Structured Interconnection of Simulation Programs

by

David G. Messerschmitt
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, Ca. 94720

ABSTRACT

A standardized and structured approach to the interconnection of different program modules involved in a functional simulation of a communications or signal processing system is described. This method is an improvement on the fifo interconnection described earlier [1], in that it retains the advantages of that approach while at the same time simplifying the programming. This interconnection we call a "random access buffer", and allows the programmer to access samples of input and output time sequences a very natural manner.

1. Introduction

In the structured functional simulation of communications and signal processing systems, there is a need for a standardized interface between software modules. This enables modules written by different programmers to be effortlessly interfaced, and enables a library of modules which can be freely interconnected to be accumulated.

The first version of BLOSIM, a structured functional simulation program, used a standardized first-in first-out (fifo) buffer to interconnect modules [1]. This interface is similar to that used in operating systems, such as the UNIX piping and I/O redirection operations.

The fifo in BLOSIM has proven with experience to satisfy two goals: it provides a standardized interface and it has proven to be very flexible and general, handling every interface problem that we have encountered.

Experience with BLOSIM has shown, however, that the fifo interface introduces some programming difficulties. In particular, in some situations considerable attention must be paid to details unrelated to the system being simulated, such as the overflow and underflow of fifo buffers. While it is tempting to say that this is the price that must be paid for the structured approach of BLOSIM, a better approach is to look for a method of interconnection which retains the advantages of the fifo, but simplifies rather than complicates the programmers task. This paper describes an interconnection approach which hopefully accomplishes that goal.

The following are the burdens placed on the programmer by the fifo interface:

1. The programmer must detect when an input fifo empties, and do a normal return to BLOSIM.
2. Similarly, the programmer must detect a full output fifo, and return.
3. A typical module, which is called a STAR in BLOSIM, generates output samples on the basis of a set of current and past input samples. With the fifo, the programmer must store and manipulate past input samples, which are discarded from the input fifo as soon as they are accessed.
4. Where a block is included in a feedback loop, it is necessary in order for the simulation to get started for the block to put out any samples it can before it accesses input samples. For example, a delay block can put out a number of zeros equal to the number of samples of delay before reading an input fifo.

The first two problems are artifacts of BLOSIM and are not factors in an operating system environment, where a process is automatically suspended and its state saved when it exhausts an input fifo. BLOSIM does not do this, because this operation is not supported by operating systems, and therefore cannot be done in a portable fashion. Generally code to perform this would be "dirty"; that is, be specific to a particular operating system and compiler.

This paper describes the block interconnection of a new version of BLOSIM, the goal of which is to simplify the user's programming task. This new version exploited the experience gained from extensive use of the previous version to achieve the following goals:

1. Maintain the generality of BLOSIM, which enables it to be used in virtually any simulation problem.
2. As one aspect of 1., continue to support synchronous and asynchronous sampling, and in the former case decimation and interpolation.
3. Retain the structured nature of the block interconnection, making the interfacing of diverse blocks effortless.
4. Define an interface which simplifies rather than complicates the user's programming task.
5. Define an interface which is referenced in a manner natural for someone doing a simulation of a signal processing or communication system.

This paper will describe the module interface in this new version of BLOSIM, and give some examples. The remainder of BLOSIM, including the hierarchical definition of blocks and independent topology definition [1] are essentially the same as the earlier version and will not be discussed here.

2. Random Access Buffer

The new block interconnection interface is called a random access buffer. It is similar to a fifo in that new samples are added to the "head" of the buffer, and old samples are removed from the "foot" of the buffer. However, it differs from a fifo in that the user program can access or even change any sample in an input or output buffer (hence the term "random access").The random access buffer also separates the incrementing of time from the reading or writing of samples to the buffer. What follows is a description of the random access buffer.

Any given buffer is connected to two blocks. To one of those blocks it appears as an "output buffer" and is the place where the block puts samples that it generates. To the other block it is an "input buffer", where the block gets its input samples. The interface of a block to an input buffer and an output buffer are slightly different. Figure 1 illustrates a programmers view of an input random access buffer. It consists of "cells" into which another block wrote data. Each cell can consist of an arbitrary data structure, such as for example a single floating point number, or a floating and a fixed number, etc. Conceptually, each cell contains some data corresponding to a single time increment of the simulation. The function pin(in#,k), where "in#" is the number of the input under consideration, returns a pointer to a cell in the buffer. The second parameter "k" specifies which cell is accessed, where k=0 corresponds to the "current" input, and for example k=5 corresponds to five time increments in the past. Using this pointer to the desired cell, the block can read the data in the cell or change a data value in the cell. Generally the buffer will include some cells corresponding to future time increments, and BLOSIM will not allow a block to access these cells (marked "forbidden" in the figure). One implication of this rule is that a block cannot increase the size of one of its own input buffers.

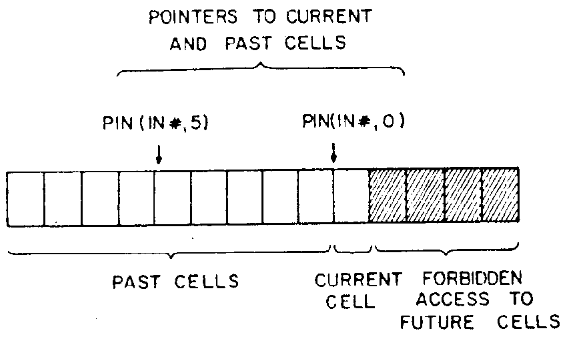

POINTERS TO CURRENT AND PAST CELLS
PIN(IN#,5)
PIN(IN#,0)
PAST CELLS
CURRENT CELL
FORBIDDEN ACCESS TO FUTURE CELLS

Figure 1. Programmers view of an input random access buffer.

Time is incremented by a second function it(in#), which moves the current cell forward in the buffer by one cell. If there are no more cells in the buffer, it(in#) returns a "0" value, which the block can test and return to the BLOSIM kernal. If the current cell is the last cell in the buffer after the time increment, it() returns -1; otherwise it returns +1.

The programmers view of an output random access buffer is shown in Figure 2. The function pout(out#,k) returns a pointer to an output cell, where "out#" is the number of the output and "k" is again referenced to the current cell. By definition, the current cell is the last cell that was written by the block, and hence the last cell in the buffer. If the block wishes to write another sample on the output buffer, it calls the function pnew(out#), which adds another cell to the end of the buffer and returns a pointer to this new cell. An output buffer cannot overflow; that is, another cell can always be added. (In fact BLOSIM places an installation defined limit on the number of cells that can be created, and the programmer must stay within this limit.)

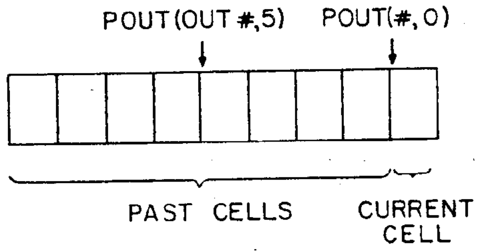


Figure 2. Programmers view of an output random access buffer.

These programmer views of an input or output random buffer fit very naturally into a typical simulation because the programmer references input samples relative to the current time (the variable "k" is essentially the same as the operator z^{-k}). The block calculates the current output(s) on the basis of the current and past inputs, and then increments time. In addition, decimation and interpolation and asynchronous sampling rates are very natural to implement. The next section will give some programming examples.

In filling an output buffer, BLOSIM minimizes memory usage by retaining only as many past cells as is necessary. In addition, BLOSIM automatically determines when it is possible to put out samples without reading an input buffer, an important feature in getting the simulation started. These two features require that the user specify the minimum and maximum delays accessed by a block in its input fifos, and the maximum delay accessed in its output fifos. These parameters are set by the functions min_in_d(in#) and max_in_d(in#) for an input buffer, and max_out_d(out#) for an output buffer. These values, if not set, default to zero.

3. Programming Examples

A few simple programming examples which illustrate the use of a random access buffer interface will be given. To avoid having to explain all the details of BLOSIM, these programs will leave out a small amount of initialization code which is necessary.

Perhaps the simplest block models a delay z^{-n}, a library routine which is used in almost every simulation. Here is the routine for the case $n=10$ (in practice n would be a parameter):

```
float *pin(),*pout(),*pnew();
#define x(A) *pin(0,A)
#define y() *pnew(0)

delay() {
        if(start()) max_in_d(0)=min_in_d(0)=10;
        while(it(0)) {
                y() = x(10);
                } return(0);
        }
```

This program is written in C, and the syntax "*pnew" means the contents of the memory pointed to by "pnew". We first declare the contents of all the buffers to contain floating numbers, and then define the contents of the input buffer delayed by 10 samples and the new sample on the output buffer to be x(10) and y() respectively using the macro substitution capability of the C compiler (this is strictly for readability of the code). The function start() returns "true" if this is the first call to delay(), at which time the minimum and maximum delays are set to 10. The remainder of the routine simply increments time, and as long as this is successful puts out the sample in its input buffer 10 cells from the current cell. Even if the block connected to the input of the delay block has not executed yet, this block will put out 10 zeros because of the details of the random access buffer implementation discussed in the next section.

As a slightly less trivial example, here is the code necessary to simulate an IIR digital filter

$$y_k = \sum_{i=0}^{10} a_i x_{k-i} + \sum_{i=1}^{5} b_i y_{k-i}$$

where the input to the filter is x_k and the output is y_k:

```
float *pin(),*pout(),*pnew();
#define x(A) *pin(0,A)
#define y(A) *pout(0,A)

iir() {
        float yout;
        int i;
if(start()) {
        max_in_d(0) = 10;
        max_out_d(0) = 5;
        }

while(it(0)) {
        yout = 0;
        for(i=0;i<=10;++i)
                yout += a[i] * x(i);
        for(i=1;i<=5;++i)
                yout += b[i] * y(i);
        *pnew(0) = yout;
        }
return(0);
}
```

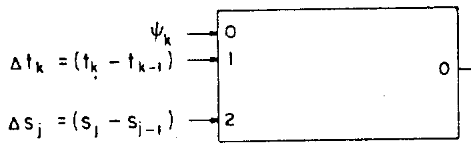
Note how the random access buffer has simplified the programming by eliminating the necessity of declaring any arrays containing the samples, shifting the samples through those arrays, and so forth, as might be required in writing a program from scratch. In effect, these services are performed by BLOSIM itself.

A final example will illustrate how the random access buffer supports asynchronous sampling rates. Suppose that a signal is represented by samples x_k, which are its samples at times t_k, and we wish to find the samples at times s_j. This problem arises quite often in simulations of communication systems, for example in the acquisition of phase locked loops or in full-duplex data transmission systems with asynchronous streams in the two directions. If the desired samples are called y_j, and $g(t)$ is the interpolation function (usually the impulse response of a low-pass filter), then the desired samples are given by

$$y_j = \sum_k x_k g(s_j - t_k)$$

This is naturally quite a bit more difficult than the previous synchronous examples, but considerably simplified because of the buffer interface.

A block which performs this task is shown in Figure 3a. It has as inputs x_k, the associated time since the last input Δt_k, and Δs_j. The time increments since the last sample are used in place of the absolute times for accuracy and to avoid overflow.

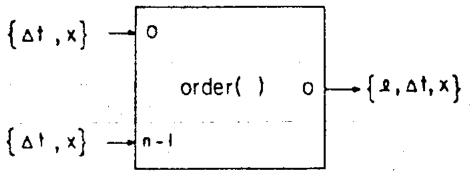


ψ_k
$\Delta t_k = (t_k - t_{k-1})$
$\Delta s_j = (s_j - s_{j-1})$
0
1
2
0
y_j

(a) Asynchronous Interpolation GALAXY

Figure 3a. An block which performs asynchronous interpolation.

In the spirit of structured programming, we break this block into appropriate pieces, and look for opportunities to solve a more general problem. In particular, the block in Figure 3b is very useful in realizing the interpolator of Figure 3a. This block takes an arbitrary number of inputs, each input a sample value and a time since the last input, and puts the samples out in order of time occurance. In addition to the ordered sample values, it puts out the time since the last output and the number of the input from which that output sample came. In the interest of space, we will show only the routine order() which implements this function. This block can be used to conveniently interleave the output sample times s_j with the input times t_k, and a second block which calculates the output samples y_j is then straightforward.



(b) STAR which orders Asynchronous Inputs

Figure 3b. A partitioning of the function of Figure 3a.

The routine is shown below:

```
order() {
/* declare variables */
        int input, input_min;
        float dt_min;
        struct {
                float dt;
                float x;
                } *pin();
        struct {
                int l;
                float dt;
                float x;
                } *pout();

/* set maximum delays which are not 0 */
        if(start())
                for(input=0;input<n;input++)
                        max_in_d(input) = 1;

/* process inputs until an input buffer is empty */
        while() {

/* find minimum time increment on input */
                dt_min = pin(0,0)->dt;
                input_min = 0;
                for(input=1;input<n;input++)
                        if(pin(input,0)->dt < dt_min) {
                                dt_min = pin(input,0)->dt;
                                input_min = input;
                                }

/* increment time on that input,
   return if input would then be empty */
                if(!it(input_min)) return(0);

/* increment time on output, then put out time and
   sample */
                pnew(0);
                pout(0,0)->l = input_min;
                pout(0,0)->dt = pin(input_min,1)->dt;
                pout(0,0)->x = pin(input_min,1)->x;

/* subtract time increment from all inputs */
                for(input=0;input<n;input++)
                        if(input != input_min)
                                pin(input,0)->dt -= dt_min;
```

Note first of all that both pin() and pout() have been declared to be pointers to structures containing two or three relevant variables. This ability to group variables together in passing through random access buffers between blocks is very convenient. The main routine is imbedded in a while() statement which executes until the internal test detects an empty input buffer and does a return. The convention is that the input times represent the time since the last *output* sample, or initially the times since the beginning of the simulation. First the input times are checked, and the minimum is found. The sample value and associated time corresponding to that minimum is then output. Finally, the time increment which was output is subtracted from all the input times residing in the input buffers. This illustrates that it is sometimes very handy to be able to modify the contents of an input buffer! In this case, it avoids the declaration (and saving as a state) of in:ernal variables to the routine. In contrast to the earlier version of BLOSIM, it is frequently not necessary to declare internal state variables.

4. Implementation of Random Access Buffer

The random access buffer is implemented in BLOSIM using a circular double-linked list data structure as shown in Figure 4. The cells are organized in a circle, and as time is incremented the data is stationary but the pointers move. The pointer pout(out#,0) points to the last data sample generated by the block connected to the input of the buffer, while pin(in#,0) points to the current data sample being accessed by the block connected to the output of the buffer. The shaded blocks are therefore forbidden for access by the latter block until time is incremented. There is another pointer "poldest" which points to the oldest sample that it is necessary to retain in the buffer (as determined by max_out_d() and max_in_d()). As new samples are added to the buffer, pout() is moved around the buffer until it reaches poldest. When that happens, more memory is allocated to the buffer since the buffer is effectively infinite in size.

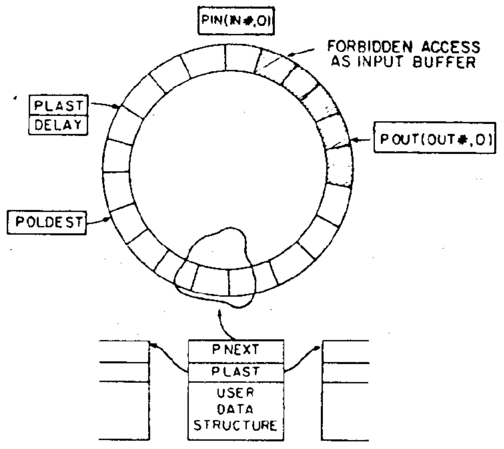


Figure 4. Implementation of a random access buffer.

Each cell in the buffer includes, in addition to the user data, two pointers which point to the two adjacent cells. When additional memory is added, the four pointers in the two cells on both sides of this new memory are simply re-linked to this memory. This data structure thus never requires the movement of data samples, and enables the size of the buffer to by dynamically increased very simply. One disadvantage is that to find a sample with some particular delay, it is necessary to move through the buffer once cell at a time. To minimize the time spend doing this search, the starting point is always at the last reference to the buffer (pointed to by "plast" with associated delay "delay"). This is efficient because most programs access delays in ascending or descending order.

5. Conclusion

Extensive experience with the new block interface in BLOSIM has not yet been accumulated, but it is apparent from the examples that the programming of block routines has been considerably simplified. The importance of an interface such as this is that the larger the community of users who will adopt a common interface for software modules, the greater the ease with which software can be interchanged and duplication of effort can be avoided. Probably the greatest weakness of the interface described here from this point of view is its implementation in C, while most simulation software is currently written in FORTRAN. Unfortunately, it would be difficult to reproduce this software interface in FORTRAN.

REFERENCE

1. D.G. Messerschmitt, "A Tool for Structured Functional Simulation", *IEEE Trans. on Special Topics in Communications*, Jan. 1984.