A SOFTWARE ENGINEERING ORIENTED COMPARISON OF SIMULATION LANGUAGES

John A. Miller, Orville R. Weyrich, Jr. and David Suen
Department of Computer Science
University of Georgia

ABSTRACT

This paper examines the process-oriented features of simulation languages. In particular it compares powerful general purpose simulation languages with a simple extension to a conventional programming language. Languages that support concurrency and modularization facilitate both language extensibility and process implementation. We extend the capabilities of Modula-2 by providing modules that handle the details of simulation. The encapsulation features of Modula-2 modules hide these details from users of these modules, thus making a Modula-2 based simulation system easy to learn and use.

INTRODUCTION

In this paper we consider alternatives in the development of discrete-event simulation projects. Specifically, we analyze languages and tools that enable reasonably rapid development of process-oriented simulation models. We consider only alternatives that are general purpose and are widely available. Hence, for comparison purposes, we picked the two most popular general purpose simulation languages, SLAM II and SIMSCRIPT II.5, and a general purpose high level language that supports concurrency and modularization. Concurrency is necessary for implementing the process-interaction approach, while modularization is needed to provide safe and powerful tools. In particular we choose Modula-2 for its simplicity and low cost. [We plan to analyze Ada in future studies. We did not consider GPSS to be general purpose enough, and SIMULA is not widely available in the United States [Law 82, Bank 84].]

As a point of reference, we present a virtual memory simulation model as a simple running example. Demand for pages will follow the principle of locality. The goal of the simulation is to compare two page replacement strategies under variations in demand. Specifically, the typical LRU (Least Recently Used) strategy is compared with the simpler FIFO (First-In First-Out) strategy. The strategies are compared on the basis of job delay and the number of page faults generated.

Analyzing this running example, drawing on our previous experience with simulation projects, and referring to the literature, we compare the languages from a software engineering point of view. Issues we consider include learning time, model development time, model understandability, testability, maintainability, and runtime efficiency. We believe that modelers are best served by a simple and uniform modeling framework. Hence we consider how well various pieces of a simulation model implemented in a language fit together. Are they at the same general level of abstraction? If not, is the interface between the levels narrow and safe, or is it a source of subtle errors?

SIMULATION WITH SIMULATION LANGUAGES

The process-interaction features of SIMSCRIPT II.5 are very good. They certainly provide more features than our Modula-2 simulate module. However, the basic capabilities and approaches are quite similar. SIMSCRIPT has *processes* which are very much like our Modula-2 processes. SIMSCRIPT can create instances of a process specification either explicitly with a CREATE statement, or implicitly with an ACTIVATE A statement. Once a process is executing it continues until it executes a WAIT/WORK statement, a REQUEST statement, or a SUSPEND statement. A WAIT/WORK statement causes the process to reschedule itself for later execution and is analogous to an ACTIVATE DEACTIVATE sequence in our Modula-2 approach. A REQUEST statement involves waiting for a *resource* such as a disk drive. A resource encompasses a queuing component and a service component. The RELINQUISH command releases unit(s) of a resource and causes waiting process(es) to be rescheduled for execution. Our Modula-2 approach has nothing analogous to the high level concept of a resource. Finally, a process may SUSPEND itself depending on some other process to find it on some list and REACTIVATE THE process. This is identical to the way we do things in our Modula-2 approach. The capabilities provided by SIMSCRIPT resources can be achieved by using this low level approach. For example, a process inserts itself on a disk queue and DEACTIVATEs itself. Additionally, SIMSCRIPT has the capability to preempt and resume processes with its INTERRUPT and RESUME statements.

SLAM II provides even higher level constructs to support process-interaction. The network statements provide the capabilities necessary for process-oriented simulations. The network statements essentially define the routing, activities, and transformations that entities undergo as they flow through nodes of a network. As an example, a preliminary version of the virtual memory simulation was coded in SLAM II. Jobs were created at a CREATE node, attributes were assigned to jobs at an ASSIGN node. Jobs then circulated around a service loop (one time for each record access). In this loop an EVENT node generted a call to a Fortran subroutine to determine if the page containing the record was in main memory. If it was, the job branched (ACTIVITY) to the memory QUEUE node ACTIVITY pair. Otherwise, the job branched to the disk QUEUE node ACTIVITY pair. When a job had accessed all the records it desired to, it branched out of the loop to a COLLECT node to gather statistics, and finally terminated by entering a TERMINATE node. Each network statement does what would require several lines of code in a conventional programming language. Hence, if the network statements nicely fit your problem, SLAM II can be used to implement concise very high level programs.

PROCESS-INTERACTION APPROACH

All discrete-event models have entities moving through a system performing certain activities or receiving service from components of the system. Hence the problem of developing a simulation model reduces to the problem of specifying what happens to entities in the system. The most common approach to developing discrete-event simulation models is the *event-scheduling* approach. In this approach one specifies the changes or transitions that entities undergo. Changes, which are modeled to occur instantaneously, are encoded as procedures that update the status of the system and the entity(ies) involved. An example would be a procedure or event routine to remove an entity from a queue to begin service. In a sense the code for the model hides what is happening in the system. One does not "see" an entity being served, only the endpoints (i.e., the beginning and ending of service) are explicitly visible. But of course the modeler is smart enough to make the inference necessary to see the whole picture.

An alternative which paints a better more coherent picture of what is occurring is the *process-interaction* approach. With this approach the modeler focuses on the behavior of entities and writes process specifications to indicate the sequence of activities that an entity performs. For example, a customer arrives at a bank, waits in a queue, receives service from a teller, and finally departs from the bank. The advantages of this approach are that it is easier to follow what is happening to entities, and the higher level nature of this approach usually results in less coding and development time. Indeed, the number of lines of code for the process-interaction approach can be much less than that for the event-scheduling approach.

The disadvantages of the process-interaction approach are that it may be a little less efficient in terms of execution time; people are more familiar with the event-scheduling approach; and it requires computing languages that support concurrency to implement it. Until recently however, there were no widely available general-purpose languages supporting concurrency. The availability of good compilers for Modula-2 and Ada has totally changed this situation. The tasking and rendezvous features of Ada provide powerful high level support for concurrency, while the more limited coroutine capability of Modula-2 is well suited for implementing processes in a simulation model (more about this in the next section). Because we feel the advantages outweigh the disadvantages, and as computer scientists use the concept of a process frequently, we consider only the process-interaction approach in this paper.

COMPARISON OF PROCESS ORIENTED FEATURES

In this section we will compare the three languages on the basis of how they implement basic types of process interaction. Focusing on how process activities affect the time advance mechanism, allows one to characterize four basic types of process interaction. Each type is described in a subsection below.

Definite Delay

A definite delay is an unconditional wait where the reactivation time is known in advance. The delay is set up by the process itself. An example from the virtual memory simulation is a process beginning to be served by the paging disk. The service is simulated by having the process delay itself for a period of time sampled from Uniform(10, 30). In SIMSCRIPT II.5 a definite delay is induced by executing a WAIT/WORK statement. This causes a process to put itself on the Future Event List before giving up execution, allowing some other process to run. In Modula-2, we accomplish this with an ACTIVATE statement followed by a DEACTIVATE statement. The ACTIVATE statement puts the process on the Future Event List at a sampled time in the future, and then the DEACTIVATE statement allows some other process (implemented as a coroutine) to execute. Finally in SLAM II, a definite delay is achieved with an ACTIVITY statement. A regular activity induces a definite delay for any entity reaching it. A service activity behaves similarly except that only a limited number of entities can be serviced at one time (the rest wait in a preceding QUEUE node).

Low Level Indefinite Wait

An indefinite wait is a conditional wait where the reactivation time is unknown. Indeed, if no other process sees this process (e.g., on the disk queue) then the process will wait forever. Low level facilities do not couple this type of interaction with a higher level object, such as a resource. In SIMSCRIPT II.5 such a wait is accomplished by a process executing a SUSPEND statement. Note, if the process does not FILE itself on some list before issuing the SUSPEND statement, it will be lost forever. At some later time, another process will (RE)ACTIVATE THE process. For example, a process that releases a write lock on an object should reactivate processes waiting to access the object. Since locking protocols can be complex, it is best to use low level facilities and explicit coding as was done in a SIMSCRIPT II.5 simulation of database protocols by Griffeth and Miller [Grif 85]. In Modula-2, we accomplish such a wait by Inserting a process on some list or queue, and then executing a DEACTIVATE statement. At some time in the future, another process will find it in the list or queue and decide to reactivate the process by executing an ACTIVATE statement. In SLAM II, the flexible use of this low level approach can only be achieved by resorting to coding Fortran event routines.

High Level Indefinite Wait

A high level indefinite wait is similar to the previous type, except that more structure is introduced. The unknown delay involves waiting for a resource (a server or some other system object) that may be held by other processes. This high level approach has the advantage that the details of queuing, being selected for service and releasing servers are easily handled with a couple of statements. However, for objects with complex protocols such as locks, the greater flexibility provided by low level approach is necessary. In SIMSCRIPT II.5, this high level facility is provided by resources in conjunction with the REQUEST and RELINQUISH statements. The REQUEST statement will attempt to acquire a specified number of units of a resource for a process. If that amount is unavailable, the process is queued. When a process no longer needs units of a resource, it RELINQUISHes them. This causes waiting processes whose REQUEST can now be satisfied to be rescheduled for execution. As was mentioned earlier, our Modula-2 simulate module does not provide this capability, so that longer low level code segments that do the same thing are required. In SLAM II this capability is provided in two flavors. When the resource is a server(s), preceding it with a QUEUE statement provides the capability, while more generally, AWAIT and FREE statements are analogous to SIMSCRIPT's REQUEST and RELINQUISH statements.

Process Preemption

If a process is in the middle of an unconditional wait, say corresponding to being in service, then another process can preempt it, taking the server away from it. Later the preempting process can reschedule the preempted process allowing it to complete its service. In SIMSCRIPT II.5 process preemption is accomplished by a process INTERRUPTing another process that is in the middle of a WAIT/WORK statement. The time remaining is saved so that the interrupted process can be RESUMEd where it left off. At present our Modula-2 simulate module has no facilities to support this. The simulate module could, however, be extended to include such facilities. In SLAM II an entity in the middle of an activity can be preempted by another entity with higher priority executing a PREEMPT statement. When this entity later executes a FREE statement, the preempted entity will be resumed.

COMPARISON ON THE BASIS OF SOFTWARE ENGINEERING PRINCIPLES

Modula-2

Rather than discussing the particular merits or demerits of our particular implementation, let us broaden the discussion to considering the suitability of using Modula-2 as a basis for building process-oriented simulation models. The essential requirement for implementing simulation processes is to have a language construct that can be suspended and later resumed where it left off. Procedures do not meet this requirement. However, coroutines like those in Modula-2 do. Additionally, dynamic calling sequences are well supported in Modula-2 (e.g., when a coroutine DEACTIVATEs it transfers control to whatever coroutines happens to be at the front of the Future Event List). One thing procedures can do that Modula-2 coroutines can not do, is pass parameters. However, just as SIMSCRIPT II.5 has attributes to pass information between processes, we have implemented this facility in Modula-2 by having a function Attributes return a pointer to an attribute record. For flexibility the type specification for an attribute record can be easily tailored to the user's needs, since it is in its own little module.

In summary, we feel that coroutines as implemented in Modula-2 provide a very good basis for process-oriented simulations, they are easy to learn, easy to use, and are run time efficient (e.g., in comparison to SIMSCRIPT II.5). A weakness we noted is that our implementation of Sun Modula-2 is lacking in coroutine safeguards. For example, if one does not allocate enough space for a coroutine, it will be quite content to use other coroutines' or the main program's address space, causing havoc. We also implemented our simulation system on a personal computer using Logitech's Modula-2. When not enough space is allocated for a coroutine it more appropriately raises a stack overflow error. Examining Wirth's writing on Modula-2, we did not see where he discusses safeguarding the address space of coroutines.

SIMSCRIPT II.5

Because of its high level nature and English sentence-like constructs, we find SIMSCRIPT II.5 to be a very readable language, even for nonprogrammers. One unique aspect to SIMSCRIPT is its flexible syntax (e.g., its optional A, AN, and THE word, choices for keywords, and the ability to redefine words and symbols). One can use this flexibility to write programs that are very English-like. In order to highlight the fundamental similarity between our Modula-2 implementation and SIMSCRIPT, we gave our program a Modula-2 flavor. Since

SIMSCRIPT code is readable and somewhat modular, program modification and maintenance are reasonably well supported. SIMSCRIPT supports subprogram units and Fortran-like independent compilation. However, it does not provide Modula-2-like separate compilation which does checking between separately compiled units [Wirt 85].

From the point of of model development, SIMSCRIPT is a very nice tool. It does however have a couple of weaknesses, namely efficiency and learnability. In terms of efficiency, SIMSCRIPT is 8 to 9 times slower to compile than Modula-2. Specifically, simc required 44 seconds to compile a 253 line, 7522 character SIMSCRIPT program, while m2c required 5 seconds to compile a 360 line, 10853 character Modula-2 program. More importantly, SIMSCRIPT is 4 to 5 times slower to execute than Modula-2 (see Tables 1 and 2). [Version 1.01 of UNIX SIMSCRIPT II.5 [CACI 87] and Version 1.0 of Sun Modula-2 [Sun 86] were run on a Sun-3/280 computer/workstation.] Perhaps these comparisons are unfair since very high level languages are slower than high level languages, which are slower than assembly languages. Additionally, the timings were done on only one type of computer. Nevertheless, since simulation models tend to run a long time, execution efficiency is an important concern. Futhermore, workstations with their nice programming environment and high resolution graphics seem an ideal tool for simulation modeling. Finally, as far as learnability is concerned, the situation is analogous to Ada. The language in its totality is formidable, providing a substantial assortment of powerful features. However, if programmers can be taught a well-defined usable subset then the complexity of the language is less of a problem. On the positive side SIMSCRIPT does provide the standard structured programming constructs (IF, FOR, WHILE, SELECT CASE).

SLAM II

In a sense the SLAM II network facilities are not a programming language like SIMSCRIPT or Modula-2, but more of a network flow description of entities. Indeed when designing a SLAM II simulation model, one first produces a network diagram and then mechanically translates it into network statements. Hence, the program is more akin to a textual specification of a network than a programming language. Here belies the beauty and the beast of SLAM II. Basing the model on a high level network specification provides a powerful concise way of producing simulation models. However, the model is best understood in graphical sense. The network statements with their terse nature and plethora of optional parameters can be hard to read and understand when SLAM II programs get long. Hence, we feel it would be nice to view the network diagram as the code and the network statements as a form of computer readable intermediate code. The TESS simulation support tool helps facilitate this. A user can enter and connect icons on a screen to create a computer readable graphical representation of the model, which is translated by TESS into network statements.

With the TESS interface the SLAM II network features provide a convenient understandable very high level tool for generating simulation models. One might say that it is analogous to building prefabricated houses. The pieces are already built, so all you need to do is fit them together. All well and good unless you need something a little different or special. For example, in the projects done in our simulation course, 90% of the model could be handled with network statements. It was the last 10% of patchwork that was the killer. To handle this, there exist two possibilities. First, many more types of network statements (most very specialized) can be added to SLAM II. Second, the network statements can be augmented with another facility. We believe that this facility should fit nicely with the conceptual high level nature of the network, and that it should be very secure with substantial error checking power. We believe that the current incarnations of Fortran are not a good choice for this. Because of the wide base of Fortran programmers it is probably best to keep it as an option. However, for those desiring a more uniform conceptual approach it would be nice to have a simpler higher level language tool to use instead of Fortran. We found that our students had difficulty translating in their heads between the high level conceptualizations of the network and the low level unchecked features of Fortran. Indeed, most of their errors involved the mixing of the two. For one, they had trouble with modes with the equivalenced array NSET/QSET, possibly because our students are spoiled by using languages whose compilers' do the type or mode checking for them. Additionally, we feel that adding features to the network statements for modularization would be helpful.

SIMULATION WITH A GENERAL PURPOSE LANGUAGE, MODULA-2

The coroutine feature of Modula-2, although low level, provides what is needed to implement processes for simulation models. Essentially a process runs until it needs to either perform an activity or wait on a resource. At such a time the coroutine can give up execution, allowing some other coroutine to run. The coroutine chosen to run is the process that should be run next in the simulation. This is accomplished by maintaining *pointers to process descriptors* (ELEMENT) on a Future Event List, and using a TRANSFER statement to cause a context switch. The address of the coroutine is one of the process descriptor's fields. The complexity of both the Future Event List and the coroutine transfers is hidden from the user by being encapsulated in a module.

The module simulate provides the basic capabilities to do process-oriented simulation. The definition module, simulate.def, evolved from the definition module given in [Hugh 87]. Our module provides procedures to CREATE, ACTIVATE, DEACTIVATE, and TERMINATE processes. They can be viewed by users of the module as language extensions, a simulation language built on top of Modula-2. Once a process is created, it can be activated or scheduled for execution by placing it on the Future Event List according to its future execution time and its priority. Scheduling options include scheduling the process to run NOW (i.e., at the current simulated time), AT an absolute simulated time, or at a DELAY from the current simulated time. Note, the ACTIVATE procedure as we implemented it does not trigger a coroutine transfer. After a process has set up its future (e.g., by activating itself to run at a later time, or by inserting itself on a wait queue) it may DEACTIVATE itself. This causes a transfer from this process to the process at the front of the Future Event List. If the process did not activate itself to run at a later time then it is depending on another process to activate it. For example, when a process finishes service it should activate process(es) on the corresponding wait queue. Finally, when a process completes, the TERMINATE procedure is called causing a coroutine transfer to occur. Note, it is necessary to place a TERMINATE procedure at the end of coroutines, since falling off the end of a coroutine generates a program halt. To initiate a simulation the procedure StartSimulation is called.

In addition to the above fundamentally important procedures the following support procedures are also in the simulate module. They are in the form of functions that retrieve information encapsulated in the simulate module. This safety feature allows the information to be read but not written to. The procedure SimTime returns the current simulated time, and Current returns the currently executing process. The following procedures return information about a process. Status returns the status of a process (whether it is running, scheduled, waiting, or terminated), EvTime returns the time a process is scheduled for execution, and Priority returns the priority of a process. The Attributes procedure returns a pointer to a process' attribute record enabling information transfer between processes. Finally, there are three procedures to provide primitive list/queue handling capabilities. The procedure Insert allows processes to be inserted in a doubly linked list in the front, back, or in time-priority order. The procedure Remove deletes and returns a process from a doubly linked list, either from the front or back. The procedure MakeQ creates a doubly linked list and returns a pointer to its header. This procedure can be used to create waiting queues, and indeed it is also used to create the Future Event List.

The simulate definition module is shown below. Currently, there are two other modules in our Modula-2 simulation system. The module statistics provides random number, random variate (11 of them), and statistics gathering capabilities. Finally, the small module attributes defines the attribute record for an application.

```
DEFINITION MODULE simulate;
  FROM attributes IMPORT ATTRIBUTES;
  TYPE
    ACTKIND = (NOW, AT, DELAY);
    STATUS  = (NONE, RUNNING, SCHEDULED, WAITING,
      TERMINATED);
    POSITION = (ORDER, FIRST, LAST);
    ELEMENT;   (* Opaque - pointer to process descriptor *)

  PROCEDURE CREATE(VAR process: ELEMENT; proc: PROC;
    wssize: CARDINAL; attr: ATTRIBUTES);
  PROCEDURE StartSimulation;
  PROCEDURE ACTIVATE(process: ELEMENT; kind: ACTKIND;
    time: LONGREAL; prio: CARDINAL);
  PROCEDURE DEACTIVATE(process: ELEMENT);
  PROCEDURE TERMINATE(process: ELEMENT);
  PROCEDURE EvTime(process: ELEMENT): LONGREAL;
  PROCEDURE Priority(process: ELEMENT): CARDINAL;
  PROCEDURE Status(process: ELEMENT): STATUS;
  PROCEDURE Attributes(process: ELEMENT): ATTRIBUTES;
  PROCEDURE SimTime(): LONGREAL;
  PROCEDURE Current(): ELEMENT;
  PROCEDURE Insert(pos: POSITION; Q, process: ELEMENT);
  PROCEDURE Remove(pos: POSITION; Q: ELEMENT): ELEMENT;
  PROCEDURE MakeQ(): ELEMENT;
END simulate.
```

EXAMPLE SIMULATION: A VIRTUAL MEMORY SYSTEM

A simulation of a simple paged virtual memory system provides a good vehicle for comparing simulation tools. It is complex enough to exercise most of the features of a language, while it is simple enough to be quickly understood. The model consists of several jobs (NumJobs) arriving according to a Poisson process (with parameter InterArrTime) at the system. Each job performs a fixed number (NumRecords) of record-oriented access requests. The access can be either a read or a write operation on a record which is contained in some page in the virtual address space. In carrying out the access, the system first determines whether the page is in main (real) memory by examining the *page table*. If the page is in main memory, then the record is accessed by going direct to main memory. We are assuming that records consist of approximately 100 bytes and that the memory bandwidth is 10 Megabytes per second. Hence, in the model for simplicity each record access requires 0.01 milliseconds (ms).

On the other hand, if the page is not in main memory, we must first retrieve it from the paging disk. In order to do this we must find a usable page slot or frame in main memory in which to load the page. Optimally we should choose the slot containing a page that will not be accessed for the longest time. It is hard to know the future, so we attempt to predict it from the past. A *page replacement* strategy that seems to work well is the LRU (Least Recently Used) strategy, where the page whose last access was the longest time ago is the one selected to be replaced. This strategy works well with jobs whose page access pattern follows the principle of locality where jobs tend to slowly drift from there current position (address). However, for more structured access patterns such as relational database joins, other strategies such as FIFO (First-In First-Out) may work better. Our simulations at this point compare only LRU with FIFO by setting a flag (LRU=1 for LRU and LRU=0 for FIFO). In this simple demonstration simulation, we generate a new page number that is uniformly distributed about the current page. If the parameter PageLocality is set to 1.0, the next page can be anywhere in the virtual address space, thus modeling uniform access. For this paper PageLocality was set to a small fraction to model locality based access.

Once we have selected a page to be replaced, we must check to see if it has been modified (dirty bit is on). If it has, then we must write back the old page to the paging disk before we can load in the new page. Therefore, if the dirty bit is on, page replacement requires two disk accesses, otherwise only one is needed. In this simulation we did not worry about the details of disk scheduling. We simply handled the disk request first-come first-served. The disk access time is uniformly distributed between 10 and 30 milliseconds (ms). The 10 ms roughly corresponds to the sum of rotational latency and transfer time on a fast disk, while the 30 ms adds seek time.

The simulation described above is implemented both in Modula-2 and SIMSCRIPT II.5. The 253 line SIMSCRIPT II.5 and the 360 line Modula-2 programs are not all that different. Listings of distillations of the two programs are shown in the appendix. The SIMSCRIPT II.5 program is smaller since it is able to use powerful built-in features of the language. The Modula-2 program has the advantage of being more modular and object-oriented. The major objects in the system, the page table, the paging disk, and main memory are each implemented as abstract data types. Because of the length of the Modula-2 program, only the disk abstract data type is shown. Of course the Modula-2 program has about 1000 lines of code associated with it in the simulate and statistics modules that it imports. Both programs have the following five input parameters: NumJobs, NumRecords, LRU, InterArrTime, and Stream. The first four are described above while the last one specifies which random number stream is to be used, thus facilitating the use of the *method of independent replications* (each scenerio was run on 5 streams). The outputs of greatest importance generated by the programs are the total number of page faults, and the mean delay (total time in system) for a job.

Let us briefly examine the results of this demonstration simulation. The results displayed in Tables 1 and 2 show that generally LRU gives a bit better performance than FIFO. However, more importantly FIFO appears to be more susceptible to thrashing. Rather than experiencing the usual few second delay, the jobs that got caught in thrashing experience a much greater delay. If enough data is collected, the mean values determined from the two programs should be very close. Differences result from differences in scheduling and random number generation (our Modula-2 system uses the LCG 16807 X mod 2^{31}-1 while SIMSCRIPT II.5 uses the LCG 630360016 X mod 2^{31}-1).

CONCLUSIONS

SIMSCRIPT II.5 and SLAM II are very powerful simulation tools. We have examined their process-oriented facilities and compared them with a modest Modula-2 simulation system. We have ignored their other powerful facilities (event-scheduling, continuous simulations, and graphics). Where then, among these well established giants might such a Modula-2 simulation system fit in. We envision five possible situations:

1. Teaching - A student will not need to learn a significantly different language, and can see the mechanisms driving the simulation by examining the simulate implementation module. Additionally, Modula-2 could be used to prepare students for a more advanced course using a general purpose simulation language.

2. Pascal Programmers - Since Modula-2 and Pascal are very similar it takes little time for a Pascal programmer to learn Modula-2.

3. Small-Scale Operation - A small scale operation (say one that needs to do simulation occasionally) may not want to incur the expense of a powerful general purpose simulation language.

4. Execution Efficiency - If execution efficiency is a critical concern then Modula-2 is a good choice.

5. Prototyping/Validation - A professor could write a less detailed SIMSCRIPT II.5 simulation model of a student's project to help validate the student's more detailed Modula-2 simulation model [Weyr 88].

In the future we plan to complete the implementation of the virtual memory simulation in SLAM II and analyze its efficiency. [We must first acquire SLAM II on our Sun machines (currently it is only available on the campus IBM mainframe)]. The simple demonstration virtual memory simulation model is currently being embellished. We are adding more detail to the model, considering other replacement strategies, and analyzing many more scenerios. We are considering other page access pattern models and other page replacement strategies. We plan to incorporate more sophisticated disk scheduling algorithms such as SCAN. We also plan to apply complexity metrics (e.g., Halstead's Software Science) to generate quantitative comparisons of the languages. Experience with our Modula-2 simulation system from using it for our simulation class, will allow us to refine it. Finally, we will consider adding graphics capabilities to our Modula-2 simulation system.

REFERENCES

[Bank 84] J. Banks and J. Carson, **Discrete-Event System Simulation**, Prentice-Hall, Inc., Englewood Cliffs, N.J. (1984).

[Brun 84] G. Bruno, *Using Ada for Discrete Event Simulation*, Software-Practice and Experience 14, 7 (1984).

[CACI 87] CACI, **SIMSCRIPT II.5 Programming Language**, CACI, Inc., Los Angeles, Cal. (1987).

[CACI 87] CACI, **UNIX SIMSCRIPT II.5 User's Manual**, CACI, Inc., Los Angeles, Cal. (1987).

[Grif 85] N. Griffeth and J. Miller, *Performance Modeling of Database Recovery Protocols*, Transactions on Software Engineering 11, 6 (1985).

[Hugh 87] D. Hughes, *Modula-2 Co-routines and Simulation*, J. of Pascal, Ada, & Modula-2 6, 4 (1987).

[Law 82] A. Law and W. Kelton, **Simulation Modeling and Analysis**, McGraw-Hill, N.Y. (1982).

[Prit 86] A. Pritsker, **Introduction to Simulation and SLAM II**, 3rd Ed., John Wiley & Sons, N.Y. (1986).

[Russ 83] E. Russell, **Building Simulation Models with SIMSCRIPT II.5**, CACI, Inc., Los Angeles, Cal. (1983).

[Sun 86] Sun, **Using Modula-2 on the Sun Workstation**, Sun Microsystems (1986).

[Weyr 88] O. Weyrich, D. Brown, and J. Miller, *The Use of Simulation and Prototypes in Software Testing*, Proceedings of the 1988 Eastern Simulation Conferences: Tools for the Simulationist, Orlando, Florida (1988).

[Wirt 85] N. Wirth, **Programming in Modula-2**, 3rd Ed., Springer-Verlag, N.Y. (1985).

Table 1. Modula-2 Simulation Results

a. NumJobs = 100, NumRecords = 1000, InterArrTime = 4000.0

	LRU		FIFO	
n	Faults	Delay	Faults	Delay
1	4274	2862	4974	4516
2	4432	2792	4327	2261
3	4244	2510	5058	4464
4	4345	2789	4822	3282
5	4592	2727	4622	2805
avg	4377	2736	4761	3466

b. NumJobs = 100, NumRecords = 1000, InterArrTime = 3300.0

	LRU		FIFO	
n	Faults	Delay	Faults	Delay
1	4766	4062	4594	3816
2	4519	3079	5116	4001
3	4029	3137	5602	9669
4	5510	10097	5126	5453
5	4321	2815	4739	3479
avg	4629	4638	5035	5284

Timing Results for b in Seconds for User Code and System Code

n	User	System	User	System
1	94.5	0.6	93.1	0.5
2	93.4	0.6	94.8	0.6
3	92.4	0.6	97.1	0.7
4	96.7	0.7	95.3	0.6
5	93.3	0.6	93.9	0.6
avg	94.1	0.6	94.8	0.6

Table 2. SIMSCRIPT II.5 Simulation Results

a. NumJobs = 100, NumRecords = 1000, InterArrTime = 4000.0

	LRU		FIFO	
n	Faults	Delay	Faults	Delay
1	4771	3328	4669	3593
2	4405	2855	4770	3223
3	4116	2607	4186	2338
4	4902	4986	4906	4181
5	4647	3058	4746	2922
avg	4568	3367	4655	3251

b. NumJobs = 100, NumRecords = 1000, InterArrTime = 3300.0

	LRU		FIFO	
n	Faults	Delay	Faults	Delay
1	4874	3628	4701	4196
2	6097	15550	7286	20368
3	4789	3452	5206	6265
4	4949	5111	4701	3726
5	4237	2757	5995	10824
avg	4989	6100	5578	9076

Timing Results for b in Seconds for User Code and System Code

n	User	System	User	System
1	397.2	1.9	391.8	3.1
2	430.9	1.5	423.6	1.8
3	392.9	2.6	399.2	1.8
4	396.1	1.9	392.7	2.7
5	386.7	1.6	409.0	2.3
avg	400.8	1.9	403.3	2.3

```
MODULE VirtualMemory;   (* Modula-2 *)
  FROM simulate IMPORT CREATE, ACTIVATE, DEACTIVATE, TERMINATE,
    StartSimulation, Current, Attributes, SimTime, ACTKIND, ELEMENT;
  FROM statistics IMPORT Uniform, Exponential, Collect, WriteStat;
  FROM attributes IMPORT ATTRIBUTES;
  FROM Storage IMPORT ALLOCATE, DEALLOCATE;
  FROM Disk IMPORT RequestDisk, AccessDisk, ReleaseDisk;
  FROM Memory IMPORT RequestMemory, AccessMemory, ReleaseMemory;
  FROM PageTabl IMPORT PageRange, SlotRange, FindPage, VictumPage,
    TouchPage, FlushPageTable, EnterPageTable, PageDirty, LRU;
  VAR
    Stream, NumJobs, NumRecords, NextJob, PageFaults: CARDINAL;
    PageLocality, InterArrTime, Delay, CreationTime, WriteProb: LONGREAL;
    process: ELEMENT;
    attr: ATTRIBUTES;

PROCEDURE Job;(* Each job (coroutine) reads/writes pages in virtual memory *)
VAR
  ArrTime: LONGREAL;
  atr: ATTRIBUTES; i: CARDINAL;
  CurPage: PageRange; Slot: SlotRange;
BEGIN
  (* Job Arrival *)
  atr := Attributes(Current());      (* Current job's attributes *)
  ArrTime := SimTime();
  CurPage := TRUNC(Uniform(1.0, FLOAT(MAX(PageRange)), Stream));

  FOR i := 1 TO NumRecords DO
    atr^.RecNum :=i;
    atr^.IsWrite := Uniform(0.0, 1.0, Stream) <= WriteProb;
    Slot := FindPage(CurPage);  (* Examine the Page Table *)
    If Slot = 0 THEN            (* Page is not in memory *)
      INC(PageFaults)
      Slot := VictimPage();     (* Use page replacement *)
      IF PageDirty(Slot) THEN
        (************************************************************
        Write back the old page to the disk thus saving its contents.
        ************************************************************)
        RequestDisk;            (* Wait for service from the disk *)
        FlushPageTable(Slot);   (* This slot is now unavailable *)
        AccessDisk(Stream)      (* Access the paging disk *)
        (* Dont release disk until finished with the page fault *)
      ELSE
        RequestDisk
      END (* IF *);
      (*************************************************************
      Load the current page into the replacement slot in memory from disk.
      *************************************************************)
      FlushPageTable(Slot);     (* This slot is now unavailable *)
      AccessDisk(Stream);       (* Access the paging disk *)
      ReleaseDisk;             (* Release the disk for other jobs *)
      EnterPageTable(CurPage, Slot)   (* Enter this page into the Page Table *)
    END (* IF *);
    (*****************************************************************
    Perform a main memory Read or Write a record in CurPage.
    *****************************************************************)
    RequestMemory;
    AccessMemory;               (* Access main memory *)
    ReleaseMemory;
    TouchPage(Slot, atr^.IsWrite);      (* Update the Page Table Entry for Slot *)
    CurPage := GenPageReference(CurPage, PageLocality)
    (* Generate new current page for next time around loop. *)
  END (* FOR *);

  (* Job Departure *)
  Delay := SimTime() - ArrTime;
  Collect(1, Delay);           (* Collect stats on job delays *)
  WriteDelay(* "Job", atr^.JobNum, "arrives", ArrTime, "departs", SimTime(), "with delay", Delay *);
  TERMINATE(Current())
END Job;

BEGIN (* VirtualMemory - initialize, create coroutines and start simulation *)
  ReadInputs(* NumJobs, NumRecords, LRU, InterArrTime, Stream *);
  WriteProb  := 0.5;
  PageFaults := 0;
  PageLocaility := 0.005;
  CreationTime := 0.0;
  FOR NextJob := 1 TO NumJobs DO
    NEW (attr);  attr^.JobNum := NextJob;
    CREATE (process, Job, 1000, attr);
    CreationTime := CreationTime + Exponential(InterArrTime, Stream);
```

```
    ACTIVATE (process, AT, CreationTime, 0)
  END (* FOR *);
  StartSimulation;
  WriteResults
END VirtualMemory.
```

```
PREAMBLE ''VirtualMemory   (SIMSCRIPT II.5)
   ''Define constants and symbols
   DEFINE TRUE  TO MEAN 1
   DEFINE FALSE TO MEAN 0
   DEFINE # TO Mean <>
   ''Define objects
   PROCESSES INCLUDE Generator
      EVERY Job HAS A JobNum, A RecNum, AN IsWrite''Job Attributes
      DEFINE JobNum, RecNum, IsWrite AS INTEGER VARIABLES
   RESOURCES INCLUDE Disk, Memory
   PERMANENT ENTITIES
      EVERY PageRec HAS A Page, A LastAccess, A Dirty''PageRec Attributes
      DEFINE Page, Dirty AS INTEGER VARIABLES
      DEFINE LastAccess AS DOUBLE VARIABLES
   ''Define variables
   DEFINE LRU, Stream, NumPages, NumJobs, NumRecords, NextJob, PageFaults
      AS INTEGER VARIABLES
   DEFINE PageLocality, InterArrTime, DiskSerTMin, DiskSerTMax, MemorySerTime,
      Delay, WriteProb AS DOUBLE VARIABLES
   ''Define functions
   DEFINE FindPage, VictumPage, GenPageReference AS INTEGER FUNCTIONS
   ''Define statistical variables
   TALLY MinDelay AS MIN, MeanDelay AS MEAN, MaxDelay AS MAX OF Delay
   ACCUMULATE MinDiskQ AS MIN, MeanDiskQ AS MEAN, MaxDiskQ AS MAX OF N.Q.Disk
   ACCUMULATE MinMemQ AS MIN, MeanMemQ AS MEAN, MaxMemQ AS MAX OF N.Q.Memory
END ''Preamble

Main    ''Main procedure - initialize and start simulation

PROCESS Generator    ''Generate job arrivals to the system

FUNCTION FindPage(CurPage)    ''Find the current page in the Page Table

FUNCTION VictumPage    ''Use the page replacement startegy to find a victum page

FUNCTION GenPageReference(CurPage, PageLocality)    ''Generate a new page reference

PROCESS Job    ''Each job reads/writes pages in virtual memory
   DEFINE ArrTime AS DOUBLE VARIABLES
   DEFINE i, CurPage, Slot AS INTEGER VARIABLES
   ''Job Arrival
   ArrTime = Time.V
   JobNum.. = NextJob
   NextJob = NextJob + 1
   CurPage = Uniform.F(1, NumPages, Stream)

   FOR i = 1 TO NumRecords DO
      RecNum.. = i
      IF Uniform.F(0, 1, Stream) < WriteProb  IsWrite.. = TRUE ELSE IsWrite.. = FALSE ENDIF
      Slot = FindPage(CurPage)      ''Examine the Page Table
      IF Slot = 0                   ''Page is not in memory
         PageFaults = PageFaults + 1
         Slot = VictumPage          ''Use page replacement
         IF Dirty(Slot) = TRUE
            ''************************************************************
            ''Write back the old page to the disk thus saving its contents.
            ''************************************************************
            REQUEST 1 Disk(1)       ''Wait for service from the disk
            Page(Slot) = 0          ''This slot is now unavailable
            WORK Uniform.F(DiskSerTMin, DiskSerTMax, Stream) UNITS ''Access the disk
            ''Dont release disk until finished with the page fault.
         ELSE
            REQUEST 1 Disk(1)
         ENDIF
         ''****************************************************************
         ''Load the current page into replacement slot in memory from disk.
         ''****************************************************************
         Page(Slot) = 0             ''This slot is now unavailable
         WORK Uniform.F(DiskSerTMin, DiskSerTMax, Stream) UNITS''Access the disk
         RELINQUISH 1 Disk(1)    ''Release the disk for other jobs
         Page(Slot) = CurPage       ''Enter this page into the Page Table
         Dirty(Slot) = FALSE
      ENDIF
      ''*****************************************************************
      ''Perform a main memory Read or Write of a record in CurPage.
      ''*****************************************************************
      REQUEST 1 Memory(1)
      WORK MemorySerTime UNITS     ''Access main memory
      RELINQUISH 1 Memory(1)
      ''Update the Page Table entry for Slot (reset LastAccess for LRU only)
      IF LRU = 1 LastAccess(Slot) = Time.V ENDIF
```

```
    Dirty(Slot) = Max.F(Dirty(Slot), IsWrite..)
    CurPage = GenPageReference(CurPage, PageLocality)''Generate a new page reference
  LOOP

  ''Job Departure
  Delay = Time.V - ArrTime
  PRINT 1 LINE WITH JobNum.., ArrTime, Time.V, Delay THUS
  Job *** arrives *****.** departs *****.** with delay *****.**
END Job
```