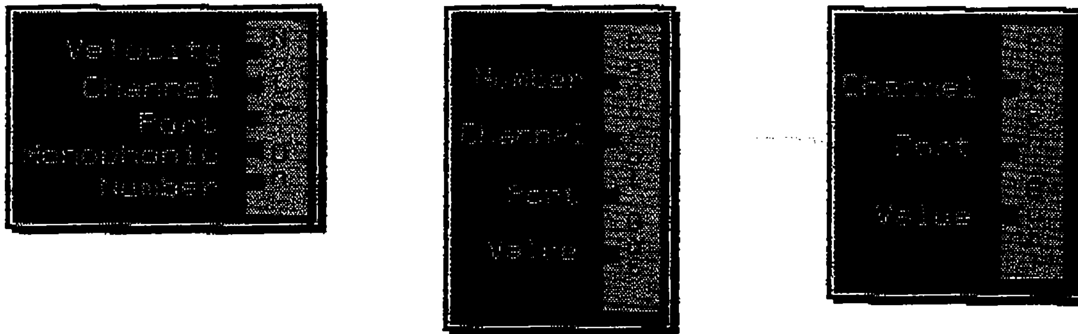


Figure 4: The MIDI components. (a) Component for generating MIDI note events. (b) Component for generating MIDI control change events. (c) Component for generating MIDI pitchbend events.

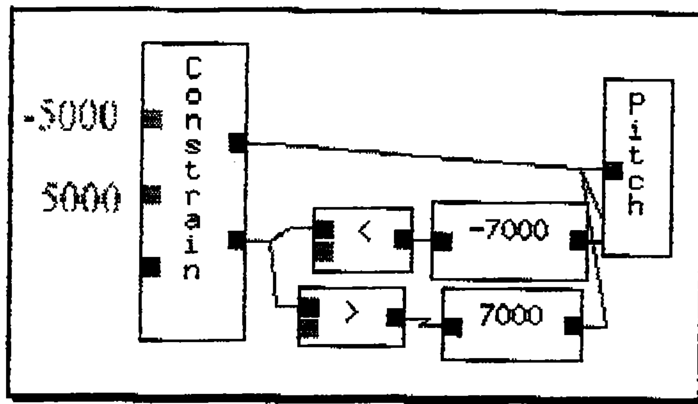


Figure 5: Constraining pitch values.

# 7 Constraining Data

Most MIDI messages are restricted to 7-bit values between 0 and 127. Pitchbend messages allow 14 bits and the **PitchBend** component requires values between −8096 and 8095. Although the results of calculations are usually normalized so as to lie within these bounds, it is often useful to place a filter in the data stream such that acceptable values go directly to the sound components and other values are redirected to some other component group, perhaps to generate an alarm indicating out of bounds. The **constrain** component implements this precise behavior. In Figure 5, the top two ports in the constrain component specify the valid range allowed for values arriving in at the bottom port and in this example, the allowable values must be within the range of −5000..5000 inclusively. If an incoming value is within this range, it is immediately transmitted through the upper output port of **constrain** where it controls the value of a MIDI pitchbend message directly. If the incoming value is outside the valid range, then it is immediately transmitted through the lower port of **constrain**. Depending on whether the value was negative or positive, a large negative or positive value is sent to the **Pitch** component.[3] The intent is that incoming smoothly changing values within the allowable range produce smoothly changing pitches. As soon as an incoming value goes beyond the valid range, a glaringly different pitch is generated.

# 8 Accessing Variables and Arrays

The **memory** component is used to access variables in a program. In Figure 6(a), the three input ports are *read*, *write*, and *offset*. The *read* and *write* ports do the obvious operation. The *offset* port was added later to allow data lying at contiguous locations to the named variable to be accessed. Although intended primarily for access to arrays, it is also useful for checking nearby memory in the case where "stepping" on other variables is a concern. The actual variable to be accessed is specified by typing its name inside the **memory** component. In this example, **memory** is associated a variable in the debuggee called Buffer. A more elegant mechanism would have been to select it in the source file and drag it to the component.

Once array access was available, it became necessary to have a component that would step through all the values of the array. This is the primary purpose of the **for** component (mentioned earlier in conjunction with the **overlord** component) which generates a sequential stream of

[3]The Pitch component here is a blackbox replicating the standard PitchOut component except that default channel and port values have been chosen.

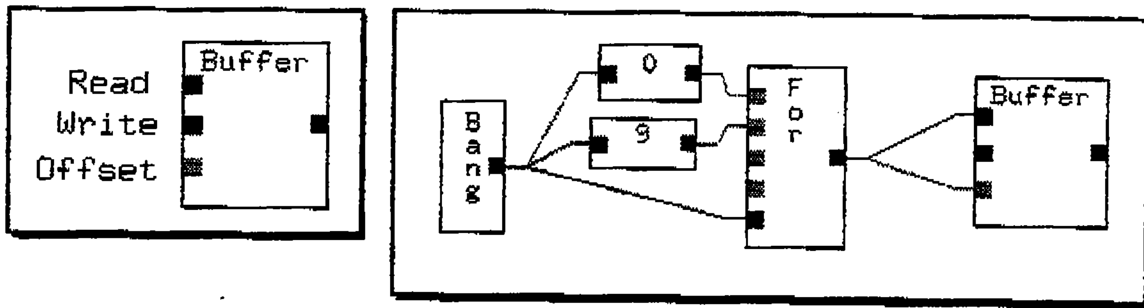


Figure 6: Accessing a program variable with the memory component. (a) Accessing a memory variable. (b) Indexing through elements of the variable.

integer packets with monotonically increasing (or decreasing) values. These packets are then used to index into the array (Figure 6(b)). Notice that the output of the **for** component is connected to the *read* and *offset* ports of the **memory** component. This does not cause a race condition since the semantics of SVPL guarantee that the *offset* will be set before the memory is actually read. Together with driving sounds via the output values of an array, triggering the for component from the running program is a powerful way to play an array continuously during runtime.

## 9 The Random Component

The **random** component was created for a specific purpose although it has general usage. In my examples of playing arrays of data during a sort operation [7, 8], I indexed through a complete array each time an operation (such as swapping elements) was applied to the array. Although this is adequate for small arrays (the prototypical "blackboard" example), it does not scale up very well. As the array gets larger, too much time will be used playing the array. The solution is to choose some subset of the array but there are three caveats. First, the chosen subset should be different each iteration so as to maximize the coverage of the array. Secondly, since we are interested in the "sortedness" of the array, the sequence of indices chosen to create the array subset should be monotonically increasing. Finally, special care should be taken to cover special cases where possible [2]. Of course, special cases depend on the particular algorithm and data structures in mind. In the case of an array being sorted, special cases should include the first element, the last element, and possibly the middle element of the array.

A **random** sequence of monotonically increasing integers can be generated using the random component and an accumulator, the latter of which is easily built from existing components (Figure 7). The two upper input ports of random are used to set the minimum and maximum allowable values. Triggering the bottom port causes a number within the defined range to be transmitted. The value 4 shown in the diagram via a view component is the most recent value generated and the value 18 is the accumulated sum of values seen so far.

## 10 Periodic timers

The timer component (Figure 8) is used to generate a packet periodically which can then be used to trigger an aribtrary run-time action. It is useful for sampling (e.g., get the value of some variable every 200ms) and, in conjunction with the **TxSignal** (see below), for generating periodic signals that are expected to be handled by the application.

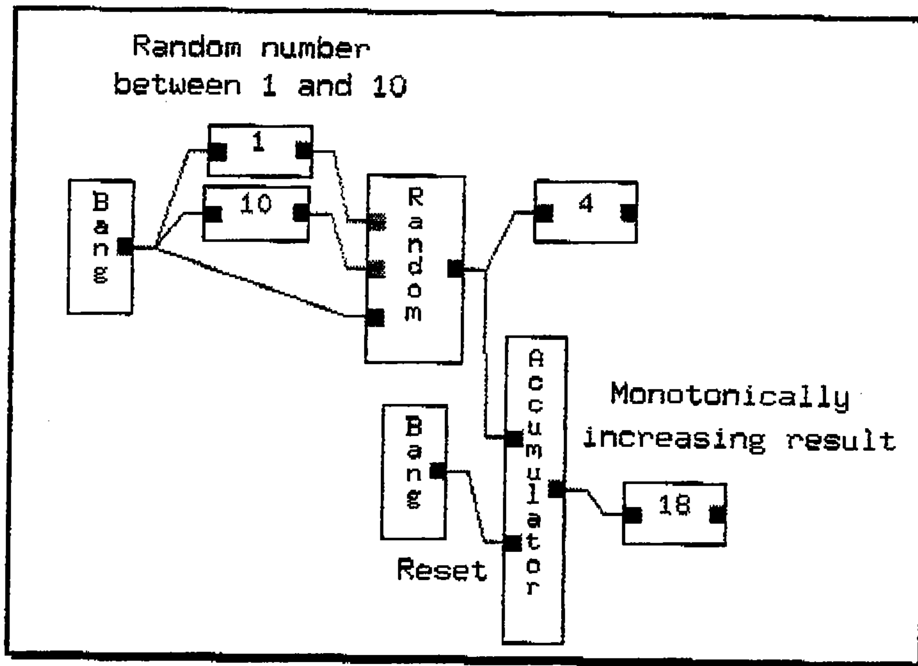


Figure 7: Generating monotonically increasing random numbers for probing arrays.

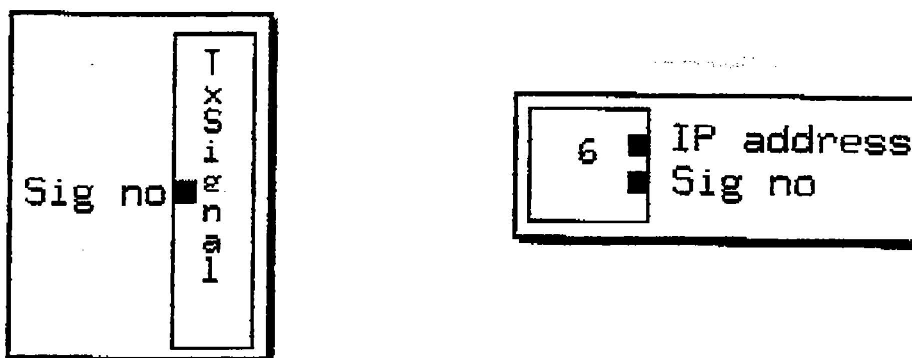


Figure 8: Signal handling. (a) Raise a signal. (b) Catch a signal.