Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

SIMULATION MODELLING SUPPORT VIA NETWORK BASED CONCEPTS

Stephen C. Mathewson

Management School
Imperial College
Exhibition Road
London SW7 2PG, United Kingdom

ABSTRACT

The paper traces the development history of one program generator. Then it reviews the constraints inherent in a very simple network definition of the system logic. Alternative solutions for strengthening the system features are presented. Distributed processing, with the network as one of several nodes, is suggested as a unifying solution. The paper considers the implementation of this approach and reports on the progress to date.

1. INTRODUCTION

Recent simulation software has emphasised generic models SAME (Renault,1987), MAST (Citroen, 1987) & SIMFACTORY (CACI,1988). In this context the appeal of program generators has diminished. However, for general modelling, the author believes that they are an efficient method of coding a logical model.

The modeller's building blocks are simple nodes of restricted function - a queue and an activity - placed within a network. The arcs flowing into the nodes indicate the required resources. The arcs out of the nodes lead to the next function. This is the core of information from which a skeletal program can be automatically derived. The program is fault free both in terms of the operating logic and the semantics and syntax of the target language. The programmer can then expand the logic to meet the needs of the application. The system combines the benefits of a block oriented program with the diversity of a general language. The software has been used to model such diverse systems as helicopter ambulances, telephone networks, cheque processing machinery, FMS cells and pilot workload.

The purpose of this paper is to review the development of the methodology underlying the generator. It assesses the avenues for functional enhancement. As the object oriented approach has been used to decompose simulation models, so it suggests a similar approach to the modelling process.

2. PROGRAM GENERATOR DEVELOPMENT

This section reviews the DRAFT program generator system. It introduces the underlying network structure and relates this to the mapping employed by the generator. Figure 1. shows the current DRAFT modules. Their development extended over a long period. DRAFT (Mathewson,1974), was initially run in 1970. This was followed by the graphics module DRAW (Mathewson,1985), which permits computer assisted development of the interactive animation. The most recent development, SSIM (Mathewson, 1987) is a response to the impact, at the modelling stage, of large simulation models, with animation. As Figure 1 illustrates, the overall system is designed so that the features of all the earlier versions are subsets of later implementations. Growth in the system complexity has been matched by changes in the target language. Practical experience lead to the realisation that the language features had to be enhanced if the performance of the overall system was to match contemporary standards.

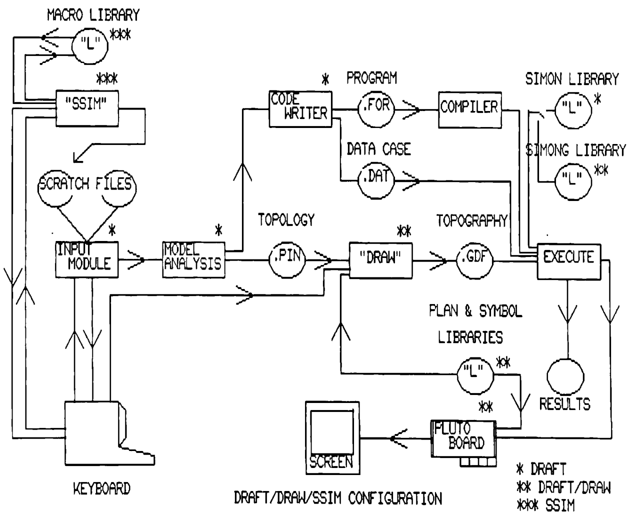


Figure 1. The DRAFT/DRAW/SSIM Structure

2.1 Entity Cycle Diagram

The 'entity cycle diagram' provides the input to the program generator. It records the physical flow of entities in the system. The flow of notional entities (e.g. permits) can also be included. The notation has a restricted set of symbols which are used to record the essential features of the model. The two basic symbols are circles and rectangles. A life cycle is defined as the path of an entity through circles that represent queues, and through the rectangles that represent activities. It may be helpful to imagine that, with the passing of time, the entity moves along the axis of the activity. Finally the entity emerges from the end of the activity symbol at the EVENT time. Activities which rely on one resource start when the required resource is released from the preceding activity, and may be called bound activities. Activities which need several resources are called conditional activities.

Figure 2 is an entity cycle diagram for the activity shown in Figure 3. A robot serves two lathes. Jobs join a common queue. The robot loads the jobs onto a free lathe. After the lathe is loaded the robot is released and the lathe runs through an automatic cycle. The finished job then waits for the idle robot to clear the machine. The robot is shared between the two lathes.

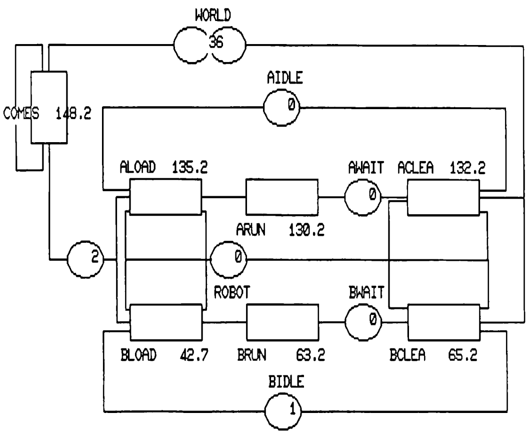


Figure 2. The Entity Cycle Diagram

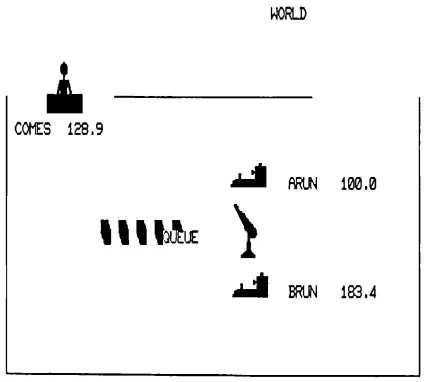


Figure 3. Iconic Modelling

The use of the entity cycle diagram for modelling resource constrained queuing systems, with computer implementations represented by HOCUS (Poole et al. 1977), CAPS (Clementson 1985) and DRAFT preceded much of the literature on network flow diagrams for simulation model analysis. Subsequently DeCarvahlo & Crooks (1976), Torn (1981), Schruben (1983,1989), and Nance and Overstreet (1987) have explored this theme. The literature shows that the entity cycle diagram can be mapped into any of the common simulation logic structures - process, activity scan, event base and three phase.

The operation of a program generator can be illustrated with a portion of Figure 2. Consider the initial arrival of jobs and the subsequent loading - ALOAD. As shown in Figure 2, a job comes, following some inter-arrival distribution, from the WORLD and ENTERS the QUEUE. If the lathe A is in AIDLE, the PUMA robot is in ROBOT, and a job is in the QUEUE then ALOAD is initiated. At this time the Puma leaves the ROBOT queue, the lathe quits AIDLE and the jobs moves from QUEUE. Completion of ALOAD releases the robot while the lathe moves on to activity ARUN.

The skeleton of Figure 4 contains part of the logic of Figure 2. All simulation languages have the appropriate statements for the actions specified here. Generation of program code therefore requires two steps. Given the network and target structure - what are the appropriate model activities and the correct statement sequence. Given the target language - what are the correct templates and the required parameter(s) e.g. ALOAD for the template:- 'CAN WE START ??????'.

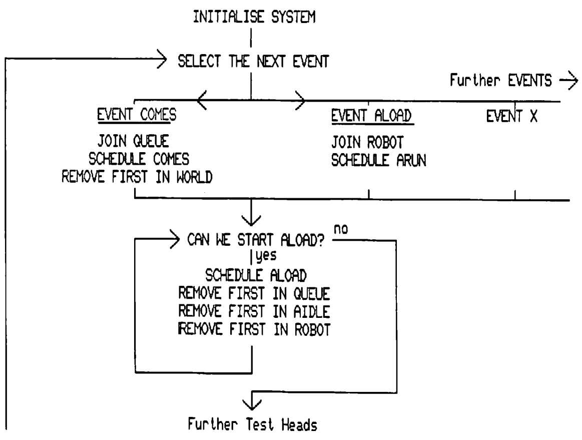


Figure 4. The Three-Phase Structure

2.2 DRAFT

The program generator is customarily structured as the single starred linked modules of Figure 1. This is specific to DRAFT but is generally representative. The model expressed as an entity cycle diagram is input via a terminal. The input/editor module identifies minor or semantic errors (e.g. use of reserved or duplicate variable names) and lets the user correct the entry. It also creates a back-up copy of the input on 'scratch' files. These may later be accessed for modification or stored as a compact copy of the model. The analysis module checks the input for errors, to be corrected on line, and prepares a coded internal file of the entity interaction within the model. This file then forms a general input to the program writer selected by the user. It is at this stage that the model description is mapped onto the particular structure - event, activity, or process - of the target language. The scope of the program generator includes the ability to: assume control over management of the host computer operation, provide error free coding which can be used as a model for further enhancement, and offer the advantages and convenience of a shorthand notation.

2.3 DRAW

The DRAW module (Mathewson,1985) was designed to support interactive animation. It operates as a post-processor to the DRAFT routine - see Figure 1 - from which it takes the topology of the entity cycle diagram. It maps this to the topography of the required animation.

Initially the animation was supported by an IO Research Pluto V.0001 board. This is an IBM compatible intelligent graphics board - with its own 68000 processor. Subsequently the software has been ported to Microsoft Fortran 5 and runs on a VGA board.

Historically animation requires the programmer to add explicit graphics code to the simulation logic. This writes to the display screen. In the context of the example and with animation instructions set in bold type, the event COMES might be modified to appear as:-

```
EVENT COMES

JOIN QUEUE
**SHOW A JOB ON THE PATH FROM COMES TO QUEUE**
**DISPLAY THE CONTENTS OF QUEUE AT QUEUE**
SCHEDULE COMES
REMOVE FIRST IN WORLD
```

Note that, typically, there is only a partial mapping between the logic of the code and the changes to the animation display. WORLD and COMES have both been assumed to be outside the screen. The path COMES-QUEUE uses some arbitrary entry point. Thus the display of the model activity can be a partial window on the system. We may clip real objects or omit artificial elements introduced as system semaphores. For example, in the program logic resource tokens might control access to single line traffic. In the final display these could be suppressed, or perhaps displayed in the form of traffic signals.

Adding explicit code for animation is redundant. The information required to animate is implicit in the coding. If, for example, an entity moves to the set QUEUE with the most recent event the completion of COMES, then the system could create animation along the path COMES-QUEUE and modify the status of the two nodes if it had access to animation rules. The parameters of the calls identify some of the required information - QUEUE in JOIN QUEUE. The relative position of program statements defines other aspects. The event COMES, as the most recent event with respect to the set manipulation instruction JOIN QUEUE, defines the active arc. Thus it is clear that an animation module can be incorporated in JOIN QUEUE if the event COMES has set a 'most-recent event' flag. In SIMONG the code offers implicit animation by reference to a graphics data base, whose information is addressed by node and arc. Thus whenever an entity moves between nodes, the arc and the relevant nodes are checked for animation rules.

During program execution these rules exist in named common, to which they are loaded from the *.GDF file (Graphics Description File). Within the GDF a queue node, for example, may be represented by the standard entity cycle diagram icon, a user defined symbol or as a point in space which is overwritten by an entity - thus creating the appearance of a queue. Similarly the associated activity and arc may also be represented by a diversity of animation styles. The topic is discussed later in the paper.

The graphics data base is the output from DRAW. This combines the topological information of the underlying network, the *.PIN file, produced by DRAFT, with a menu of animation effects to provide a topographical file (*.GDF). DRAFT simplifies the network flows before passing them to DRAW. This ensures that storage is not occupied by redundant polylines specifying the movement vectors of the individual entities which have been previously grouped in some fashion.

There are three possible styles of animation. Figure 2 shows a 'logical' representation of the entity cycle diagram. This can be created semi-automatically from the network information, using standard icons for the blocks of the entity-cycle diagram.

In practice, the complete entity cycle diagram often contains confusing detail. An increased feeling of identity with the model can be created by using 'realistic' icons in a 'realistic' context. In Figure 3 there is one lathe icon - the change of status is denoted by changing the colour of a lamp above the motor housing. This is the iconic presentation. It requires a user library of identifiable symbols on a spatially realistic background.

A further development which emphasises the importance of the spatial relations, and uses a real background, superimposes the movement of the icons upon a CAD quality backdrop. Figure 5 [taken from a system discussed in Production Engineering July/August 1985] demonstrates the clarity of this presentation. Backdrops can be copied from AUTOCAD files and can even use CAD techniques to effectively change the vantage point or zoom in and out of a model. The animated model of Figure 5 represents about eight hours work. The approach is also flexible - Figures 2 and 3 represent alternative outputs from the same program displayed on different screens.

When animation was added to the functions supported by the program generator, implicit animation removed the need to add code. The program generator section was kept untouched. Only statements in the target code concerning the definition of a graphics work space, represented as simple NAMED COMMON, were added. This maintained the integrity of the generator development. Animating via a set of display rules stored in an ASCII data base permits the user to modify the presentation after compilation, or to switch between libraries with and without graphic support. As a tool for management insight the system can be run on a micro, otherwise a fast mainframe may be preferred.

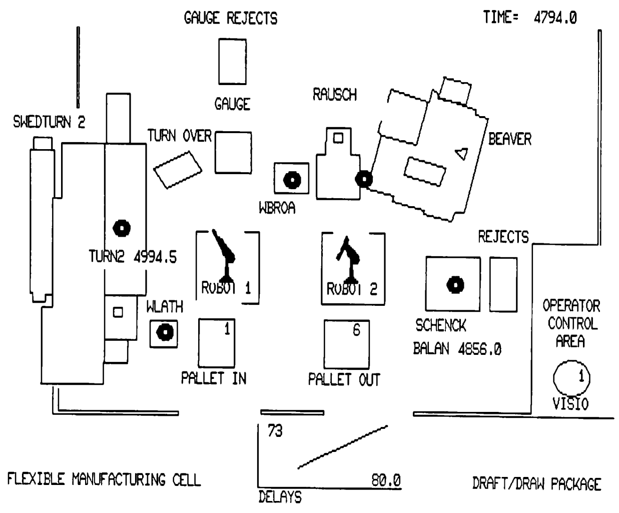


Figure 5. An FMS Cell

2.4 SSIM

The tedium of modelling large animated systems with identical but parallel processes triggered the next development (Mathewson, 1987). SSIM (Spreadsheet SIMulation) uses accepted techniques of spreadsheet programming as an alternative input to DRAFT. The cells hold the paths and symbols of an entity cycle diagram. As with spreadsheets, blocks may be selected and copied to replicate elements of the system. Blocks can be saved and retrieved as macros. Thus the user may define his own library of building blocks. System macros, represented as single cells, include belt and roller conveyors. TESS (Standridge, 1985) and RESQME (Kurose et al, 1986) are recent modular systems.

SSIM runs on personal computers using the IBM extended character set. Figure 6 shows the lathe model specified on the software. Models are built by specifying cells as queues or activities and then linking the nodes to reflect the entity flows. Control commands are analogous to other spreadsheet software. Arrows on the numeric key pad cause cursor movement and pan over the worksheet. Parameters can be selected from pop-up menus. Blocks can be used to duplicate sections of the flow and the housekeeping function of renaming nodes and entities is automatic. The left hand upper window serves as a prompt screen. This can be overwritten by other scrolling menus specific to selected program features. Below the prompt window, a status window relates to the cell on which the cursor is situated. At the base of the screen the 'cue line' indicates possible user response.

Cells may be Queues, Activities or Transporters. "Q" defines a set at the current cursor position and initiates further data entry for the NAME and the RULE. There are four available rules:- FIFO & LIFO or HIGH & LOW where entities are given priority depending on attribute value. "A" defines an activity at the current cursor, requests entry of the NAME and displays a menu of DISTRIBUTIONS. "T" defines a transporter at the current cursor position and initiates entry of NAME, TYPE and PATH. TYPE may be a belt conveyor, or a roller conveyor with stop belts, or an AGV.

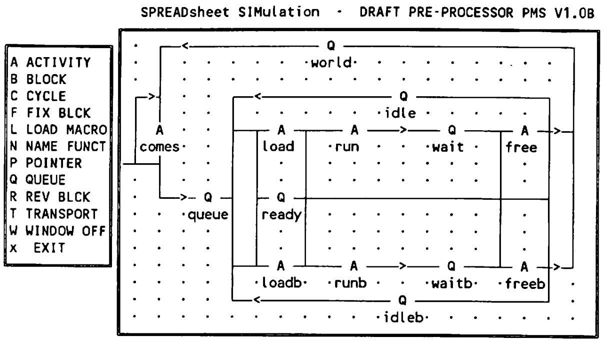


Figure 6. An SSIM Diagram for Figure 2

Tracing an entity path is analogous to entering the entity cycle in the DRAFT. The representation of a path is essentially a cosmetic - the path is defined by the nodes it visits. Obtaining the required appearance can be difficult and requires a process of trial and error. Apart from a rub-out feature, the 'Block Delete' is also useful in error correction. Path graphics are limited by the character based screen, so SSIM paths must merge prior to activities. This contrasts with the normal graphical representation. A completed SSIM diagram prepares an input file to DRAFT via the scratch file structure (See Figure 1.).

The spreadsheet format eases model building by providing an environment in which the user can develop the model in a natural manner, rather than in the rigid entry sequence enforced by DRAFT. Model building can also be spread over several sessions as the diagram can be stored at any point.

3. THE NETWORK STRUCTURE

DRAFT has an elegant algorithm for decomposing the network into quasi-parallel processing and a unique method of grouping the resource entities in a complex system. Adherence to a simple network maintains the investment in code, especially that concerned in program generation up to the point when the language dependent template is invoked.

This section discusses the nature of the compromises which maintain a simple network for logic definition yet provide a tool with broad capabilities. The network structure implies a default behaviour, variations are superimposed at the generator stage or later.

Successful schemes include those for 1) flow discontinuity, 2) the effective animation of the network and 3) the control of incomplete arc specification. Inelegant but functional solutions cover 1) the problems of statement sequence 2) limited nodal function and 3) the indirect link between network specification and animation.

3.1 Mapping the Network Structure to Reality

Interruption, whether a military threat to ships oiling at sea, a production breakdown, or hardware interrupts are common characteristics of the real world. These features are rarely supported in simulation software. The example below shows how an interrupt can be embedded in the simple network.

The lathe example has several breakdown scenarios. Perhaps the tool breaks, damaging the job beyond repair and requiring a teardown of the job from its jig. Or the coolant supply may become blocked, merely delaying the turning operation while the swarf is cleared from the filter on the recirculating pump. The difficulty of handling these concepts has been resolved by adding new interpretations to the structure shown in Figure 7.

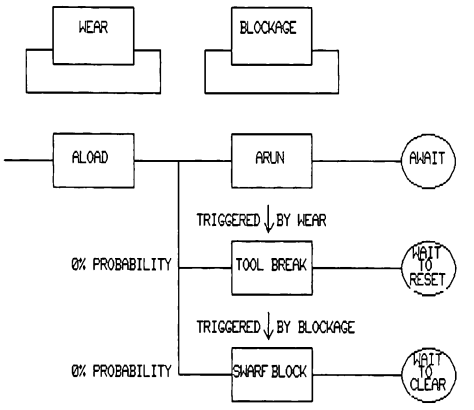


Figure 7. Introducing Interrupts

In the obvious sense, this may be interpreted as an activity ALOAD which leads to three alternative outcomes. When the generator identifies this pattern in the network, the user is asked whether the optional outcomes of ALOAD are conditional on the JOB (e.g. some castings require more machining than others), or occur by chance. Given the probability of TOOL BREAK and SWARF BLOCK as 0, the system knows that these are not alternative paths from ARUN. They are interrupts to ARUN, created by other events. In response to a query, the user states which events cause which interrupts. The resources in ARUN are those that have merged at ALOAD. The interrupt event need only change the event flag of the 'active data set' (Mathewson,1989) associated with ARUN to that of the chosen outcome. A separate interrupt event node offers a clearer definition than coding multiple alternative outcomes of the ARUN activity. It might appear a disadvantage that, at the level of the entity cycle diagram - Figure 7, the logical connection between TOOL BREAK and, say, the causative event WEAR is hidden. If this presentation is unacceptable, we can add, say, a cosmetic dotted line while presenting the program logic to a third party.

3.2 Mapping Animation to the Network Structure

DRAW introduces the idea of replacing explicit animation code with implicit calls to an animation data base. The animation must reflect the physical nature of the process and this implies different mappings for different situations. Consider the entity flow in Figure 3. At one place entity movement between nodes (COMES-QUEUE) equates with actual movement. At another, movement between nodes (ARUN-AWAIT) denotes a machine's change of state.

Table 1. Optional Animation Styles

ELEMENT	TYPICAL ANIMATION REPRESENTATION
QUEUE	CIRCLE WITH CURRENT QUEUE SIZE POINT WITH QUEUING ON INPUT ARC SYMBOL DEPENDENT ON STATUS - empty queue = busy robot - actual queue = idle robot HIDDEN
ACTIVITY	RECTANGLE - COLOR SET BY STATE STATUS LIGHT ON ICON - red is busy - green is idle ICON SUPERIMPOSED ON SYMBOL - job loaded on machine POLYLINE FOR AGV PATH/CONVEYOR - entity makes scale moves on downstream arc NUL - for multi-state machines
ARC	FLASHING POLYLINE ON MOTION PATH ENTITY ICON MOVES ALONG PATH HIDDEN - path is hidden
TITLE	ALWAYS ON ON WHEN QUEUE EXISTS ON WHEN ACTIVITY IS IN PROGRESS HIDDEN

Table 1 shows a non-exhaustive list of the available display options. They are chosen by setting the GDF variables. Effective animation can also require the removal of elements of the network. This is shown as the 'nul' or 'hidden' option. However there is also, with the current SIMON structure, a situation where increased complexity is required. Without graphics we can use subscripted activities [TURN(1), TURN(2) etc.] possessing class attributes. With animation each activity must be individually identified as a distinct node. Consider a hospital ward. The activity RECOVER must include an individual bed, with an individual coordinate reference for each patient. The preparation of models with multiple occurrences of the same class of action was the motivation beneath the 'Block Copy' in the SSIM program.

To conclude, effective presentation of a model requires a diversity of styles. The GDF can achieve this goal and isolate the user from program code.

3.3 Incomplete Arc Specification

There are disadvantages in the use of a rigid, data driven definition of animation. For example, it implies a complete specification of all the possible routes in the model. A machine shop may have many machines, with jobs scheduled by routing slips containing tens of operations on any order of machine. To include all possible routes in the GDF file would be a huge task - twenty machines is 20! routes. A simpler approach is to define a network where the job entity visits each activity at least once. This provides a template for all possible activity animation. Then the user codes a module to determine the geometry for any unknown move. In a factory crossed by access roads, it is, for example, possible to find the nearest road and then move to the next site. Whenever an unknown path occurs, SIMONG automatically refers to the user defined subroutine to determine the vector from source to sink. This vector is then handled as though it were attached to the standard input arc for the particular node.

This demonstrates a specific application of an underlying philosophy whereby the program provides defined interfaces for areas where the user wishes more freedom than is provided by the network based data.

3.4 Statement Sequence Versus Natural Logic Sequence

The use of display drivers within the modelling statements, e.g. JOIN QUEUE, create animation effects which depend on statement sequence. These effects were not foreseen. What are they? Inconsistencies normally arise in the 'C-Phase' portion of the code. As written this is:-

```
SCHEDULE ALOAD
REMOVE FIRST IN QUEUE
REMOVE FIRST IN AIDLE
REMOVE FIRST IN ROBOT
```

In the entity cycle representation this would, sequentially, change the state of the rectangle ALOAD, by redrawing the borders in red, decrement by 1 the content of QUEUE and indicate flow along the path QUEUE-ALOAD. An identical process would be followed for the AIDLE and ROBOT queues. From the viewer's perception this sequence:-

1) alerts the viewer to the area of change - ALOAD
2) explains the consequences of the activity in terms of the reduction in the QUEUE and service resources.

Although contrary to reality, the sequence is generally effective in informing the viewer of what is happening. In this context it does not appear inconsistent.

In a more sophisticated animation, the state of ALOAD may be indicated by an overlay which represents the work in progress at the site of the activity. Here the suggested sequence does impair understanding, as may be illustrated by Figure 8. Here an aircraft baggage container is delivered on a truck before being moved, via a scissor lift, to a train in an automated handling system. The picture shows a container on the lift. The next action moves the container onto the turntable of the train. The default animation would initially show another container on the turntable, thereby instantaneously displaying two containers, before the container on the scissor moves to the turntable. Manually rearranging the code sequence by placing the queue manipulation before the event scheduling and pre-setting the event flag is an ad hoc method to create a credible animation sequence. Indeed it may be that a policy of deleting from the queue before scheduling the event is more generally applicable and should be followed by DRAFT - but not always so. This is an outstanding problem to which the paper returns.

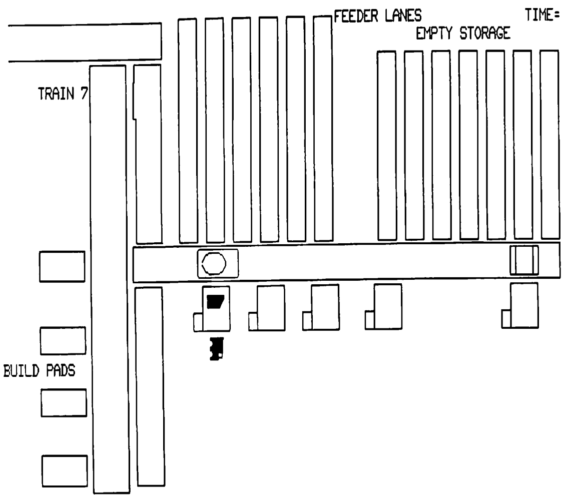
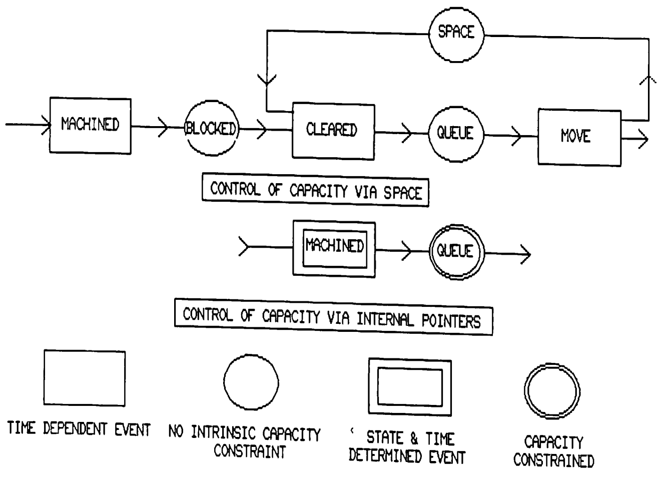


Figure 8. An Air Freight Container System

3.5 Nodal Function

Generic models possess further features, e.g. capacity constraints on queues. With a fixed capacity queue, any preceding activity, upon completion of its processing function, may be baulked because it cannot release its entity downstream. It is possible to represent such features by capacity flags as shown in Figure 9. Here SPACE, together with the additional queue BLOCKED and activity CLEARED can represent blocking. However addition of a capacity constrained queue plus a pointer from the activity to the downstream node permits the language to manage such situations implicitly as in the simpler, and arguably, more natural alternative shown below.



SPACE
MACHINED
BLOCKED
CLEARED
QUEUE
MOVE
CONTROL OF CAPACITY VIA SPACE
MACHINED
QUEUE
CONTROL OF CAPACITY VIA INTERNAL POINTERS
TIME DEPENDENT EVENT
NO INTRINSIC CAPACITY CONSTRAINT
STATE & TIME DETERMINED EVENT
CAPACITY CONSTRAINED

Figure 9. A Capacity Constrained Queue

Thus the demand for extended features can be met by redefining concepts and structures within the target language. Here the historical 'EVENT' of the SIMON language, an instant defined as a time dependent change of state of the system has been modified by an additional state dependent feasibility constraint - Is there available capacity in the output queue? The activity node has become more complex. The state of the node depends not only on time but on the contiguous states. While the alternative solution, using the original concepts, is available and while the facilities of SSIM can duplicate this as a macro, it seems premature to redefine the basic set of modelling primitives. However it is pertinent to ask whether the interpretation of the connectivity of the network, as constrained by the direction of entity flow, is generally sufficient.

3.6 Closely Coupled Systems

A system with a closely coupled problem definition and animation provides the best modelling environment. Imagine that we wish to add a third lathe to Figure 3. In, say, Witness the animation screen has a window at the top, in which to define a third machine, fill in the appropriate numbers for the reliability and performance, and specify the input and output buffers. After the chosen icon is positioned on the screen, the animation will run with the revised system.

In contrast DRAFT/DRAW generally requires a redefinition of the model followed by a redefinition of the screen. The problem lies in the conceptual differences of the two approaches. Generic models have a 1:1 relationship between the program specification and the screen, generalised programs have an infinity of mappings. Much of this flexibility arises from the wish to suppress explicit cycles controlling, say, breakdowns or capacity limits on buffers.

Generic models have functionally complex nodes with defined input and output cells, e.g. machines and conveyors. Call these DEVICES. The frequency of breakdowns and repairs, the match between levels of input and of output (e.g. 12 cans plus a carton produces a case) can be transferred to the program generator. We can provide a node defined as the device M (for machine). With a file of such nodes, the generator can replace the simple activity in the C-phase template with an activity test which looks for 12 cans and one carton. This does not fully emulate the features of 'on-line' change at the user level. The scheme has been used in a paper presented at the 1982 ORSA Conference - "Computer Simulation, A Research Focus" (Mathewson 1982). It is not however a general solution.

4. DEVELOPMENTS - CURRENT AND FUTURE

In the context of program enhancement, the value of comparisons with peer group simulation software is large. Without doubt the increased functionality of the latter has motivated improvements to this system. The published enthusiasm for Object Oriented Programming has also suggested a suitable path for system development.

4.1 Object Oriented Processing

In SIMONG any access to a node triggers an animation request. Because the PLUTO board sits on the bus and accepts GKS level graphics commands, the animation design is filtered and implemented at statement level. Albeit with a reference to a common graphics description block, every statement originates its own display instructions. The experience of developing other features in the language suggests another method. This subsection reviews these ideas in the context of a single node.

At the statement level, any language can be enhanced. For example consider the queue handling routines and trace the development of the original calls as their functionality has increased to include graphics and statistical monitoring. The code is programmed in Fortran but the concepts are general.

The original commands controlling the queue are those of entity manipulation - adding entities, removing entities, and sorting. The activity associated with a command is explicit in the name of the call:-

```
    CALL ADDLA (JOB, QUEUE)
```

adds the entity JOB as the last member of the QUEUE.

SIMONG demonstrates that the ADDLA statement can be enriched by the addition of an animation driver to the subroutine.

```
    CALL ADDLA (JOB, QUEUE)
```

explicitly adds the entity JOB as the last member of the QUEUE. Implicitly it sends a message to the PLUTO board which determines the animation for QUEUE, and for the arc between the preceding event and QUEUE.

Comparisons with generic models (e.g. Simfactory) emphasise the importance of automatic statistical records and summaries. The concepts which have been applied to the control of animation can also provide statistical features.

```
    CALL ADDLA (JOB, QUEUE)
```

explicitly adds the entity JOB to QUEUE. Implicitly it sends a message to the statistical record manager to update the queue record and so provide performance statistics on request. Polling a queue generates output as shown.

```
    RESULTS FOR QUEUE - QUEUE
MAXIMUM QUEUE SIZE -    6
TOTAL THROUGHOUT -   18
CURRENT OCCUPANCY -    0
EMPTY FOR  28.76% OR (TIME UNITS) -     65.20
OCCUPIED FOR   71.24% OR (TIME UNITS)- 161.52
AVERAGE OCCUPANCY       (TIME UNITS) -     30.61
```

Customarily the queue node is implemented as a pointer to an area of memory where some form of linked list is maintained. The queue node is an entry point to a dynamic data structure not the data structure itself. In this way, as entities move around the system, memory is released and reused in an efficient manner. In implementing the statistical monitor it seemed sensible to reflect this policy for the storage of statistical records. The focus thus shifts from functionality within the node, to message passing to a central monitor managing central storage.

The problem of queue monitoring arose at the time when the modelling of AGVs and conveyors was in review. The conceptual basis of the entity cycle diagram is that all state changes occur at events. At the coarsest level this constrains vehicle animation to moving to an intermediate point on the path, holding there for the duration of the total movement activity and, at that event instant, making a jump to the destination. In contrast the movement of AGVs or conveyors is a continuing change of state along co-ordinate axes! This default behaviour was modified by coding a centralised AGV/conveyor manager. Placed prior to the 'find next event' statement and running in background mode, it advances time at an incremental δt and updates any movement activity, so providing a realistic appearance.

4.2 Distributed Function Management

At this juncture, for historical reasons the software includes 1) variable action on receipt of a message 2) global suppression of messages and 3) local suppression of messages or local resequencing of messages. The first and second options are code free changes to the program. They rely on the user access to an ASCII data base which can be modified via an editor or via a form handler, with range and variable type checking on the fields. The third option arises for several reasons including the dichotomy between the information content of the entity cycle diagram as it illustrates the flow of entities and the information content required to **control** that flow. Thus while we know entity A and entity B are required to start activity C, we do not know how many of each are used. When event C occurs, we assume that entities can always flow into queue D. If D has a finite capacity, then we can only baulk activity C by using the diagram of Figure 9.

Although the flexibility of a system which incorporates several model styles may be useful, it is not an excuse for ignoring the approach which offers the strongest unifying concepts and the greatest possible functionality.

Combined with the insights of a central database for storage of queue statistics, the centralised vehicle motion manager suggests an alternative strategy for animation and control. In the new system a centralised animation sequence manager receives messages originating from the nodes and arcs. Such a manager creates appropriate displays whenever a 'paragraph' has been transferred to the animation buffer. A paragraph is defined by the time between the last event and the instant just prior to a time advance of the central clock. In a sense we replace the left hand system of Figure 10 by that on the right and trigger the manager whenever the code to 'Select the next event' is invoked.

With centralised animation rules defined in ASCII, as in the GDF file, the manual intervention which is now required to change the sequence of the automatically coded statements is replaced by an automated sequencing of the buffer commands.

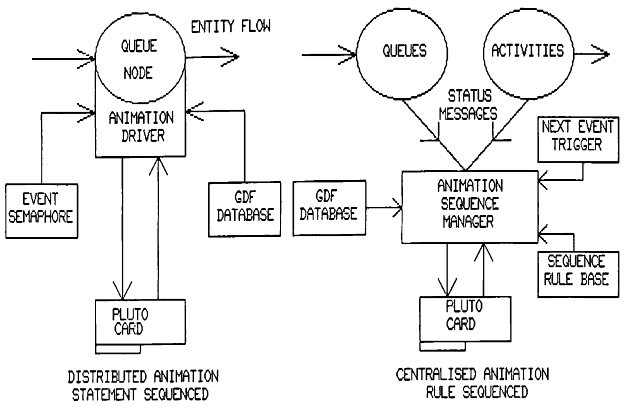


Figure 10. The Revised Animation Manager

Additionally porting the code, say from the PLUTO board to Microsoft Fortran 5, is made easier because the animation features are centralised.

There is thus an object oriented analogy to control of the model execution. In this context animation, statistical monitoring and system management can be expressed as the transfer of messages between the node/arc and specialist system functions. Figure 11 shows this.

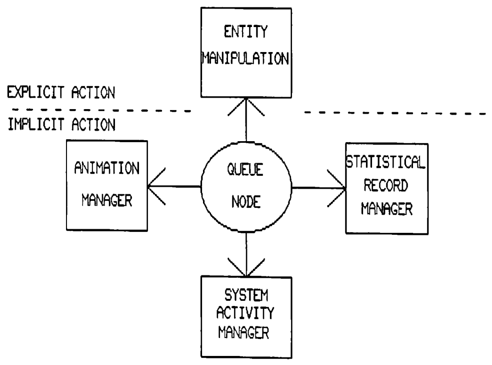


Figure 11. A Multi-Processor Node

The approach - a single message at the statement level causing differing responses at the simulation system level permits the user to tailor the model behaviour. Using the enhanced animation system as a paradigm we can identify further system managers.

The statistical monitoring manager refers to a SDF - statistical description form - to select the appropriate level of monitoring. This is analogous to SIMAN (Pegden 1982) which controls the experimental process via the definition of the experimental frame. Each node has a unique run-time definition of the required message response. This is set out as an ASCII file.

The system activity manager refers to the RDF - rule definition file. We have already observed that the entity cycle diagram, as a record of entity flow is an insufficient specification of the model. Rather than modify the primitives, the RDF adds another level of model definition. As the GDF file is developed from the interaction of the user with the topology produced by DRAFT, so the control rules can be generated by interactively building on this knowledge. Specification of a finite queue size automatically identifies the upstream activity from the network topology. In operation each C-Phase Test Head previously coded explicitly, is replaced by a single test of an activity flag. The flag is set whenever the control criteria are met. Returning to Figure 4 where, say, we are batch processing the contents of QUEUE in units of two, the RDF may have the form:-

```
    ALOAD 2*QUEUE ; 2*OUT
```

implying two units are used from QUEUE to start ALOAD, the other resources defaulting to 1. Output requires two free spaces in OUT.

The parametric selection of the control rules permits interfacing with optimising modules, who can read the system state from the messages and reset the RDF file.

In addition to the benefits of increased flexibility, the combination of object nodes communicating via messages to managers with their own data bases provides ease of program restart from given points in time. Such restarts provide the option of viewing the implication of alternative decision rules and are an important aid to system optimisation. A program with limited intrinsic features will have code added and new variables defined whenever features are extended. To dump the program status with a view to providing restart facilities rules is therefore complex. In contrast a program with modular storage and a minimum of extra variables is easier to manage.

There are also hardware implications, especially in view of the development of multi-processors and transputers. We note the present benefits of the Pluto board card as an off-line graphics processor. Similar applications devoted to simulation are sparse (Zenios 1987). Current research appears directed towards speeding the simulation by identifying relatively independent areas of activity and distributing these between processors. In contrast the off-loading of implicit 'service' calculations - statistical monitoring, scheduling and animation has no implications for the clock time synchronisation and would appear easier to implement for coarse grained parallel processing.

5. CONCLUSIONS

The paper demonstrates the value of a network structure and computer aided support in the coding of simulation models. In this context the network structure is a partial formulation of the rule base used by those who present their work as AI in program development. The particular benefits of a program generator - efficient code and flexibility for later modification - have been eclipsed by the management information that is built into generic models. The paper discusses how a revision of the system can redress this balance. A parallel is drawn between this solution and object oriented programming. The possibility of splitting the simulation into functional tasks rather than independent sub-networks is suggested as a possible implementation of parallelism for coarse grained processing.

REFERENCES

CACI (1988), *SIMFACTORY*, CACI Products Company, La Jolla, CA.

Citroen Industrie U.K. (1987), *MAST Manual*, Citroen Industrie, Automation Division, Warwickshire, U.K.

Clementson, A.T. (1985), *ECSL-Extended Control and Simulation System Users Manual*, CLE.COM Ltd., Birmingham, U.K.

DeCarvahlo, R.S. and J.G. Crookes (1976), "Cellular Simulation," *Journal of the Operational Research Society 29*, 1, 31-40.

Kurose, J.F., K.J. Gordon, R.F. Gordon, E.A. MacNair, and P.D. Welch (1986), "A Graphics-Oriented Modeler's Workstation Environment for the Research Queueing Package (RESQ)," In *Proceedings of the ACM/IEEE Fall Joint Computer Conference.*

Mathewson, S.C. (1974), "Simulation Program Generators," *Simulation 23*, 6, 181-189.

Mathewson, S.C. (1982), "Design for Software Modelling a Canning Line," Internal Papers, Department of Management Science, Imperial College, London, U.K.

Mathewson, S.C. (1985), "Simulation Program Generators: Code and Animation on a PC," *Journal of the Operational Research Society 36*, 7, 583-589.

Mathewson, S.C. (1987), "DRAFT/DRAW/SSIM – An Integrated Network Based Toolkit for Simulation," In *Proceedings of UKSC Conference on Computer Simulation*, R.N. Zobel, Ed. 154-159.

Mathewson, S.C. (1989), "The Implementation of Simulation Languages," In *Computer Modelling for Discrete Simulation*, M. Pidd, Ed. John Wiley and Sons, 43-44.

Nance, R.E. and C.M. Overstreet (1987), "Diagnostic Assistance Using Digraph Representation of Discrete Event Simulation Model Specifications," *Transactions of the SCS 4*, 1, 33-57.

Pegden, C.D. (1982), *Introduction to SIMAN*, Systems Modelling Corporation, Sewickley, PA.

Poole, T.G. and J.Z. Szymankiewicz (1977), *Using Simulation to Solve Problems*, McGraw-Hill (U.K) Ltd., Maidenhead.

Renault Automation (1987), *SAME Manual*, IFS Publications, U.K.

Schruben, L. and E. Yücesan (1989), "Simulation Graph Duality: A World View of Transformations for Simple Queueing Models," In *Proceedings of the 1989 Winter Simulation Conference*, E.A. MacNair, K.J. Musselman, and P. Heidelberger, Eds. IEEE, Piscataway, NJ, 738-745.

Standridge, C.R. and A.A.B. Pritsker (1987), *TESS – The Extended Simulation Support System*, Halstead Press.

Suri, R. and Y.T. Leung (1987), "Single Run Optimization of a SIMAN Model for a Closed Loop Flexible Assembly System," In *Proceedings of the 1987 Winter Simulation Conference*, A. Thesen, H. Grant, and W.D. Kelton, Eds. IEEE, Piscataway, NJ, 738-748.

Torn, A.A. (1981), "Simulation Graphs: A General Tool for Modelling Simulation Designs," *Simulation 37*, 12, 187-194.

Zenios, S.A. (1989), "Parallel Numerical Optimization: Current Status and an Annotated Bibliography," *ORSA Journal on Computing 1*, 1, 20-43.