ON THE RELATIVE MERITS OF TWO MAJOR METHODOLOGIES FOR SIMULATION MODEL CONSTRUCTION

Charles M. Shub

ABSTRACT

Given a model of a discrete change system there are two major methodologies for developing a simulation program of the model. They are (A) describing the flow of typical units of traffic through the model, and (B) describing what occurs at those instants of time when the state of the system changes. This paper compares, without entering into the controversy of which language is best for modeling, the relative advantages and disadvantages of the two methodologies. The comparison is made at the modeling or source language level of detail and does not specifically adjudicate the relative merits of language package internals such as dynamic storage implementation, the notion of an events chain, be it current or future, or the notion of a timing routine except as necessary for comparisons. The paper is concerned with the relative advantages of the two methodologies in handling such problems as simultaneous occurrences, synchronization of traffic units, communication between traffic units, complex decision algorithms involving waiting line manipulation, the general notion of "style" in the code produced, the ease of gathering meaningful statistics, the level of knowledge required of the system internal structure, and the refinement of level of detail.

INTRODUCTION

The analyst faced with implementing a discrete change system can select from among a number of programming languages and simulation packages and languages for his implementation. Though the languages and packages vary widely in detail, the basic choice of methodologies boils down to a choice between two alternatives; namely describing the flow of typical units of traffic through a system or describing what happens at those instants of time when a change in the state of the system occurs.

As a simple example, consider a barbershop with a single barber who only cuts hair, and a waiting area with an infinite number of chairs. In any implementation, one will need some information concerning the distribution of interarrival times of clients and the distribution of haircut time for the barber. These details, while interesting in their own right, are not within the scope of this paper. One approach to simulation of this system would involve describing the action of a customer as follows: The customer arrives and waits his turn and uses the barber and leaves. This methodology will be called the "FLOW" method. The other approach would be to describe the actions at the times the system changes state. Such a description would include the following: (A) when a customer arrives, depending upon whether the barber is busy or not, he either sits down to wait or starts getting his hair cut; (B) when the barber finishes cutting a customer's hair, the customer leaves, and then, depending upon whether there are any more customers, the barber either starts cutting the hair of the next customer or sits down to read a book. This methodology will be called the "STATE" method.

In a model such as this, typical measures of interest would include how many people are waiting, how long people wait, how long people are in the barbershop, and what fraction of the time the barber is busy cutting hair. In either implementation methodology, such statistics are routinely available, though some knowledge of the package internals may be necessary to obtain these statistics in an easy manner.

This paper explores the differences in coding a model using each of the above methodologies in the hope that a programmer can use the information presented to make a better informed choice as to how to solve his problem. Following a brief discussion of the question of why a language, and thus a methodology, should be chosen, a section on implementation details provides a short description of enough implications of the internal details to make the remainder of the discussion meaningful. The simple system described above is then analyzed to set the discussion tone. The difficulties of simultaneous occurrence are then considered. Next, the general problem of data structure manipulation is considered. Synchronization, communication, and refinement of level of detail are covered in an integrated fashion. A brief comment is made on the style and readability of the code. Finally, the notion of statistics gathering is described and conclusions are drawn. Several excellent works [1, 3, 5, 10] not cited elsewhere in the paper have provided much valuable information regarding differences in methodologies.

FACTORS INFLUENCING CHOICE OF METHODOLOGY

All too often, when asked why he chose a particular language or methodology, a modeler gives one of the following replies:

A) That is the language or methodology I know or feel comfortable with;
B) That is the language or methodology my manager or supervisor directed me to use; or
C) That language or methodology was readily available on our system.

Since, with one possible exception, choice of language dictates choice of model description methodology, the above reasons should not be used as a major design criterion for the model. The literature suggests several factors for making such a choice. They include:

A) Suitability of the language, not only for the solution to the problem, but also the future users of the program;
B) Ease of implementation of the model in the chosen language;
C) Ease of explanation of what the program and model do, not only to managers and supervisors, but also to professional colleagues;
D) Evaluation of previous use of the language;
E) Cost effectiveness of the choice of language;
F) Technical and functional characteristics of the language;
G) Portability requirements of the model and related portability of the program;
H) The ability to use partial results of others; and
I) Flexibility, compatibility, and expandability considerations.

If, for example, the problem is much easier to implement in one methodology, or a similar model is available and can be easily adapted, the most effective technique for solving the problem may involve either learning enough about a previously unknown language in order to develop changes within the scope of that language to an already existent model, or to purchase outside computer time to solve the problem. Knowledge of alternative approaches and their relative merits can be a particularly powerful tool, not only to ease the burdens of doing ones job, but also to enable the modeler to make persuasive arguments to management regarding justification of his decisions and desires relative to the task guidelines.

Jean Sammet [4] not only provides an excellent in-depth discussion of the major factors influencing language selection, but also gives several other sources for further information on the topic. Moreover, she also provides a few details concerning over ten different simulation languages. More recently, Robert Shannon [6] has provided a general flow chart for selecting not only a type of simulation (analog, hybrid, or digital), but also some generalizations which can often be quite useful as an aid to language selection. He also provides a number of references for various languages.

IMPLEMENTATION DETAILS

While it is not the purpose of this report to provide a detailed comparison of package internal details, some knowledge of the internals, and inherent differences in implementation methodology, is necessary to make comparisons. The following notational conventions will be used throughout this report.

Definitions:

SYSTEM: That system which the program is trying to model.

MODEL: The computer program.

CUSTOMER: A unit of traffic in the system. Always represented by a block of memory cells. Typical names used in packages for a customer include transaction, process, entity (usually temporary), etc.

SERVICE: A person or thing in the system which is normally used or utilized by one or more customers. Normally, a service is represented in memory by several words which usually contain status-type information. Typical names used in packages for a service include resource, entity (usually permanent), facility storage, etc.

LINE: A mechanism for keeping track of customers, all of which have something in common. Typical names used in packages include sets, queues, chains, etc. There are two distinct implementations used for lines.

Ordered Line
This implementation involves some physical sort of linked list arrangement as described so well by Knuth [2]. With this arrangement, the software can traverse the list, insert and delete customers at will, and perform any of a variety of list manipulation actions.

Unordered Line
The line is treated similar to a service in the respect that there is a control block containing status information concerning the line, and, in addition, each customer who is in the line has an "in line" status bit in his own memory set to indicate he is in the line. No ordering is directly possible with the unordered line implementation.

SOFTWARE: The code or instructions which implement the algorithms to perform the various actions which the language supports and the language statements imply. For example, when processing a QUEUE instruction, the software takes the necessary steps to show that the customer to whom the QUEUE action was applied is shown as being in the appropriate line. The manner in which this is done is, of course, an implementation detail which need not concern us.

TIMER: That portion of the software which decides which software action is to occur next. The timer may embody several ordered lines. Typical names used for the timer include events chain, timing routine, controller, executive, etc.

Implementing the Flow Technique:
There are two major implementations of the flow technique. One involves essentially a translation by the compiler to the state technique, and the internal operations are quite similar to that of the state technique internals using ordered lines of customers waiting for a service and an ordered line of customers actually using a service. The other involves the more classical use of flow timing algorithms as described below.

Time Details:
The first major point to notice is that no matter what descriptive methodology is used, the actual implementation must involve a timer section to coordinate actions so that they occur in proper sequence. If one uses a "STATE" model methodology, the transformation from program to timer inputs is fairly straightforward and rather simple. Thus, the "STATE" model implies that the translation task is somewhat easier and intuitive. Furthermore, the description maps very nicely in an almost one-to-one fashion onto the implementation. This can lead to some simplifications when error correction is needed, but can also lead to complications and is thus, perhaps, a mixed blessing. In fact, normally, what the timer must do is to select the first event descriptor from the timing line and "call" it as a subroutine.

The interaction of the "FLOW" description method is somewhat more subtle but is, as far as the timer is concerned, essentially similar. The timer must have at its disposal all active customers, and must further have them in an ordered line which is ordered first by time of occurrence of the next incremental step in the flow of the customer through the system, and then by some other as yet unspecified criterion. Typical implementations usually involve the use of two lines, the first being those customers who cannot progress in their flow at the current time but can progress at a later time, and the second being those customers who are willing to continue their flow immediately. As an example of the first situation, consider a customer utilizing a service. The customer will continue to utilize the service until he is done and, thus, cannot continue his flow through the system until his use of the service is complete. Consider, as an example of the second type, a customer who wants to use a service. The customer is ready to progress through the system as soon as he can but circumstances, such as the service being busy or occupied, may prevent that. Perhaps the best way to differentiate between the two classes of timer lines is to classify the customers on one line as customers who do not wish currently to flow through the system (a passive timer line), and those on the other as customers who do wish to currently continue to flow through the system but who may not be able to (an active timer line).

The actions of the timer in this situation are somewhat more complex. For each customer in the active line, the timer must use the software to effect the flow of the customer through the system until the customer can no longer flow or the software has moved that customer into the passive timer line. When no customer on the active timer line can flow, the simulation clock is updated to the time when the first customer on the passive timer line becomes active and that customer is moved to the end of the active timer line. Of course, if more than one customer is due to become active at that same time, all of them are moved to the active timer line. Then the scan of the active timer line is restarted.

Order In An Unordered Line:
Though an unordered line can have, by definition, no direct ordering, an indirect ordering can be implied on an unordered line. If every customer in an unordered line is also in an ordered line, the order of the ordered line is then implied on the unordered line. Note that in the "FLOW" description method, all active customers are in ordered timer lines so that the implied ordering is also transmitted to all unordered lines.

ANALYSIS OF THE SIMPLE SYSTEM

Clearly, in the simple barbershop system any model will have customers. There is also a service, namely the barber. Some sort of waiting line is implicit in the system. In a state model, the line must be explicitly provided in order to keep track of the customers. However, with the flow model, the the line can be implicit within the timer and need not be explicitly provided. Therefore, a cursory glance indicates that the flow model is actually much simpler than the state model. Proponents of any given language can easily generate these simple models in a few moments. Most programmers using flow models will, almost as a matter of course, add a line explicitly to their model. If this is implemented as an unordered line, it is usually to take advantage of the built in line status statistics. If an ordered line implementation is used, the reason usually is to offset the effects of timer slowness and the resultant execution inefficiency due to a buildup of customers on the active timer line.

In the final analysis, either model will include customers and a service and at least one ordered line. Some ordered lines will be hidden within the timer software and, therefore, not readily apparent to the programmer. However, based upon shortness of program and simplicity of the model, the flow model appears to be the choice for this system. Despite this, one must remember that in terms of actual execution effort at the computer instruction level, the actions are both extremely similar, and the apparent superiority of the flow model derives from its software doing several things which the state model programmer must do.

As an example of the difference, consider the service. In either model, at any given time, the service will be either idle or busy. In the case of the flow model, the software effects the status changes, while in the state model, the programmer

must explicitly change the status. Within the state model, the timer is extremely efficient because it has at most two alternatives to cope with. The next state change is either due to an arrival of a customer or the departure of a customer. In the flow model, the timer (assuming an unordered line) must manage each customer in the barbershop and must, therefore, cope with an unbound line. Within the scope of the flow model, one does have the alternative of explicitly using an ordered line. If this choice is made, then the flow model is no longer that much simpler than the state model because the line manipulation operations are specified by the programmer in both models.

SIMULTANEOUS OCCURRENCES

A problem which can occur in many models is the problem of simultaneous occurrence. In terms of the state model, when two perhaps unrelated state changes occur at the same time. In terms of a flow model, this happens when two customers become able to flow at the same time. The resultant system behavior may depend quite significantly upon which of the two actions is processed first.

Consider the simple example used previously. Modify the system so that there is only a finite number of (say 7) chairs in the barbershop, and the customers all follow the rule that if no chair is available when they arrive, then they leave without waiting for service. Suppose that a customer arrives at exactly that instant of time that the barber is finishing. Suppose further that all (7) chairs are full. Does the customer stay or leave?

In the state model, the answer depends upon whether the arrival algorithm is processed before or after the departure algorithm. In the flow model, the answer depends upon whether the arriving customer flows before or after the customer being served. In both cases, the programmer can specifically control which action occurs first. However, the mechanisms are very different and not always straightforward in all models.

Recall that in the flow model, the timer has an ordered line. This line can be, and in fact is, further ordered by assigning a dynamic priority to each customer who can be on the active timer line. The active timer line is then ordered first by priority class and second by the placement algorithm within each class. Thus, the modeler can assure that the customer whose hair is being cut will flow before a customer newly arriving by ensuring that customers receiving service always have higher priority than newly arriving customers. Thus, the timer will attempt to make the customer receiving service depart before attempting to make the new arrival flow. Once the departed customer has flowed through the system, a blocked customer will be assigned to the barber thereby providing a place for the new arrival to sit. Care must be taken to assure that all three customers are in the proper order in the timer line. Specific details of what priority ordering is necessary will be dependent upon what decision algorithm the timer uses to decide when to restart scanning the active timer line from the front.

Recall that with a state model the timer must select the next state change algorithm and then "call" the appropriate algorithm. Normally, the ordering is based upon the time of occurrence of prospective state changes. The modeler then can control simultaneity by instructing the timer that one algorithm should take priority over the other if they are both due to happen at the same time.

Whatever the methodology, the technique of coping with the problem is to force the desired ordering upon the timer by assigning priorities. However, the two methodologies require that this be done in drastically different fashions. This difference can, and usually does, have a profound effect on the difficulty of the task. The state method controls statically the priority of algorithms, while the flow method controls dynamically the priority of customers.

In a state model, all state change algorithms are explicitly written, so the programmer has a list of all possible state change algorithms. It is a simple task for him to assign the relative priorities of the completely known possible set of conflicts and thereby assign the ordering to be followed in case of time ties. Moreover, this specification is usually done one time on a static basis and can be implemented as part of the language or package translation process. All that the modeler must do is give the order for algorithm performance. The important point is to note that several algorithms with widely different priorities can process the same customer. This implies that even though the algorithms have a statically assigned priority, the customer will have a derived priority that is dynamically changing.

In the flow methodology, controlling the priority of a customer is usually a fairly simple task. The problem lies in how the programmer decides to set priorities. In the example given, the programmer must first recognize that there may be a simultaneity problem between a customer arriving and a customer departing, and then must explicitly assign priorities. The problem, in this case, is compounded by the existence of a third customer waiting in line, and the priority assignments must be such that the scan algorithm in the timer has a customer depart first, then a customer go from the line to the barber chair, and finally place the arriving customer into line. In more complex models, the interactions can be extremely subtle and complex, and may easily be overlooked by the programmer. Moreover, the use of the customer priority may well be difficult to implement. For example, in a medical center some services may be provided on one priority ordering, such as a lab technician collecting samples on a room order basis, while other services might be based on some other factor, such as degree of illness. The consumer (patient) would then have to be waiting in two different lines with different priorities for each.

In summary then, the handling of simultaneous occurrences in all but the simplest systems appears to be much simpler in state models using a static global algorithm priority scheme. The flow model analysis is more complex, not only because of the subtleties of the interactions which may be obscured by the model methodology, but also because of the requirement of understanding so well the timer role in the problem and the inherent complexities of dynamically readjusting the priority of each consumer at every necessary point in his flow through the system.

ON SEARCHING LINES

One touch in models which is often necessary and useful is the capability to base an action on whether or not certain conditions hold and, furthermore, organizing lines in manners other than first in first out. As an example, consider a system where there are various classes of consumers, say patients on various floors of a hospital. The service, say a lab technician collecting samples, desires as a matter of convenience to collect all samples from a single floor before proceeding to the next floor. A priority arrangement with priority level corresponding to floor level will just not work. Moreover, if the hospital has a large number of floors, the technique of using separate lines for each floor may become unwieldy as well. Suppose that the initial implementation uses a line ordered by floor with the high floors having high priority. If there are always several requests for service, it is possible that the lower floor (priority) requests will never be serviced. Also, if the service has handled all requests on the top two floors and a request for top floor service arrives while the service is serving the fifth floor down, the priority scheme will force him to make a trip to the top floor before finishing the floor he is currently serving. Basically, the line handling can be described by the following sequence of steps:

A) If there is a customer in the same class in the line already, insert the new customer in line at the end of the class.
B) Since there is no customer in the line in the same class, the decision as to placement at the front or back of the line is made based upon whether the service is currently serving a member of the class.

Selecting on this basis provides that the service will serve all customers on the same floor before leaving that floor, and will then choose which class of customers (floor) to process next based upon the earliest arrival of a request for service from a particular class (floor).

Solving this problem is difficult, if not impossible, with only unordered lines and a timer line. If, however, an ordered line is utilized, there are two possible implementations. The first involves placing the customer at the appropriate point in the line when the customer is placed in line, and the second involves removing a customer with the proper attribute values when deciding whom to remove from the line. The ease or difficulty of these tasks is related to the specific line manipulation primitives available much more than it is to the programming methodology. If, for example, one can search a line to determine if a customer with a specific attribute value is in the line (or can set a common variable to indicate presence) and can then insert a new customer in line after a specified customer, the solution is straightforward. Another alternative is to attempt to remove a customer with a particular attribute value from the line (no matter where in the line the customer happens to be placed) and to condition the next step on success or failure of the attempt. Several other options exist as well. The major requirement for any option is the ability (within the language or system) to do a bit more than order a line based only on the value of some particular attribute and remove from either end.

The primary difference between the two methodologies is that, in the state model, the algorithm is, to a large extent, divorced from the line and customers. The algorithm is separate and, thus, can do such things as reorder the line or reassign values of customer attributes while the customer is in the line. Conversely, in a flow model, one of the customers is performing the actions of attempting to manipulate other customers, and this can be much more difficult.

In certain problems, an unordered line solution is possible, even though a cursory glance might indicate otherwise. Consider the barbershop example. Suppose that if the line is over a certain length, an arriving customer will remain only if there is somebody he knows already in the barbershop. (Presumably the arriving customer is willing to wait longer if he has somebody to talk to.) Assume that the algorithm to determine whether or not the arriving customer knows a waiting customer is fairly complex. A solution with full line manipulation abilities would be to check, for each member of the line, whether he is known by the new arrival. With an unordered line, the information which is required would have to be preset in some common storage area in some fashion to allow for checking. While complex, this problem can be solved in this fashion in a flow model.

Finally, the familiar line hopping problem (the other line is moving faster, so I want to get out of this line and into that line), while not simple in any methodology, appears to be much easier to handle with an event model.

In summary then, the problem resolves to one of having the appropriate tools to handle the rather complex line manipulation problems. Usually, the event models come out well ahead on this score. This is because they normally allow for any variety of manipulation on any member of any linked list structure, as well as the opportunity to actually store pointers to particular customers. Moreover, a state model can normally allow any of the state change algorithms to change easily any attribute values of any customers whether or not they are in a line. Flow models, on the other hand, generally restrict operations to inserting and removing. Moreover, decisions can only be made at insert time or remove time. While on a line, a customer is generally rather inaccessible except to be actually removed.

SYNCHRONIZATION AND COMMUNICATION

The problems related to synchronization and communication are well-known. Several excellent treatments of the topics exist in the literature. Almost any book on Operating Systems devotes considerable effort to these topics. Shaw [7] provides an especially clear exposition. He devotes the major portion of his exposition to semaphore techniques, and the "P" (decrement if possible) and "V" (increment) indivisible semaphore operations. He gives examples of the use of semaphores as resource counters and synchronizers. In his discussion of implementation, he mentions many analogous names for similar primitives and provides the notion of "Blocking" progress to avoid a customer continually attempting to decrement a zero semaphore value. The important point, relative to simulation, is that the whole notion of a semaphore and the need for indivisible primitives is a result of problems which can occur when two things are going on in parallel. Given the sequential nature of both simulation methodologies being considered, the only necessary ingredient is to ensure that we implement semaphore primitives with indivisible chunks of software. This is, of course, trivial in an event model algorithm. It is also almost trivial in a flow model. The only point where care must be taken is the assurance that the customer can always completely flow through the steps of the semaphore operation before the timer attempts to cause another customer to flow. A little care can assure that this is always the case.

Most of the difficulties normally associated with critical sections and synchronization are not problems in simulation models, and the sometimes rather cumbersome techniques which must be used in operating systems can be avoided in simulation models by implementing whole sections of actions in which care must be taken in an operating system as an indivisible chunk of software.

In terms of communication and cooperation, as opposed to mutual exclusion, a number of problems arise which are of interest to the simulation modeler. While the modeler can resort to the techniques of operating systems to handle the problems, there often are simulation techniques which are not nearly so cumbersome. Consider, for example, a baton in a relay race changing hands. In an event model, there could probably be an algorithm to model removing the baton from the first customer (racer) and giving it to the second which would occur at the time the first racer got to the end of his leg of the race. Note that this algorithm has global control and, though triggered by the actions of the first racer, has the ability to communicate the information to the second racer who can be quiescent in the model until the algorithm acts. In a flow model, however, the communication is not quite so simple. The second racer (customer) must somehow flow to his starting point and must wait there until he receives a message (baton) from the first racer. Normally, this is implemented by using an auxiliary global variable to indicate presence or absence of the baton at the change point, and to cause the second customer to be unable to proceed until the baton presence indicator is set. A somewhat more complex example of handling synchronization and communication will be discussed below as a part of the discussion of the merits of the different methodologies with respect to refining a model to a finer level of detail.

In summary, the problems of synchronization and communication are usually not difficult to handle whatever the methodology used.

REFINING LEVEL OF DETAIL

One problem often encountered in modeling is that once a model is working, it becomes necessary to refine the level of detail. Consider as an example a model of a simple multiprogramming system with a fixed quantum. A flow model might describe the actions as follows: Wait in line for a quantum of processor time. After the quantum is complete, either go to the end of the line or depart depending on whether you are done. A state model might describe the two state change algorithms this way: A) On arrival, either get on line or start a quantum. B) At end of quantum, the finishing customer either gets put on line or departs. Then give the processor to the first customer on line or go idle if the line is empty. Coding either model is rather straightforward. The major implementation difficulties revolve around keeping track of how much time a customer has left and setting the short last quantum.

Suppose that one wishes to modify the model to account for the quantum expiration processing and arrival overhead processing. These modifications make the model more complex by an order of magnitude, assuming that one wants to detail the scheduling operations. In a state model, either of the state change algorithms becomes a sequence of state change algorithms performed at slightly separated times. This can lead to a veritable plethora of algorithms unless some technique to describe a finite nonzero algorithm occurrence time [8] is utilized. Neither method is extremely clean or elegant. In a flow model, it would appear that the change is relatively simple insofar as one can insert delays into the flow of a customer in a relatively straightforward fashion. However, synchronization and communication effects do add a complication that can be quite subtle in its form.

Consider the end of a quantum in the refined model. Assuming no complications (such as another customer arriving during the period of quantum end processing) arise, the state change can be nicely broken into four steps:

A) At T1, the processor is released by the finishing customer and, though unavailable for immediate use, is not doing productive work.
B) At T2, the finishing customer actually enters the line to wait for another turn.
C) At T3, the starting customer actually departs the line to begin his quantum.

D) At T4, the starting customer begins his productive use of the processor.

In a state model, then, the programmer must either describe four distinct state change algorithms with much information passed from algorithm to algorithm or must use a nonzero occurrence time technique. In a flow model, the use of an ordered line technique is impossible unless the programmer introduces a "ghost" customer to flow from time T2 to time T3 to remove the starting customer from the line at time T3. With an unordered line technique, one need not introduce a "ghost," but the problem does involve causing the finishing customer to continue to flow after being placed in line and then causing the starting customer to start flowing when he should flow out of the line. A gating technique can be used to synchronize the actions of the two customers. The finishing customer could go through a sequence as follows:

A) Open a gate for the starting customer.
B) Delay until a second gate is opened.
C) Flow through the second gate.
D) Shut the second gate behind himself.

The second or starting customer could then use the sequence below:

A) Delay until the gate is opened.
B) Flow through the gate.
C) Shut the gate behind himself.
D) Do the interim T2 to T3 delay processing.
E) Open the second gate for the finishing customer.

These sequences imply, of course, that the finishing customer would have to test to determine if the gate or synchronization protocol described above was necessary or not.

In this example, the state model is characterized by a proliferation of state change algorithms which is not desirable, not only because of software housekeeping complexity, but also because a logical unit, namely the sequence of changes, has been separated into parts and can be less clear in this fashion. The other alternative is to use a nonzero time technique, but this is not a clean solution either because a subalgorithm is used like a subroutine to do a minor piece of processing at a slightly later time. The flow model, while it does not have the above difficulties, does require actions that are not straightforward. In an ordered line implementation, a "ghost" customer is required because once the finishing customer goes into the ordered line, he can no longer do things until he comes out of the line, and, thus, the "ghost" must flow from time T2 to T3 and then remove the starting customer. Thus, the "ghost" handles the synchronization. In an unordered line implementation, a gating technique can be used to do the synchronization. However, much care must be taken to assure proper ordering in the timer line, especially when an unrelated customer might be able to start to flow at an inopportune time.

In summary, neither method is perfect, and the advantages of each method lie in complementing areas. The logic of the cooperation can be easily delineated within the guise of a state model, but the incremental state changes cannot be handled simply. With a flow model, the incremental state changes are easily delineated, but the logic of the cooperation can become extremely convoluted.

STYLE

In any large computer program, there should be concern regarding understandability, readability, and flexibility of the code. Wirth [9] is one of many who suggests a top-down or stepwise refinement method of programming as a vehicle for better understandability and readability. A state model lends itself nicely to these notions. At the highest level of description, the programmer defines the units and specifies the state changes which can occur. At the next level, each state change is detailed. The programmer has available the standard techniques of subroutines and functions to carry the refinement even further. A flow model, on the other hand, does not lend itself quite so cleanly to stepwise refinement in the usual sense of the term. However, to denigrate the flow languages on this basis would not only be unfair, but would also be inaccurate and a gross disservice. A strong case can be made that the flow model is also written with a refinement technique, though not in the traditional sense. At the top level is the specification of model segments or classes of customers. The refinement then involves detailing the flow as it occurs. The question, then, really is one of understandability of a technique rather than whether a specific technique maps cleanly onto the common notions of stepwise refinement or top-down programming. The question of understandability, divorced from the details of the technique, then boils down to what is explicitly delineated in the code versus what is hidden in the software.

In a flow model, the style is to set down in a rather precise and concise fashion what exactly a customer in the system does. This style will leave implicit and sometimes a bit to the imagination some of the interactions between different types of customers, and also interactions between two customers of the same type at different points in their flow. A state model, on the other hand, provides a very clear picture of the customer interactions but usually can obscure the flow of a specific customer, more so if the interactions are fairly complex. The trade-off in the style and readability area thus is related to which method of description is cleaner, more comfortable, and easier to present by not only the programmer, but also his supervisor and those who are using the model.

GATHERING STATISTICS

In any model, a number of relevant statistical measures are readily available as provided by the software. A problem can arise when statistical information which is not readily available is desired. Typically, any desired statistic can be gathered by the programmer. The concern is the ease or difficulty of the task and the accuracy of the information obtained. Keeping track of transition times for customers between various stages in their progress through a model can be easily

done. The value of the simulation clock is saved at the first point, and then at the second point a difference is calculated. This is usually fairly easy in either methodology. For the sake of simplicity, assume that what is desired is the statistical properties (as a function of time) of the value of a variable. In a flow model, a "ghost" customer can be introduced to periodically interrogate and save the values of the variable. In a state model, an identical technique can be used. A statistics gathering algorithm can periodically save the desired information. A state model also allows (as does a flow model, but not always in a straightforward manner) the insertion of code to gather the desired statistical information each time any state change algorithm changes the variable under consideration. This can become burdensome if several algorithms modify that variable. At least one state language provides a capability to specify on a global basis that the statistics be gathered every time the variable is modified, no matter which algorithm does the modification. This mechanism provides the cleanest solution to the statistics gathering problem for those statistics which are not routinely gathered by the software.

The trade-off then is the ghost technique, versus the brute force technique, versus the global specification. The ghost has a possibility for inaccuracy as it is a periodic sample of a random variate, and the mere periodicity may obscure a related periodic fluctuation in the random variable being observed. The brute force method can become quite involved and add a great deal to the complexity and length of the program. The global specification technique works nicely. One can think of it as a specification that a particular statistic be gathered routinely. The only problem is that in addition, the programmer may have to specifically assign a value to the variable so that the global specification works properly.

CONCLUSIONS

This paper has presented analyses of the differences between two major methodologies for modeling discrete change systems. This has been done in several contexts. The purpose has not been to advocate one technique over the other but rather to make a comparison available so that a modeler can make a better informed choice as to which technique better suits his (or her) purposes. It is hoped that an outgrowth of this short analysis will be a better awareness of both techniques on the part of those individuals who are now inexorably wedded to one over the other. One paper cannot hope to "bring the two camps together," but it can, and hopefully has, provided a basis or foundation for better understanding and furtherance of simulation as a discipline.

BIBLIOGRAPHY

1. C.A.C.I., "SIMSCRIPT II.5 Reference Handbook," C.A.C.I., Inc. - FEDERAL, 12011 San Vincente Blvd., Los Angeles, CA, March, 1976.
2. Knuth, Donald E., "The Art of Computer Programming," Volume 1, "Fundamental Algorithms," Addison-Wesley, Reading, MA, 1969.
3. Russell, E. C., "Simulating with Processes and Resources in SIMSCRIPT II.5," C.A.C.I., Inc. - FEDERAL, 12011 San Vincente Blvd., Los Angeles, CA, July, 1976.
4. Sammet, Jean E., "PROGRAMMING LANGUAGES: History and Fundamentals," Prentice-Hall, Englewood Cliffs, NJ, 1969.
5. Schreiber, Thomas J., "Simulation Using GPSS," John Wiley and Sons, NY, 1974.
6. Shannon, Robert E., "SYSTEMS SIMULATION: The Art and Science," Prentice-Hall, Englewood Cliffs, NJ, 1975.
7. Shaw, Alan C., "The Logical Design of Operating Systems," Prentice-Hall, Englewood Cliffs, NJ, 1974.
8. Shub, Charles M., "Modeling Events Which Take Finite Time to Occur," in Proceedings of the 1976 Summer Computer Simulation Conference, Washington, D.C., July, 1976.
9. Wirth, Niklaus, "Program Development by Stepwise Refinement," in Communications of the A.C.M., Volume 14, No. 4, April 1971, pp. 221-227.
10. Xerox Data Systems, "Xerox General Purpose Discrete Simulator (GPDS) Reference Manual," XDS, El Segundo, CA, Document Number 90 17 58A, April, 1971.