# KEYNOTE ADDRESS

This opportunity to talk to the 1979 Winter Simulation Conference gives me great pleasure on several grounds; the lack of contact between the two sides of the Atlantic on simulation matters is of great concern to the workers in Europe and this contact must do some good; the work of the Simulation Centre set up in Britain by the Science Research Council (responsible for funding research in British Universities) is not well known in the US and a description of it should be of interest; and last, but by no means least, the need for a sound methodology of our subject is becoming more pressing and aspects of this should be rehearsed.

The history of simulation studies is full of contradictions; in some respects there have been resounding successes, in others, complete failures. Even its origins are hard to pinpoint; some would argue that Buffon's experimental determination of $\pi$ by throwing needles on a grid is an early example of a simulation, others would claim that this admission would open the gates too wide and that on this criterion nearly every practical project would be a simulation. The first activity on any significant scale was probably the work on switching networks done by Erlang and his fellow Scandinavians at the turn of the century. In the late twenties and the thirties there was a vogue for determining sampling distributions by experiment and this no doubt contributed to the common fallacy that a simulation must have random components.

I suppose that the technique could not gain ascendancy until the advent of the computer and this was coupled with the start of industrial applications of OR. The first attempts at providing tools for simulation followed the style of all software of that time - it consisted of routines to advance time, take samples and so on. The real breakthrough came in 1958 with the publication, in Europe, of the description of the simulation language GSP - this tackled the problem of the structure of a simulation program. It was a breakthrough for more than simulation - it was the first problem oriented language. In the US, SIMSCRIPT arrived about the same time.

Since that time there have been dozens, if not hundreds, of simulation languages offered to simulators. They are all based on one of a few, very different methodologies and differ otherwise only in convenience and flexibility in use. They share one thing in common - they are not used to any great extent. In Europe a count at any gathering of simulators will show that the majority of simulations written do not use any simulation language; they are written in FORTRAN, BASIC or even by a lunatic fringe in

APL. The only bright feature is that no one seems to have tried to write a simulation in COBOL!

Why is this? The usual explanation is that to use a simulation language one has to learn a lot of detail and it does not seem worth it for the few occasions one will write a simulation. Instead, they burden themselves with solving the synchronisation problems of a simulation - get it wrong and induce a long development and debugging process.

Those who have ever designed a simulation language know that the apparent simplicity of the task is a delusion. The secure serialisation of parallel real world processes on a computer was, until the advent of time sharing operating systems, one of the most challenging problems in the art of computer programming. Indeed, the design of operating systems could, and some cases did, benefit from the experience of simulation language writers.

By extending the duration of simulation projects, these amateur system designers gave simulation a bad name. Stated bluntly, the fault lies in the laziness of would-be simulators who expect to acquire instant knowledge painlessly; they are not unique in this modern trend,which is always excused by asserting that if it is difficult for them to understand, it will be impossible for their clients (automatically classified as intellectually inferior !)

I do not know the remedy for this; I do know that the path being taken by some British workers is fraught with danger. This school propose to make it easier and easier to write a simulation so that less and less has to be learnt before the language can be used. While these simplifications merely remove the burden of detailed specifications from the user, this is good. Unfortunately, it also removes any need to understand at any level the structure of the program being generated; the resulting program is then not understood by anyone.

The ideal for these workers is a computer dialogue with the user which extracts all the facts about the system to be studied from replies,all with a numeric or alphabetic format. There is no syntax to learn. It remains an ideal; such a scheme can only accept descriptions of systems which conform to a certain limited structure - all real systems have at least one part which will not fit such a structure. The user is expected to patch the program in the high level language to accommodate the reality, he has been

relieved of the necessity of learning the easy parts of the language; all he needs to know are the difficult parts.

Of course, what he actually does is accept some simplification of the real world that the computer can deal with and then be disappointed that his model does not behave realistically.

I am now convinced that it is not the syntax which acts as a barrier to the user - he may generate mistakes because of it but it holds no terrors for him; after all, most of it is common from one language to another and with mathematics. The real barrier that I have seen at work is the fact that each individual statement of fact about the system is liable to require statements in the program at several different places. The user tries to understand the language without understanding its structure which is a reflection of the structure of the model of the system that he is trying to build.

I believe that for simulation activities to be successful on a big scale, the common structure of the models built, by a simulation language must be understood. If a computer dialogue can teach this, then real progress will be made.

For this aim to be realistic, we must agree on the general structure to be used in describing simulation models. This we have not done; there are three, possibly four, different structures and each has their adherents and languages based on them. Worse, the various schools have agreed that the structure is not important enough to argue about and have in effect agreed to differ, indeed agreed to pretend that there is no difference between the structures.

I know of only one worker in the field, Ziegler, now at the Weizmann Institute, who has contributed any serious analysis of the structure of a simulation model. He has got so excited about his discovery of the paradigm of reductionist science and a notation for obscuring its limitations, that he too treats the various methodologies as equally valid and useful.

I cannot accept this; we need one and only one methodology and in choosing this we must consider several dimensions of the modelling process. The usual excuses that "it all depends on what your problem is" are not good enough. If we squabble half-heartedly about methodology, and then each of us describe our favoured methodology in our own terms, disregarding the existence

of the others, how can we be surprised if the potential users turn away from us and re-invent some rather wobbly wheels?

The model of a general discrete event system must satisfy certain criteria:

i. it must be based on some simple concepts easily understood by the user, but which are powerful explicators of the behaviour of any system

ii. it must be decomposable into parts which have simple relationships to each other

iii. it must lead to a computer representation which enables the various operations to be performed efficiently

iv. for easy debugging, it must be capable of describing its own behaviour in the language in which it is described

v. it must be capable of parameterisation so that one model can be used for several configurations or with several different initialisations

vi. it must be capable of running free-standing or embodied in a program which explores the consequences of parametric changes.

The first step is to define the entities involved in a system. These have an endurance in the system and the purpose of the simulation is to describe the histories of those entities. Time plays a crucial role in our systems and at one level the simulation consists simply of a monotone set of times at which events occur. An event happens to an entity and changes its state or its relationship with other entities.

Now what drives time forward? The events form a network so one event is the cause or partial cause of another. The model predicts the time of each event at the occurrence of some earlier event. Unlike any real system, these predictions turn out to be completely accurate.

Times are variables and in any problem oriented language variables are descriptors of attributes of entities. But which entities should carry time, the events or the entities to which events happen? Before we answer this we consider events more closely. Since events happen to entities, then entities experience events. Is there a common component of these experiences? The entity either starts doing something or if it already is doing something, it stops it. A change of activity can be thought of as a double event - stop one thing and wihout any time lapse, start something else.

Thus every entity experiences an alternation of status - from being committed (busy) to available (idle) and back again. When it starts doing something we predict how long it will take and so predict the next event to that entity. We know what will happen at that event - the entity will go idle, the computer system can also know this - half of the events can be dealt with automatically.

How are the other events generated? When the state of the system (i.e. of the entities) is correct to start another task or activity. What happens at this event? - some entities change both their status and the values of other attributes. Which entities and which attributes are the user's responsibility. He describes the conditions for the start of an activity and the changes which take place at that start.

Thus we have a third type of entity - the activity.

Our general model can be described thus. Call the original entities machines. Machines alternate between the status of available and committed. Each change of status is an event. A machine moving from available to committed signals the start of an activity, one moving from committed to available signals the end of its participation in that activity and generally of the activity. The former event is generated by the conditions given in the description of the activity being satisfied. The latter, the end of the committal, is generated by the duration of the activity also part of its description.

Thus events are merely a secondary description of the outcome of machines taking part in activities. Our question of where to attach times is altered - do we attach them to machines or to activities? A computer science approach is helpful in the resolution of this.

Any machine can only be involved in one activity at any one time - there is one instance of the machine. However, there can be many instances of an activity, varying according to which machines are involved in it. These instances must be named; the list of participating machines is the only differentiating characteristic of these instances and so is the natural name to use.

Thus, although each machine can have an activity name as an attribute (we call this the state), it is more difficult to have the instance of the activity

as an attribute because the instance exists only when the machines are collected into the list. There is a circular definition involved.

Now consider the access paths to a time in the two cases. First, suppose times are associated with the machines; the time of the instance of an activity is derived from any machine on its defining list. On the other hand, if times are associated with the instances, then to find the time for a machine, the instances of its state (the activity name) have to be searched to find that using the machine; the time of this instance is the required time.

Instances of activities are temporary objects in the system and the number existing at any one time is variable. The number of machines on the other hand is fixed. List and graph structures are required to store the former, vectors are sufficient for the latter.

Thus the advantage lies with associating times with machines. Fortunately, this accords with the natural lay description of a system.

This is the activity approach; it uses machines and activities as atoms in which activities act on machines to form instances of activities. Events are merely the beginning and ends of instances of activities.

The representation of an activity is a set of statements divided into two parts. The first consists of tests of conditions for the activity to start. This is known as the testhead; it is separated from the remainder called the action body by a delimiter. The action body contains statements which effect changes of states which occur at the start of the activity; it also contains statements which specify the duration of the activity.

The process approach also uses machines as atoms and joins the activities into chains with a common machine to generate a chain of instances of activities; this - the process - is the second type of atom.

The event approach treats the machines and events as atoms. An event occurs to a machine at a time with consequences extracted from the activity description. In the process approach, time is associated with the instance of a process and signals the start of the next stage. In the event approach, time is associated with the event. In both cases, the number of the entities holding time is variable and involves list processing accesses.

The usual argument at this point is that these three approaches simply describe the same methodology with slightly changed emphasis, and each emphasis is best appropriate in certain different cases. But what we choose as atoms is determined by the properties we can assume for these atoms. The ideal in any reductionist theory is that the behaviour of atoms is independent of each other; reductionist theory describes behaviour of the whole by describing the behaviour of the parts.

Viewed this way, the event approach fares very badly. We speak of event networks to emphasise that events depend on each other. The process approach is not much better; when two machines co-operate the two controlling processes become entangled and indeed in some variants of this approach, one process has to split up into a part before the entanglement, and another which starts after it. The state of one process affects the behaviour of another.

On the other hand, the activity approach does give atoms with independent behaviour. Each possible instance of an activity is a statement of truth. Either the conditions for its start are not satisfied when it does not start or they are and it does. The existence of one instance of one activity does not affect the behaviour of any other instance or any other activity directly. Of course, because of resource (machine) limitations it may do so indirectly but if this indirect influence did not exist between atoms there would be no system - just a collection of isolated parts.

The implications for the user are enormous. He supplies a static description of the system - what machines it contains and a description of all the activities that can take place. These form a disconnected set. When assembled together the computer discovers the connections generated by resource limitations. The user does not write a program; he supplies data which the executive program runs to generate the model. The fact that half the data, that for the activities is best expressed in program-like language obscures the crucial fact that the interconnections are deduced from the data by the computer itself.

The usual counter to this argument is that the computer finds this sorting out of what will happen very expensive. We pay for user convenience in computer inefficiency. There are four counters to that argument. First, we could use a popular argument used by all sloppy thinkers, "computer power is now so cheap we no longer need to consider efficiency." I would

not do so, because if I can double the speed of my simulation I can afford within a given budget to double the amount of experimentation I can do; I have never yet met a case where more runs were not useful.

Secondly, there are two devices available to the user to give hints to the computer which remove the majority of the inefficiency. This is what a three-phase structure does. Thirdly, again apply a little computer science and we can improve the efficiency once again by getting the computer to construct a homomorphism of the described system which can act as a filter to avoid attempting to enter activities which cannot start.

Lastly, the time advance mechanism can be improved by classifying the entities we have called machines into two classes, active and passive. An active machine, hereinafter called a machine, is involved in an activity to process in some sense a passive machine, hereinafter called an item. The duration of the activity is carried by the machine. The item does not need a time. This reduction in the number of times improves the speed of the time advance.

Items cannot have a status; a machine is characterised by its state and status but an item after being processed by a machine is held in a queue or list. This defines the set of items which have become "available" after some activity. Machines on the other hand are never put in queues and are found by scanning a set to find those available in a given state.

This efficiency is generated by the computer,and does not rely on the user except in the most general terms. It is reliable as it cannot know how to take a risk or forget a possible circumstance; it has a consistency of efficiency as it applies general rules to generate the code,and does not need ad hoc inventions to achieve it.

The activity approach is not well known in the US and I have taken this opportunity to describe its advantages not only in the hopes that I can interest you to give it serious consideration, but also because an understanding of it is necessary to follow the description of the work of the Simulation Centre at Lancaster University.

The central aim of our research is to find ways of constructing very large and/or very complicated simulations, in an economic manner. We apply the classic reductionist technique of decomposing the simulation into parts;

each part is a simulation and can be developed as a free-standing simulation before the parts are assembled into a whole.

There are four dimensions of complexity which we have tackled; first,we considered how to generate a system which could model both continuous and discrete variables in one model. Attempts have been made to do this in the past but most workers admit that this is an exceedingly hard task; they have used a process or event approach to the discrete part. Our work shows that the activity approach generates a simple robust scheme.

Our second project is concerned with very large simulations involving a large number of machines. There are two, dual, ways of regarding this. We can decompose a large simulation into parts and connect them together. Alternatively we can start from already existing simulations of parts of a system and assemble them into one simulation. This latter approach requires that the parts to be assembled can be used without alteration, preferably without recompilation.

Our third project allows the decomposition of a simulation into levels. Each level is responsible for one aspect of the machine's behaviour. For example, one level may be a normal simulation of a set of failure-free machines, another is concerned with the breakdown of the machines based on an elapsed time life between breakdowns,and a third with breakdown based on a wear controlled life.

Another example is that a second level can be added to a normal simulation to observe the behaviour of the system and display it on a VDU; alternatively the second level may be used to calculate a production schedule by a deterministic simulation.

In all these cases, the original simulation is used without alternation. Each level proceeds independently but certain events cause interventions of one level upon another. The interventions are represented by another simulation - an intervention level simulation.

The last and most ambitious project is concerned with unifying these three projects. All of them involve the common concept of running several simulations in parallel and merging them to form a single time history. A simulation could involve both discrete and continuous elements, be so large as to require geographic decomposition,and involve so many aspects of the machine's behaviour as to require several levels of each geographic part.

Can we find a means of decomposing a system in all three ways simultaneously?

Each of these projects involves solving two computer science problems. Each part is a simulation which contains various system variables, the clock time, the times and states of machines, etc.

These are given, by both the system and the user, common names which have different meanings in the different parts. They must all be accessible by a controlling program which co-ordinates the various simulations. We require a programming language which allows the necessary flexibility of binding names to objects.

Conversely, the mechanisms for communication between parts must involve data structures accessible from both parts, but for security not accessible by any other parts; moreover each part may have chosen different names for the common data structures. This is the problem of different names for the same data structure in different contexts; the former problem is that of the same name for different data structures in different contexts.

Most programming languages do not have the necessary flexibility of name binding to allow our requirement for the use of unaltered program text in different combinations. Fortunately, we had access to a language which has been designed with this flexibility and by its use many difficulties were avoided. I give now a sketch of these computer systems.

The mixed simulation scheme treats continuous and discrete variables in separate parts. The activities of the discrete part now have two sections to their test heads; the first concerns the condition of discrete variables and the second those of the continuous variables. These latter conditions, known as "triggers", are gathered together in one program block known as the trigger block. This requires a facility to compile code for one block from text for another, another facility of the language we use, not commonly found in other languages.

A simulation starts by executing the discrete simulation but only proceeding as far as the trigger in each activity. An activity satisfying the discrete tests is said to be "hanging" and a record is made of the triggers encountered. The discrete tests (only involving discrete variables) will be satisfied by any time between the last two time beats; the whole activity test head will be satisfied when the trigger is satisfied.

The continuous part now integrates the differential equations, testing all the recorded triggers after each time step. When a trigger is satisfied, the time is adjusted to a value on the discrete time scale and this is taken as the current time. The discrete activities are executed again and this time their action bodies are also executed.

These action bodies can alter the definitions of the differentials of the continuous variables and thus effect step changes in differentials. Step changes in the continuous variables are dealt with in activities of the continuous part which are also executed at this time.

The work of the first pass through the activities is to limit the number of triggers which are tested at each time step for the continuous sytem and thus improve its efficiency. Another device to increase efficiency is to partition the continuous variables into groups and integrate each group with an independently chosen step length. The groups are formed so that variables that occur together in a trigger or a differential definition are in the same group; the compiler constructs these groups and the user is not concerned with this book-keeping task.

The equations are integrated by an RK-method with automatic step adjustment based on the estimate of the truncation error. Inverse interpolation is used to find the time at which the trigger is satisfied.

This formulation using the activity approach seems to dissolve all the synchronisation problems encountered in other approaches and attention has been focussed on deriving a notation which allows economy of description and execution. For example, the differential formulae are written in parametric form and any one formula can be applied to a whole set of dependent variables.

The general structure can be described as two simulations which intervene with one another. The continuous simulation intervenes with the discrete simulation forcing extra time beats into its history when triggers are set. The discrete simulation intervenes with the continuous simulation to provide the triggers and changes of the differential equation system definition. All the interventions take place in the time advance.

Turning to the geographical decomposition scheme, each region in this is simulated independently. The interaction with its environment is achieved by

the dispatch and receipt of messages. A message on arrival triggers a time beat, which in turn invokes an activity to read and distribute the message. Conversely the output of a message sends a trigger to the recipient region.

Messages are kept in records of a file; each file is shared between two regions and a mapping of message ports between regions is constructed so that triggers can be sent correctly.

The time advance mechanism maintains a vector of times. The time to the next event is found for each part and entered as an element of the vector. Time is advanced to the minimum of these elements. A list of returning parts is then made and the activities of each returning part are executed. These may generate triggers which in a last phase cause further activities to be executed to pass the messages.

On the next time advance a new time is found only for those parts which returned; this reduces the work of the time advance significantly. Slow moving parts are not advanced on every time beat.

There are various modes of use of this scheme. Each part can be developed independently; alternative programs are provided to send messages to it and the messages sent can be recorded or displayed. These alternative programs are pseudo-simulations.

If the parts can be ordered so there is no feed-back from one part to an earlier one, then during one experimental run the messages can be recorded. Other experimental runs which do not affect the performance of the earlier parts can be omitted, and alternative programs used to supply the messages at the correct times from the records.

Given the records of messages received and sent by a part, a statistical representation of the part can be developed, and an alternative program written to generate output messages from input messages. Once again, the simulation of the part can be replaced by a simpler program.

Some changes of parameters may only affect the values of variables in the output messages. For example, the volume of product may be systematically altered. By processing the record of messages sent, a further run will again be able to avoid executing the simulation of this part.

An important class of problems, which can use this decomposition are those involving human intervention. One part represents the human controller and displays messages to him and accepts messages from him for output.

In general, each part has several alternative programs which can represent its externally observable behaviour (the messages it sends) and a complete program can be made up of any combination of choices of alternatives for each part.

The implementation of this scheme depends heavily on the facility of correctly binding names to data structures,and it would not be feasible to construct it without that facility.

The interventions between the parts consist of injecting time beat triggers into the time histories; they are all dealt with after the execution of the activities of a time beat i.e. just before the next time advance.

The multi-level decomposition scheme is more complex. The interventions are so numerous that the provision of message ports is impractical. Instead, the status changes of the machine on each level are signals of a potential intervention. Analysis shows that only a minority of status changes (specified by the original joint status at the interacting levels) actually require intervention. A mapping of these is provided by the system. The actual work of an intervention is defined by the state of the machine in the recipient level and a mapping of activities to the states is provided by the user.

Sometimes the intervention takes the form of additional tests in activities yet to be invoked. These tests are interpolated through a redefined interpretation of the delimiter used to separate the test head from the action body in an activity. This can induce work in both the recipient level and the source level. The recipient level work causes an intervention to the source level,and the normal state dependent mechanism invokes the source level work.

All the intervention work is done by simulations operating on the variables of one level,which can be written as additional activities executed at an injected time beat,prior to any normal work for that level.

Although the internal structure of the scheme is quite complex, the user is

quite unaware of it and he merely writes the additional simulations for execution at one of the original levels. Most of the activities are of standard form and can be invoked by the name sharing mechanisms.

Anyone who has attempted to write, for example, a simulation involving the possibility of the breakdown of machines will know how difficult it is to ensure that all contingencies are covered. This scheme gives a complete structuring of the system and because the original simulation can be developed before adding breakdowns the development is made much easier. The requirement that the original simulation is not touched is vital to this rapid development.

As with the geographic decomposition scheme, each level simulation can be replaced by an alternative program,and this is another powerful aid to development.

The full range of applications has not yet been explored, but once the general concept is grasped it seems to offer a fruitful solution to almost all the difficult problems of model formulation that we have explored.

Work on the project to allow all these forms of decomposition simultaneously is not yet very far advanced but the principles are quite clear and it will only involve detailed programming problems to implement it.

It will have been noticed that there have been several mentions of computer science during this talk. This is no accident; in my opinion, the problems of correct formulation of simulation systems is intimately bound up with their possible computer representation, the resolution of name ambiguities and the efficiency of their execution. If computer science cannot help with these, what else can?

This talk has concentrated on rather theoretical structural problems in simulation; this does not belittle the many successful practical applications that workers on both sides of the Atlantic have done.

My concern is to discover ways in which those successes can be achieved with less effort,and be repeated by more people. That involves a sound theoretical structure to the formulation of models of complex systems - not just described in general system science terms but in specific practical concepts and mechanisms which can be applied to a wide range of problems.