David G. Messerschmitt, "A Tool for Structured Functional Simulation",
IEEE Journal on Selected Areas in Communications, Vol. SAC-2, NO.1, January 1984,
pp.137-147

A Tool for Structured Functional Simulation

DAVID G. MESSERSCHMITT, FELLOW, IEEE

Abstract—BLOSIM is a general purpose time-driven (as opposed to event-driven) simulation language. It is written in C language, and is intended to provide a highly structured environment for simulations, thereby making practical the accumulation of libraries of simulation routines which can be reused and making multiprogrammer simulation efforts more practical. It is written with the philosophy of not including any simulation primitives within the language itself, but rather complete generality is maintained by having the user provide these as C routines (either coded from scratch or from a user-provided library). It includes as features a hierarchical specification of blocks, interconnection of blocks by first-in first-out buffers, the passing of parameters to blocks, multiple instances of blocks, and automatic scheduling of the order of block execution. It has been used for the multiprogrammer simulation of data transmission and speech processing systems, in both academic and industrial environments, with good results.

Manuscript received August 15, 1983; revised September 10, 1983. This research supported by Racal-Vadic, Advanced Micro Devices, Fairchild Semiconductor, Harris Semiconductor, National Semiconductor, with a matching grant from the University of California Microelectronics and Computer Research Opportunities Program.

The author is with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720

I. INTRODUCTION

BLOSIM, which is pronounced "blossom," stands for "block simulator," and is a general purpose simulation program for sampled data systems (or systems which can be represented as sampled data systems). It is a time-driven (as opposed to event-driven) simulator, and hence is efficient for simulating systems which operate on data at regular time intervals. Such a simulation approach has been called a "next-state simulator" [5]. BLOSIM can easily accommodate systems with different sampling rates present at the same time, or systems with internal asynchronous sampling rates.

0733-8716/84/0100-0137$01.00 ©1984 IEEE

The most common approach to simulations of this type is to write a special purpose program (often in Fortran) for the particular system being simulated. Often a structured methodology is not used, with the result that the simulation quickly deteriorates into hopeless complexity and the simulation code is of no value in future simulation efforts. Another alternative is to use one of the previously available block or next-state simulation languages [5]–[8]. These languages were written with the philosophy that many of the elementary blocks often appearing in systems are provided as an integral part of the simulation language. This approach suffers from inefficiency, as well as a lack of generality since rarely does the language designer's vision extend far beyond his own simulation experiences. A third approach is to write a general simulation program with many system building blocks built in, a simplified user interface, and the ability for the user to add custom programmed blocks. Many of the papers in this issue fall into this latter category.

BLOSIM is quite different from these other approaches to simulation. BLOSIM itself does not include internal models for any particular system to be simulated; these must be provided by the user. In this way complete generality is maintained, and the use of the program is not limited by the designer's choice of which models to include. Of course, this has the disadvantage that the user must provide modeling routines for the entire system, in a sense making BLOSIM similar to writing a special-purpose simulation program for the system. However, BLOSIM is designed so as to encourage and make it easy for the user to build a library of routines which can be used in future simulations.

Philosophically, BLOSIM is closest in approach to writing a special-purpose simulation program, and offers the same advantages of efficiency and generality. At the same time, it overcomes the deficiencies of many special-purpose simulation efforts by providing a structured environment for programming simulations. This structured environment encourages the programmer to use highly structured programming techniques, but without requiring any prior familiarity with these techniques. This structured simulation environment pays two major dividends. First, BLOSIM encourages the writing of simulation code for small pieces of the system being simulated, and this code is written to a single carefully defined interface. As a result, libraries of routines which can be reused in future simulations can be developed. Second, multiprogrammer simulation efforts become relatively painless since routines written by different programmers readily interface one another.

BLOSIM is also very natural to use for system implementors since it encourages them to divide the system into small interconnected blocks in the same way that a system is partitioned for implementation. The user then provides a simulation program for each of these blocks, and in a separate routine a specification of the topology of interconnection of these blocks. BLOSIM then handles the details of the actual interconnection and execution of the block programs.

The key design choice in BLOSIM was the manner in which the blocks are interconnected. Data samples output from one block and input to another pass through first-in first-out buffers (FIFO's) implemented internal to BLOSIM. This method of interconnecting the blocks is similar to the communication method often used in operating systems, and has the important advantage that the topology of interconnection of the blocks is completely separated from the internal implementation of the blocks. Thus, the programs implementing the blocks are written to a standard interface to BLOSIM FIFO's, and do not interface each other directly. This interface is carefully defined to hide the internal impelmentation of any particular block from other blocks, ensuring that the implementation of any block can be changed with minimal likelihood that the remainder of the simulation will be affected. The topology of interconnection of the blocks is specified in a separate place, making it easy and trouble-free to add and delete blocks, and generally make changes to the system being simulated. Parameters can also be passed to the blocks from the same place where topology is defined, thereby specializing their functionality to a particular simulation and making it simple to try different combinations of parameters.

It should be emphasized that a BLOSIM simulation consists of a single executable program, and only depends on the operating system for services like memory allocation, file access, etc. In particular, the interconnection of blocks is internal to the program, and not through the operating system. The use of the UNIX operating system "pipe" and "fork" mechanisms for block interconnection were considered in the early stages of design, but were rejected because of the inefficiency which would result and the limitation which would be imposed on the number of blocks.

BLOSIM supports a hierarchical definition of blocks. That is, it is simple to define new blocks which are made up of specified interconnections of other blocks. In subsequent simulations, these higher level blocks are indistinguishable from blocks implemented directly by a user program. In addition, multiple instances of blocks are supported; that is, the same block can appear as many times as desired in a given simulation (usually with its functionality specialized by setting different values for its parameters). This is particularly convenient for simulation of systems which have a replication of similar or identical functions (for example a systolic array or a system with multiple filters).

BLOSIM makes many consistency checks during topology definition and execution, allowing it to detect and correct many user programming errors. In addition, BLOSIM has a narrative mode, in which it prints out on the user's terminal what it is doing at each step, allowing the user to determine if the topology has been properly defined and the user block routines are functioning properly.

BLOSIM is implemented in C language [1], and the user-provided routines must also be written in C. Many of

the features of C discussed in a companion paper [4], and particularly dynamic allocation of memory and recursion were used extensively in the implementation of BLOSIM. While it has not been tried, it should be possible to link Fortran programs (for example, an FFT routine) to a BLOSIM simulation in a UNIX operating system environment.

This paper describes BLOSIM in general terms; more detail can be found in the user's manual [2]. The sections that follow describe BLOSIM from the general to the specific. Section II gives an example of a BLOSIM simulation, illustrating the capabilities and approach of the simulation language. Section III describes in general fashion the features of BLOSIM which affect the user. Section IV discusses how the user specifies the topology of interconnection of blocks, and defines the hierarchical definition of blocks in BLOSIM. Section V describes how the user programs the functionality of a block, and Section VI briefly explains how parameters are passed to blocks. Section VII mentions several enhancements to BLOSIM which have been provided in response to user feedback, and Section VIII gives a brief programminging example to give the reader a better idea of how a BLOSIM simulation is programmed. Finally, Section IX describes user experience with BLOSIM and suggests extension of its capabilities.

II. Example of a BLOSIM Simulation

This section illustrates by way of example the approach that a programmer using BLOSIM uses to develop a simulation. Fig. 1 shows a block diagram of a simulation which could be performed using BLOSIM. The system being simulated is a full duplex data transmission system, consisting of two data transmitters, one at each end of the system, a single data receiver (simulating both receivers is not necessary), and the channel. A detailed description of this type of system is given in [3]. For purposes of simulation, as well as actual implementation, the system is partitioned into a number of blocks. Section IV will illustrate how each of these blocks can be further subdivided. Each block is simulated by a user-provided program, and BLOSIM combines these individual simulations into a simulation of the entire system.

An example of multiple instances of blocks is the filt() block in Fig. 1, which is used a couple of times where a filtering function is required; another example is the block transmitter(), which is used for both the near and far data transmitters. Each instance of a filter is specialized by passing an impulse response as a parameter, or alternatively the impulse response can be stored in a file and read by the block routine from that file. As an example of the latter, where filt() is used to simulate a transmission line the companion program LINEMOD [4] can be used to determine the impulse response of the transmission line, which LINEMOD stores in a file. This same file can then be read by a BLOSIM filter simulation block at run time to provide an accurate simulation of the transmission line.

The most common method for interconnecting blocks in

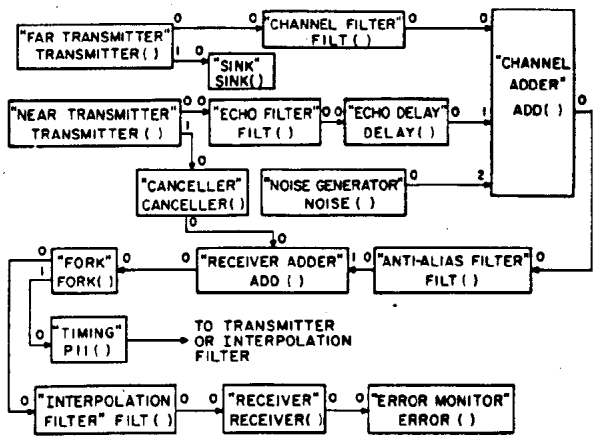

Fig. 1. BLOSIM simulation of a full-duplex data transmission system.

a simulation programmed without the aid of BLOSIM would be to have each block routine obtain data from its predecessor via a function (subroutine) call. This method has several drawbacks, including an unfortunate implementation dependence of one block on another as well as the undesirable embedding of the topology of interconnection in the routines themselves. In BLOSIM, the blocks are connected using intermediate first-in first-out (FIFO) buffers. These buffers are set up and managed by BLOSIM, rather than the user programs.

This use of FIFO's serves several purposes. First, the FIFO's provide a standard interface between blocks. A given block is written to deal with a standard FIFO interface, rather than directly with another block. This standard FIFO interface serves to hide from the rest of the blocks the internal implementation of a given block. Take, for example, a filter block. It can be implemented, for example, as a sample-by-sample recursion which takes one input sample at a time and generates one output sample at a time, or it can be implemented as a fast convolution, which operates on a block of input samples to generate a block of output samples. With the intermediate FIFO's, these details of internal implementation are completely hidden from the blocks connected at the output. Further, the connection of blocks by FIFO's enables the use of different, and even asynchronous and time-varying sampling rates in the system. This will be illustrated in Section II-B, and is extremely useful for the simulation of interpolation, decimation, and the acquisition of phase-locked loops. Finally, by having the user block routine interface to BLOSIM-maintained FIFO's rather than to other blocks, the topology of interconnection can be specified in a single topology definition routine, where it is simple to make rearrangements and changes.

Fig. 2 illustrates, for the example of Fig. 1, how different sampling rates can be accommodated. Important for this particular simulation is the fact that even asynchronous sampling rates can be accommodated. Fig. 2 is a simplified block diagram of the full duplex modem, with the sampling rates written next to the FIFO's.

It is assumed that the "Data" block generates bits at rate f_a in the local transmitter and f_b in the remote transmitter,

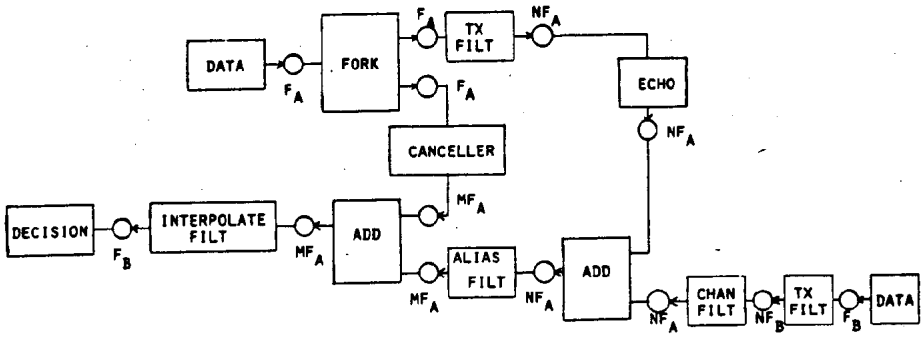

Fig. 2. Sampling rates in asynchronous full-duplex data transmission.

and that these rates are asynchronous. At the output of the local transmit filter, the bandwidth exceeds half the bit rate, and thus the sampling rate must be greater than the bit rate. Thus, assume that the sampling rate is $N \times f_a$, for an integer N. In terms of the implementation of the filter in BLOSIM, this means specifically that the filter block generates N output samples for each input sample.

Similarly, the output of the remote transmit filter has sampling rate $N \times f_b$. In the receiver, after the first "Add" block, a superposition of both local and remote data signals must be represented by the same sample stream. In fact, the "Add" block, which simply performs a sample-by-sample addition of the two input sample streams, requires that they have the same sampling rate. Hence, there is the need to change the sampling rate for the asynchronously sampled remote data signal prior to its connection to the "add" block. This service is performed by the channel filter block, which has sampling rate $N \times f_b$ at the input and $N \times f_a$ at the output. In simplistic terms this means that the block generates f_a/f_b output samples for every input sample on average. In practical terms, this block is more difficult to program than a filter with synchronous sampling rates, and requires a pair of additional inputs which are the time intervals between adjacent input and output samples.

This example illustrates how the most natural sampling rate can be chosen at each block interface, rather than choosing a common sampling rate for all the block interfaces (as is often done in simulations of this type). For example, since the bits at the output of the "Data" block are generated at rate f_a, it is natural to choose a sampling rate f_a at the output, rather than, say, $N \times f_a$.

It is important to note from this example that the relative sampling rates in the system are completely controlled by the internal implementation of the blocks. For example, a doubling of the sampling rate results from the input to the output of a block when the block generates two output samples for every input sample. BLOSIM does not check for the consistency of sampling rates in the system; that is the responsibility of the programmer and user of the block routines.

III. Program Structure

This section describes the structure of the BLOSIM program itself, particularly as it relates to the user interface. As shown in Fig. 3, the user provides two types of routines, the topology and block routines. These routines interface to BLOSIM (rather than directly to one another), which consists of the kernel and the FIFO management routines.

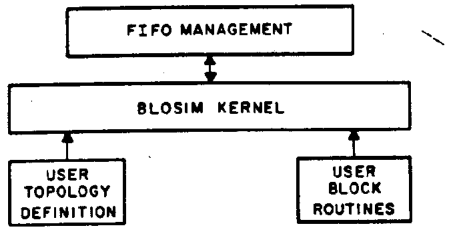

Fig. 3. Program structure of a BLOSIM simulation.

A. User Topology Definition

A user-supplied "topology definition" program specifies the names of the blocks, the routine which implements each block, and how the blocks are interconnected. Further detail on this program is given in Section IV. The topology definition routine is also able to pass parameters to the blocks to specialize their function, a capability which is described in Section VI.

B. User Block Routine

For each block in the system, the user must provide a routine which defines the internal functionality. As explained in Section IV, there are two ways in which this functionality can be specified. Of concern here is the method in which the functionality is simply embedded in a C program, called the "user block routine," which gets input samples from input FIFO('s), and puts its generated output samples into output FIFO('s). In addition, the user block routine can access environmental factors such as how many input or output FIFO's are connected to the block, the number of samples currently on an input or output FIFO, or the maximum number of samples an input or output FIFO can hold.

Another service provided by BLOSIM is the permanent storage of internal state variables for the block. It is necessary for BLOSIM to store these state variables, rather than the "block routine" itself, since there may be multiple instances of a given block routine, and it would be a grievous error for them to share common state variables.

C. Other BLOSIM Services

Aside from the services described above, such as the creation of blocks and their interconnection, this section describes other features of the program of concern to the user.

The FIFO management routines take care of the creation and management of FIFO's. They keep track of the number of samples on a FIFO, put samples on FIFO's, and take samples off FIFO's. The storage for FIFO contents is allocated dynamically at the time of FIFO creation, so that there is no BLOSIM imposed limitation on the total storage required for the FIFO's (a limitation may be imposed by the operating system). All other memory for the simulation is also allocated dynamically at run time, thereby imposing no limitation on the number of blocks, the size of parameter or state variable storage, etc. Conversely, no more storage is allocated than is required, maintaining maximum run time execution speed.

The other major service provided by BLOSIM is the determination of the order of execution of the blocks. In terms of the correctness of the simulation, the order of execution of the blocks does not make any difference. The FIFO discipline insures the proper synchronization of the blocks, since a block will find an input FIFO empty if it attempts to execute before the requisite data samples have been generated by another block. However, some efficiency is lost if blocks are often called when an input FIFO is empty and no useful data samples are available. To alleviate this source of inefficiency, BLOSIM automatically schedules the blocks to execute in the order of "signal flow," as determined by the topology of interconnection. This means that when a block is executed, all the blocks which are connected to its input have been given the opportunity to execute since the last execution of that block, and data are likely to be available in the input FIFO's.

BLOSIM executes the blocks in the order determined by the scheduling described above. Before a block is executed, it is not checked whether the block has available samples on all its input FIFO's, and room for samples on all its output FIFO's, since there are instances where a block could still do something useful under those circumstances (consider, for example, a delay block, which can generate output samples even when its input FIFO is empty!). However, BLOSIM does note whether the block successfully accesses an input or output FIFO. When a block does not access a FIFO, or on every attempt to access a FIFO finds that it is empty (in the case of an input FIFO) or full (in the case of an output FIFO), that block is said to be deadlocked. The simulation is automatically terminated when all of the blocks in the system are deadlocked. Thus, the criterion for termination of the simulation can be put into a single block, such as the data generator "data" in Fig. 1. Shortly after this block stops generating samples, every block in the system will deadlock and BLOSIM will terminate the simulation.

More detail on the BLOSIM scheduling algorithm will be given in Section IV.

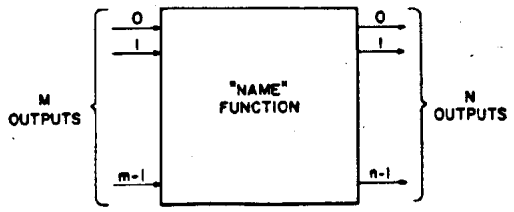

Fig. 4. Block abstraction.

IV. More on User Topology Definition

This section describes in more detail the definition of topology of interconnection of the blocks in BLOSIM. In particular, BLOSIM enables the user to specify a hierarchical definition of blocks, in which blocks can be made up of interconnections of other blocks. The motivations for this feature, as well as the details of how it works, are described in this section.

One detail which is omitted from this section is the passing of parameters to blocks. For ease of understanding, this aspect of the topology definition is deferred to Section VI.

A. The Block Abstraction

Except for the passing of parameters, the internal structure of a block is not relevant during the topology definition phase of the simulation. Hence, for purposes of the topology definition, a "block" is an abstraction as illustrated in Fig. 4. This block abstraction consists of a name, which is a character string, and a user provided function, called the "user routine," which specifies the internal structure or implements the block. Both the name and a pointer to the function must be specified during the topology definition. In addition, a block abstraction has an arbitrary number m inputs, which must be numbered consecutively from 0 to $m-1$, and n outputs which must be numbered consecutively from 0 to $n-1$.

B. The Cosmology of Blocks

Pursuing the example of the data transmission system of Fig. 1, it is desirable to further subdivide the system into smaller blocks for actual programming. This could be done by simply defining blocks with a finer granularity, but BLOSIM provides a more structured method.

BLOSIM allows the user to define the internal structure of a block in terms of an interconnection of other (smaller) blocks. In terms of the block abstraction defined in Section IV-A, these "composite" blocks are identical to blocks which are directly implemented by user provided routines, and can be used in exactly the same way.

The terminology that is used in BLOSIM in describing the different types of blocks is modeled after cosmology: a block which is elementary or atomic in the sense that it is implemented directly by a user-provided routine is called a STAR. A block which is defined as a connection of other blocks, on the other hand, is called a GALAXY. Finally, BLOSIM defines a special GALAXY which encompasses

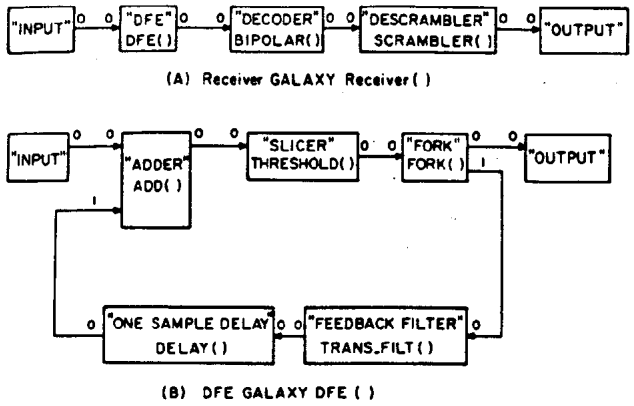

Fig. 5. Internal structure of the receiver GALAXY block in Fig. 1.

the entire system, and which therefore has no inputs or outputs, as the UNIVERSE. By definition, the UNIVERSE unlike all other blocks can only have a single instance, and a routine defining the topology of the UNIVERSE, universe(), must be provided.

The cosmology of blocks differs from ordinary cosmology in that a GALAXY need not contain exclusively STAR's, but rather can contain other GALAXY's. There is no limit on the number of allowed nestings of GALAXY's.

Viewed in these terms, Fig. 1 is the UNIVERSE for the system being simulated. Fig. 5(a) illustrates how the "receiver" can be structured as a GALAXY made up of several STAR's, as well as another GALAXY, "DFE." Fig. 5(b) illustrates the internal structure of "DFE," which is itself made up of several STAR's.

C. Structuring for Efficiency of Execution

It was mentioned in Section II that the order of execution was automatically scheduled for efficiency. The rules by which this is done have implications with respect to the efficiency of execution and should be understood by the user.

The scheduling of the order of execution of the blocks in the UNIVERSE is done without regard to whether the blocks are STAR's or GALAXY's. The rules for this scheduling are as follows. All the blocks are examined to find any which have no inputs. These blocks are scheduled to execute in arbitrary order. At step k, all the blocks which have not been scheduled in steps 1 through $k-1$ are examined to find any which have inputs connected exclusively to blocks which have been scheduled in steps 1 through $k-1$. Blocks satisfying these criteria are scheduled next in arbitrary order. Where deadlock exists, i.e., not all the blocks have been scheduled and the unscheduled blocks do not meet the criteria, the execution of a single arbitrary block is "forced" and the scheduling proceeds as before. These steps are repeated until all blocks within UNIVERSE have been scheduled.

At the execution phase, the blocks in the UNIVERSE are executed in the order in which they were scheduled. The simulation is terminated when all these blocks deadlock, i.e., all fail to successfully access an input or output FIFO.

If the UNIVERSE contains a GALAXY, the blocks within this GALAXY are scheduled to be executed in the same manner as the blocks within the UNIVERSE. Any connection to the input of a block from the GALAXY input is treated as if it did not exist; that is, it is assumed that the FIFO's connected to these inputs have available samples since the GALAXY was appropriately scheduled at the next higher level of the hierarchy. During the execution phase, when a GALAXY block within the UNIVERSE is executed, what actually happens is that the blocks within the GALAXY are executed in the order determined by the schedule until these blocks deadlock. Thus, the definition of deadlock for a GALAXY block within the UNIVERSE is actually a deadlock of all the blocks within that GALAXY. Similar rules are applied to any GALAXY within a GALAXY.

These rules for execution have important implications for the efficiency of the simulation. In particular, where a part of the system contains a feedback loop, it is important to define a GALAXY containing that feedback loop. Consider the case of GALAXY dfe() of Fig. 5(b). Generally, when dfe() is executed, external blocks will have placed a number of samples in the FIFO connected to the input of the GALAXY. However, because of the feedback loop, each block in the GALAXY can only generate one sample per pass through the loop. However, since dfe() is defined as a GALAXY, it will be allowed to execute until the input FIFO is empty, whereas if it was not a GALAXY, each block would only execute once and only one sample on the input FIFO would be exhausted. In the latter case, the entire simulation would be forced into a one sample per pass mode, slowing it down considerably. Thus, defining dfe() as a GALAXY will result in much greater efficiency in the execution phase of the simulation.

V. Programming the User GALAXY Routine

As previously mentioned, there are two types of user routines which must be provided, the user GALAXY routine(s) which specify the topology of interconnection, and the user STAR routines which simulate the functionality of the system. This section describes the primitives provided by BLOSIM for use in the routines defining the internal structure of the UNIVERSE or a GALAXY. The next section repeats this task for the user STAR routines.

Two BLOSIM functions, star() and galaxy(), create a STAR and a GALAXY block, respectively. They must be provided a name for the block (a character string) and a function specifying the internal structure of the block (a function pointer). In addition, as explained in Section VII, they are also provided a pointer to a parameter storage area. The user is obligated to provide the promised routines, and in particular the user must provide a routine universe() which defines the topology of the UNIVERSE.

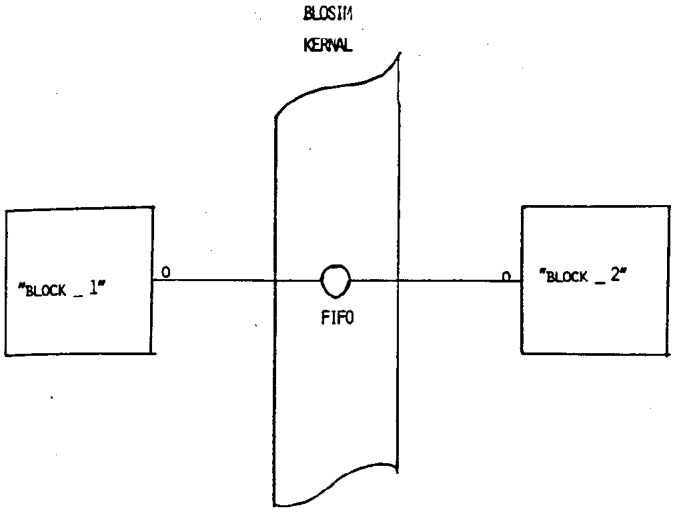

Fig. 6. Block interconnection abstraction.

The user can connect two blocks together, where each is either a STAR or GALAXY, using a supplied routine connect(). Provided as arguments to this routine are the names of the two blocks to be connected, the numbers of the output and input connection, respectively, and the length of the FIFO to be supplied in the connection. When the connection is between a block and the output of the GALAXY that the block is contained in (which cannot be the UNIVERSE), then by convention the name of the block to which the connection is made is "output" (and similarly "input" for an input connection). These rules imply that "input" and "output" are not legal names for blocks.

Finally, it should be noted that only a single FIFO is created even though a connection may be defined through a hierarchy of GALAXY's. As a result, the FIFO length provided by the user for any connection to "input" or "output" is ignored by BLOSIM (and can in fact be omitted).

VI. Programming the User STAR Routine

This section describes the functions available in programming the user STAR routines.

A. Block Interconnection Abstraction

A user STAR routine is divided into two parts: the initialization part and the part which implements the functionality of the simulation of the corresponding block. The functional part of the routine must interface the FIFO's which are connected to the block. This interface is abstracted as shown in Fig. 6. The STAR which is generating data puts that data in the FIFO to which it is connected using the provided routine put(). The STAR need not know anything about this FIFO, like its location in memory, since these details are handled by BLOSIM. However, the STAR does need to know if the FIFO is full; that is, cannot accept any additional data. The routine put() itself returns a status which indicates whether or not the FIFO is full. The status of any FIFO can also be checked by functions which return the size of the FIFO and the number of samples it currently holds. Similarly a function get() is provided which returns a sample from an input FIFO together with a status indicating whether the FIFO is empty.

A user STAR routine can also determine the number of input or output FIFO's to which it is connected. This enables the STAR, if desired, to be tailored to the number of connections at execution time. Where the number of connections is fixed, this enables the STAR to determine that the expected number of connections were made in the topology definition as a consistency check.

An important service performed by BLOSIM is the maintenance of state variable storage. State variables are those variables whose value must be maintained from one call to the STAR to the text. Static memory cannot be used for storage of these variables, since multiple instances of a block are allowed and these instances cannot share state variables. The STAR therefore stores state variables in a single block of memory (using the "structure" construct of C), and requests this storage on the first initialization call to the STAR. BLOSIM then dynamically allocates this permanent storage, and subsequently passes a pointer to this storage to the STAR on every invocation. Thus, the state variable storage must be anticipated by the STAR on its first invocation. A STAR can later dynamically allocate more storage itself, but it must maintain a pointer to this storage in the state variable storage area allocated by BLOSIM.

B. Use of Macro Processor

All the calls to BLOSIM described in this section are actually macros, which the C macro preprocessor uses to substitute the appropriate code into the STAR in the first stage of compilation. Thus, most calls to BLOSIM from a user STAR function do not involve function calls, but rather result in direct access to BLOSIM data structures. This implementation is new to a later version of BLOSIM, and was chosen to improve the efficiency at run time.

As an indication of the overhead that BLOSIM incurs, two profiles were made of BLOSIM simulations on a VAX 780 computer under the UNIX operating system. The first simulation was a minimal one in which the topology is set up and only 100 samples are generated and printed out. Thus, this case can be assumed to give the minimum memory and execution time for any BLOSIM simulation. The static memory requirements for this run were 27 kbytes and the total execution time was 2.4 s. Of this time, 5.6 percent was used for topology setup, 2.8 percent was consumed by the user STAR routines, and 4.2 percent was taken in the FIFO management routines. The remainder of the time was used by system functions such as the printing, memory allocation, etc.

The second profile was of a moderately large run on a system encompassing most of Fig. 1, performed on 1000 samples. In this case the total static memory requirement was 54 kbytes with an execution time of 98 s. Of this time, an insignificant percentage was devoted to topology setup, 8.4 percent was used by the routine which called the STAR's in the order of the schedule, 16.2 percent was used by the FIFO management routines, and 52 percent was spent in the user STAR routines themselves. Since the remaining system overhead time would be similar for any approach to simulation, this illustrates that the overhead that BLOSIM imposes on a typical simulation has successfully been kept to a minimum.

The use of macros does have one disadvantage which the programmer of a user STAR function must keep in mind: all arguments of macros should be simple variables, rather than expressions. The reason is that all the macros cause the evaluation of any expression which is an argument more than once, which will often have unintended side effects. For example, if the statement

$$a = 2*a$$

as an argument to a macro call were to be executed twice, the result would be unexpected.

VII. Passing Parameters to User STAR or GALAXY Routine

BLOSIM uses a quite general scheme for passing parameters which is similar to the state variable storage scheme. A GALAXY routine which wishes to pass a set of parameters stores them in a block of memory (using the C "structure") and passes a pointer to this memory and an integer containing the size of the storage in bytes to BLOSIM in the star() or galaxy() call in a user topology routine. BLOSIM makes a permanent copy of this storage area, and in each subsequent call of the user STAR or GALAXY routine, passes the pointer and integer as arguments (a user STAR routine has a third argument, a pointer to state variable storage). The size argument is usually used in the user routine to check if the parameter storage is the expected size, but it could also be used as a way to pass a variable number of parameters. This method of passing parameters is completely general—in no way does it restrict the number or size of parameters which can be passed to a routine.

VIII. BLOSIM Enhancements

The preceding describes the basic BLOSIM kernel routines and primitives provided to the user. This kernel is designed to be very general, so as not to limit the potential applications. However, this generality leads to a program which is more difficult to use than is necessary in severa respects. For one, every change in the simulation, even a simple as a change in a parameter, requires a recompila tion. Even though in the UNIX operating system environ ment this process is easily automated, it is nevertheles slow (30–60 s would be a typical real-time). Anothe difficulty is the relatively large amount of code required t interface a user STAR function to BLOSIM. While thi code is very similar from one STAR to another, it repre sents easily half the total code of a typical small STAI routine (and in a system properly partitioned all STAI routines should be small). Section VII gives an illustratio of this interface code.

Fortunately, due to the generality of the BLOSIM kernel it is simple to overlay layers of software which make th facility easier to use (at some loss of generality). As use experience with BLOSIM has been assimilated, a numbe of these enhancements have been implemented, and the are briefly described in this section. It should be emphasize that these enhancements have involved absolutely n change to the BLOSIM program itself, but they rather us the BLOSIM-provided primitives as a basis for buildin more convenient primitives for the user interface. Interna changes to the implementation of BLOSIM have also bee made, but with no impact on the user interface.

A. PARAM—A Parameter Passing Facility

PARAM is a standardized method of passing parameter between blocks in BLOSIM. It is not a program, but rathe an include file "param.h" which can be included in an user STAR or GALAXY function which wishes to use thi style of parameter passing. It is not necessary for a parameter passing to use this facility within a give BLOSIM simulation. PARAM is a specialization of th BLOSIM parameter passing scheme, in that a particula format for the parameter storage block is assumed. PARAM restricts the type of parameters which can be passed t integer and floating point variables and character string However, this restriction is quite useful in that paramete passing becomes more standardized and easier to maste and understand.

PARAM also allows the user to specify a parameter a DEFAULT or NOSET. In the former case, the receivin routine is asked to set the parameter to a default value. A NOSET parameter implies that the sending routine ha deliberately not set the parameter. This parameter type i included for the situation where it is desired to pass variable number of parameters, since the array of parame ters can be terminated with a NOSET parameter.

PARAM uses the C "union" construct to pass parame ters which can be one of several types (basically th allocates storage sufficient for the type which requires th largest storage). It is simple to use because PARAM als defines two useful macros which make it more convenien for the user to address a parameter type and value.

B. FILETOP — Reading Topology from a File

Normally BLOSIM topology information concerning the interconnection of blocks is contained in a set of user GALAXY routines, which are C programs. This implies that whenever a connection or parameter is changed, a recompilation and linking of the BLOSIM object module is necessary.

FILETOP is a replacement for the normally user-provided universe() routine. This replacement reads the topology information from files specifying the topology of the UNIVERSE and each GALAXY. Not only are these files easier to generate than the corresponding C programs, but avoiding a recompilation and linking speeds up the process considerably (if a user STAR routine is changed, however, a recompilation of that STAR and linking is still required).

The format of a FILETOP file is similar to that used by the user STAR function preprocessor utility STARGAZE, to be described in the next section. Both FILETOP and STARGAZE use PARAM as their parameter passing mechanism. FILETOP provides the additional capability to pass a parameter from the input to a GALAXY directly to the block within the GALAXY.

The format and use of FILETOP will be illustrated by the programming example in Section IX.

C. STARGAZE — A STAR Routine Preprocessor

User STAR functions for BLOSIM, in addition to implementing the functionality of a simulation, must do a number of things to effect the interface to BLOSIM, include accept arguments, allocate storage for state variables, check the size of parameter storage and the number of input and FIFO's, default parameter values and initialize state variables, etc. STARGAZE is a program which facilitates and automates many of these functions, thereby permitting the user to concentrate on the simulation functionality rather than the interface to BLOSIM.

STARGAZE is a preprocessor for user STAR functions which allows the user to express many of the interface parameters in a form similar to and compatible with FILETOP, and then turns this specification into C code for compilation and execution. It is thus natural for the topology of GALAXY's to be developed using the FILETOP utility. However, it is possible to write user GALAXY routines in C and still use STARGAZE to develop STAR's (or vice versa). It is, however, in that case necessary to use the PARAM utility for passing parameters.

STARGAZE provides a number of control line formats. The user must of course provide in the input file the C program which implements the functionality of the user STAR function. This C program generally includes three parts: declarations of variables, initialization code which is to be executed the first time the user STAR routine is called by BLOSIM, and the main body of code which is executed every time the user STAR function is called. This code performs get()'s and put()'s and processes the samples. Each of these three blocks of code is delimited by a keyword line; for example, the declarations code is delimited by a line containing the word "declarations" and a line with the word "end." This will be illustrated in the programming example in Section IX.

The PARAM and FILETOP utilities have received wide user support, but many users have preferred to develop their own STAR routines from scratch because of the enhanced feeling of control. An alternate approach to reducing the STAR interface code is therefore planned. This approach would work like FILETOP, in that initialization parameters (like the number and type of parameters, initialization of parameters, number of FIFO's, etc.) would be stored in a file associated with the STAR. A provided routine could then be invoked by the STAR during initialization, and this routine would read this file and perform the necessary actions.

D. AUTOMAKE — BLOSIM Run Time Support

Running a BLOSIM simulation requires the compilation and linking of the user STAR and GALAXY routines together with the BLOSIM kernel routines. Keeping track of all the files and routines involved in a given simulation, which might typically be as high as fifty, would be difficult. Fortunately, most users have run BLOSIM in a UNIX operating system environment. UNIX provides an important tool for keeping track of all the routines involved in a simulation, the "make" command. This command reads a file, called the "makefile," in which all the steps required for the compilation and running of the simulation can be stored, and automatically executes these steps where required.

Even with the "make" facility, it is necessary to keep the "makefile" updated as STAR's are added or deleted from the simulation. Even this step has been eliminated by AUTOMAKE. AUTOMAKE requires the exclusive use of the FILETOP facility for topology specification, and automatically examines the user's FILETOP topology files to find the names of all the STAR's involved in the simulation and invokes the "make" command.

Running a BLOSIM simulation using AUTOMAKE is thus a single step process: after editing a STAR routine source code, or making any other change, the single AUTOMAKE command is executed. All the subsequent steps through the actual execution of the object code then proceed automatically. AUTOMAKE, unlike the remainder of BLOSIM, is UNIX operating system specific.

IX. Programming Example

Readers not familiar with the C language and UNIX operating system may not have a good idea of how a

```
#Galaxy implements receiver for full-duplex data transmission system

#Create the blocks in the Galaxy

#Parameters for dfe()
param int 2  #Number of taps
param array 2 3.4 2.2  #Tap weights
galaxy dfe dfe.c

star decoder bipolar.c
star descrambler scrambler.c

#Connect the blocks with fifo lengths 100
connect input 0 dfe 0
connect dfe 0 decoder 0 100
connect decoder 0 descrambler 0 100
connect descrambler 0 output 0
```

Fig. 7. FILETOP input file for receiver() GALAXY of Fig. 5(a).

```
param samples_delay int

no_input_fifos 1
no_output_fifos 1

state zero_count int initialize 0

declarations

        SAMPLE x;
        int return_code;

end

main_code

        /* Put out zero samples until zero_count = delay */

        while(zero_count < samples_delay) {

                /* try to put out a zero sample */
                return_code = put(0,0.0);

                if(return_code >= 0)

                        /* zero was put out, so increment zero_count */
                        ++zero_count;

                if(return_code != 0)
                        return(0);  /* output fifo now full */
                }

        /* enough zeros have been put out, so
           move samples from input to output fifo */

        /* first make sure the output fifo is not full before
           we get a sample from an input fifo */

        if(length_output_fifo(0) == maxlength_output_fifo(0))
                return(0);  /* output fifo full */

        while(get(0,&x) >= 0) { /* get sample from input fifo */

                if(put(0,x) != 0) /* put sample on output fifo */
                        return(0);  /* output fifo full */
                }

        return(0);  /* input fifo empty */

end
```

(a)

Fig. 8. Input to STARGAZE for block delay(). *C* program for delay(), which is output of STARGAZE.

```
#include "type.h"
#include "star.h"
#include "param.h"
#include <stdio.h>
typedef struct {
        int z_ero_count;
        } STATE,*STATEPTR;
#define samples_delay param_value(0,d)
#define zero_count pstate->z_ero_count
delay(pparam,size,pstate,pstar)
PARAMPTR pparam;
int size;
STATEPTR pstate;
STARPTR pstar;
{

        SAMPLE x;
        int return_code;

        if(pstate == NULL) {
                pstate = (STATEPTR) alloc_state_var(1,sizeof(STATE));
                if(size != 2*sizeof(PARAM))
                        return(200);
                if(param_type(1) != NOSET)
                        return(201);
                if(param_type(0) == INT)
                        ;
                else return(202);
                if(no_input_fifos() != 1)
                        return(203);
                if(no_output_fifos() != 1)
                        return(204);
                zero_count = 0;
                }

        /* Put out zero samples until zero_count = delay */

        while(zero_count < samples_delay) {

                /* try to put out a zero sample */
                return_code = put(0,0.0);

                if(return_code >= 0)

                        /* zero was put out, so increment zero_count *
                        ++zero_count;

                if(return_code != 0)
                        return(0);  /* output fifo now full */
                }

        /* enough zeros have been put out, so
           move samples from input to output fifo */

        /* first make sure the output fifo is not full before
           we get a sample from an input fifo */

        if(length_output_fifo(0) == maxlength_output_fifo(0))
                return(0);  /* output fifo full */

        while(get(0,&x) >= 0) { /* get sample from input fifo */

                if(put(0,x) != 0) /* put sample on output fifo */
                        return(0);  /* output fifo full */
                }

        return(0);  /* input fifo empty */
```

(b)

Fig. 8. (*Continued.*)

simulation is developed using BLOSIM even after reading the preceding sections. For this reason, a short programming example is included here.

Assume that it is desired to simulate the system of Fig. 1. Fig. 7 is a FILETOP input file which specifies the topology of the UNIVERSE. The commands in this file are self evident.

As an illustration of the programming of a STAR routine, consider the simple star function delay(), which simply takes samples off a single input FIFO and delays them by a specified number of samples. The delay function surprisingly does not require internal storage, but is implemented by putting out a number of zero-valued samples equal the delay before accessing the input FIFO's. The file which is used as the input to STARGAZE is shown in Fig. 8(a). The resulting C program, which is similar to what the user might develop from scratch, is shown in Fig. 8(b).

X. User Experience and Conclusions

BLOSIM has been in use for about a year, and has been used in several companies and several academic environments. Most of the users have been former Fortran programmers accustomed to developing simulations fro

scratch. While programming in a modern structured language like C is, in the author's opinion, of long-term benefit, it is not an especially easy transition for Fortran programmers, especially in a programming environment like BLOSIM which uses virtually all the facilities of C. Thus, there has been generally a transition period of one to two months during which the users have first accustomed themselves to C and then experimented with BLOSIM. Once users have overcome this (in some cases painful) transition, they have almost always been very pleased with the results. The increased productivity and greater flexibility in trying different simulation configurations is evident to most users. At least one large simulation effort has utilized STAR routines programmed by at least six persons at different times, and no difficulties have been encountered in interfacing these routines. Some of the STAR routines have been ported to industry, again without difficulty.

It is hoped that this experience with BLOSIM will have impact on the methodology used in simulation efforts. In particular, it illustrates the benefits of using structured programming methods and environments, particularly in complex programming efforts. If a simulation vehicle such as BLOSIM were widely used in the simulation of communications and signal processing systems, the sharing of routines throughout the research and development community would be much more practical and effortless than it is today.

REFERENCES

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.
[2] D. G. Messerschmitt, "BLOSIM—A block simulator, version 1.1," Univ. California, Berkeley, CA, int. memo.
[3] O. Agazzi, D. A. Hodges, and D. G. Messerschmitt, "Large-scale integration of hybrid-method digital subscriber loops," *IEEE Trans. Commun.* p. 2095, Sept. 1982.
[4] D. G. Messerschmitt, "A transmission line modeling program written in C," *IEEE J. Select. Areas. Commun.*, this issue, pp. 148–153.
[5] B. Gold and C. M. Rader, *Digital Processing of Signals*. New York: McGraw-Hill, 1969, ch. 5.
[6] J. L. Kelly, C. L. Loshbaum, and V. A. Vyssotsky, "A block diagram compiler," *Bell Syst. Tech. J.*, p. 669, May 1961.
[7] C. M. Rader, "Speech compression simulation compiler," *J. Acoust. Soc. Amer.*, June 1965.
[8] B. Karafin, "A sampled-data system simulation language," in *System Analysis by Digital Computer*, F. Kuo and J. Kaiser, Eds. New York: Wiley, 1966.

David G. Messerschmitt (S'65–M'68–SM'78–F'82) received the B.S. degree from the University of Colorado, Boulder, in 1967, and the M.S. and Ph.D. degrees from the University of Michigan, Ann Arbor, in 1968 and 1971, respectively.

He is a Professor of Electrical Engineering and Computer Sciences at the University of California, Berkeley, where he has been since 1977. From 1968 to 1977 he was a member of the Technical Staff and later Supervisor at Bell Laboratories, Holmdel, NJ, where he did systems engineering, development, and research on digital transmission and digital signal processing (particularly relating to speech processing). His current research interests are analog and digital signal processing, speech processing, digital transmission, multiprocessor approaches to signal processing and circuit simulation, digital subscriber loop transmission, and adaptive filtering. He has published over 50 papers and has 10 patents issued or pending in these fields. Since 1977 he has also served as a consultant to a number of companies.

Dr. Messerschmitt is a member of Eta Kappa Nu, Tau Beta Pi, and Sigma Xi. He is also currently a member of the Communication Theory Committee and the Board of Governors of the IEEE Communications Society. He has also organized and participated in a number of short courses and seminars devoted to continuing engineering education. He is presently serving as Director of the Industrial Liaison Program of the Department of Electrical Engineering and Computer Sciences at the University of California.