A METALANGUAGE FOR INTERACTIVE SIMULATION

R.C. Heterick, J.A. Gerth, and N.D. Huebner

Virginia Polytechnic Institute & State University

ABSTRACT

Interactive computing environments ought to offer increased flexibility to persons interested in discrete system simulation. Much of this potential advantage may be lost, however, if the languages available to the model builder are not inherently interactive, or if he must be continually concerned with the basic housekeeping tasks involved in the execution of a simulation. Those who have become accustomed to the interactive environment desire to retain its full power and diversity for whatever tasks must be accomplished.

The intent of this paper is to describe several metafunctions, which, when taken together, form a metalanguage for interactive, discrete system simulation. The metalanguage may be used alone to effect a simulation or it may be embedded in a macro-structure to create an application language for simulation. That is, the metalanguage can be used directly by those who wish non-standard queueing disciplines, special statistics, etc. as well as by those who wish to create a pre-coded application language such as GPSS. An application of each type is described.

INTERACTIVE SIMULATION

Discrete system simulation is generally effected through either a unique program in a procedural language such as FORTRAN or by use of a more general command interpreter such as GPSS. Both are usually batch oriented uses of the computer.

The proponents of the unique program approach point to the inflexibility of queueing disciplines, output formats, statistic gathering, etc. as reasons for not using problem oriented languages. Advocates of the problem oriented languages use the high development cost of procedural language models to support their use of pre-coded packages. In either approach, debugging the code or logic can be time consuming because of the orientation to batch processing.

Reduced to the most basic terms, discrete system simulation involves creating and destroying transactions which move through entities. In GPSS, for example, these entities are gates, facilities, storages and queues. All of these are generically similar entities but with different constraints on size. Thought of as storages, a queue has unlimited storage, a facility unitary size, and a gate zero capacity. Transactions may be delayed in their progress through entities, thus requiring that a housekeeping mechanism ascertain the next event to occur and the transaction involved. The process may be viewed as either event or transaction driven – or both, if some provision is made for exogenous events to occur.

On the presumption that all queueing control is by transaction priority, a simple set of housekeeping and flow control functions, a metalanguage, can be constructed. Queueing discipline lies outside the metalanguage under the control of the modeller. The priority transaction parameter can be controlled and modified by the modeller to effect FIFO, LIFO or any other algorithmaically definable discipline. Differing queueing disciplines are easily applied to different entities in the model without recourse to chains or artifical constructs.

In order to construct the housekeeping and flow control functions comprising the metalanguage, the interactive environment must somehow provide a mechanism for ascertaining current execution status. Without such a feature, the programmer must create a sub-environment to execute the simulation – potentially losing much of the power and flexibility he originally had. Few interactive languages provide direct access to the execution status. In *APL*, (A Programming Language) [1], the programmer has available a system variable containing the line number of the statement currently being executed. This feature has permitted the construction of interactive simulation algorithms – an earlier example of which can be found in the work of J. Kaufman [5].

The metalanguage functions outlined above appear, and perhaps are, trivial – but coupled with a mathematically sophisticated interactive computer language, this simple set of housekeeping routines can be quite powerful. A reasonably complete set of GPSS-like commands for use as an application language for students was written in one week. The power and interactivity of *APL* can often reduce coding effort by at least one order of magnitude.

METALANGUAGE DESCRIPTION

APL, for those unfamiliar with it, is an algebraic notation whose strengths are consistency, terseness, and the ability to process N-dimensional, rectangular arrays. As a computer language, it generally operates as an interactive interpreter in terminal-oriented environments. The primitive functions and operators of *APL* are either monadic, requiring only a right argument, or dyadic, requiring both a left and right argument. Each line of *APL* is interpreted from right to left with functions executed as they are encountered. There are no implicit precedence rules, precedence must be explicitly stated with parentheses. Variables may have any number of elements in any rectangular structure and both of these attributes may be changed during execution with no need to prespecify storage requirements. Beyond a rich set of mathematical functions, *APL* provides a variety of primitive functions to select, order, and restructure data including powerful indexing and parallel processing capabilities.

In addition to the primitive functions, *APL* allows the creation of defined functions. Defined functions are composed of one or more *APL* statements and are executed in their entirety whenever they are encountered. Each function may have one of three syntaxes – monadic or dyadic like the primitives, or niladic, requiring no argument. Also, a defined function may or may not produce an explicit result, and may localize a name to itself and its descendants. Many books are available on the language, among them are references [1] and [6].

The metalanguage is comprised of six variables and eight functions. In addition, the *APL* system variable *⎕LC*, the line counter, is used to extract the line number where the simulation is to execute next. The following brief description of the metalanguage should provide a reasonably clear picture of its structure. None of the functions is over ten lines long.

METAVARIABLES

<u>*T*</u> – Transaction Matrix
This matrix contains transaction related information. Each column represents information about one transaction; the rows represent the different elements of information. Rows one to four contain system information used to maintain the housekeeping of the simulation. This information is:

- STATUS – a 0 if active, a 999 if the column is currently not in use, all other numbers indicate that the transaction is currently blocked and waiting for the entity designated by that number.
- NEXT CLOCK – The next clock time for an active transaction to become current. This number does not have meaning for nonactive transactions.
- NEXT LINE – The line number of the simulation model that should be executed when the transaction becomes current.
- PRIORITY – A number used to break clock ties between transactions. Ties are resolved in favor of the larger number.

Further rows may be added to this matrix by use of the *INITIALIZE* function. The application language author can then assign any desired transaction related information to these rows.

<u>*E*</u> – Entity Matrix
This matrix contains information concerning entities in the simulation model. Each column contains data concerning one entity. The first two rows of this matrix contain specific system information which is used by the metafunctions *GET* and *FREE*. This information is as follows:

- SIZE – The maximum capacity of the entity, a 1 for a sequential processor, larger for a parallel processor.
- CURRENT USAGE – The amount of the entity currently in use.

This matrix may be augmented with other entity related information, entity related statistics being a typical use. The number of extra rows can be specified in the *INITIALIZE* function.

<u>*C*</u> – Clock
The current internal time of the simulation.

<u>*I*</u> – Index
This variable is the index of the column in the transaction matrix, where the current transaction is located.

<u>*EX*</u> – Exogenous Event Matrix
See the description of functions *EXEVENT* and *NEXTEVENT*.

<u>*ES*</u> – Empty Slot Vector
This is a vector of free columns in the <u>*T*</u> matrix used for storage management by the functions *CREATE* and *DESTROY*.

METAFUNCTIONS

INITIALIZE
A monadic function that sets initial values for several metavariables, as well as

setting the size of both the transaction matrix (T̲) and the entity matrix (E̲). The right argument is a two element vector. The first element is the number of user rows to be added to T̲, the second is the number of user rows to be added to E̲.

CREATE
A monadic function that adds transactions to the transaction matrix. The right argument is a matrix argument the column(s), of which represent the transactions. The number of rows of the right argument must equal the number of rows in T̲. No result is returned by this function.

DESTROY
A monadic function that accepts a vector argument of indices of transactions (columns) in T̲. These transactions are then removed from the T̲ matrix. The result returned is chosen by *NEXTEVENT*.

NEXTEVENT
A niladic function that returns the next line in the model to which the flow should transfer. This function also updates the clock (C̲) to its next value and the transaction index (I̲) to its next value. The line number that is returned is either selected from T̲ or from the exogenous event matrix (EX̲) based on the following criteria:

- The event with the lowest next clock time is selected, and the associated line number is used.
- If there is a tie between an exogenous event and a transaction, then the exogenous event is chosen.
- If multiple transactions tie, then the transaction with the highest priority is chosen.
- If the transactions priorities tie, then the one with the lowest index in the T̲ matrix is selected. This effectively results in a psuedo-random choice for a warmed-up model.

EXEVENT
A monadic function that adds events to the exogenous event matrix (EX̲) and keeps these events sorted in ascending clock sequence. The right argument is a two element vector. The first is the exogenous event time, and the second is the line number in the model to branch to at that clock time. A typical use of this function is to set an exogenous event to return to the *CREATE* function to generate a new transaction. No result is returned by this function.

GET
A monadic function that controls the flow of transactions through an entity. The right argument is a two element vector the first element being the entity number (column index in the entity matrix (E̲)), and the second being the quantity of the entity that the transaction is requesting. The transaction will either be blocked, or the entity will accept the transaction. Blocking will occur in the following circumstances:

- A currently blocked transaction has a higher priority than the seizing transaction.
- The transaction is requesting more space than is currently available in the entity.

The result returned by this function is the line number in the model that should be executed next. If the *GET* is successful then the result will be a null vector, if it fails then the result will be determined by *NEXTEVENT*.

FREE
A monadic function that accepts a two element argument. The first element is the

entity that should be freed, and the second element is the amount by which the entity's current contents will be decremented. If the entity is a parallel processor (size>1), then all blocked transactions for that entity will be marked active again. If the entity is a sequential processor (size=1) then only the transaction with the highest priority will be marked active. No result is returned by this function.

DELAY

A monadic function that sets the current transaction's next clock time and next line number in the transaction matrix and then calls *NEXTEVENT*. The right argument is a two element vector. Element one is the clock offset, and element two is the line number to branch to when the transaction becomes current. The result returned is the line number chosen by *NEXTEVENT*.

METAFUNCTION LISTINGS

```
INITIALIZE ARG
T←((4+ARG[1]),0)⍴0
EX[1;1]←⌊/ES←⍳1↑,EX←2 1⍴C←0
E←((2+ARG[2]),0)⍴0

CREATE ARG;NS
→FILLSLOTS×⍳0≥NS←(1↓⍴ARG)−⍴ES
ES←ES,(1↓⍴T)+⍳NS
T←T,((1↑⍴T),NS)⍴0
FILLSLOTS:T[;(1↓⍴ARG)↑ES]←ARG
ES←(1↓⍴ARG)↓ES

LN←DESTROY ARG;I
I←((1↑⍴T),1)⍴(1↑⍴T)↑999
T[;,ARG←ARG[⍋ARG]]$I
TEST:→DEC×⍳(1↑⍴ARG)=1↓⍴T
ES←ES,ARG
→EXIT
DEC:T←0 ¯1↓T
ARG←1↓ARG
→TEST×⍳1≤⍴ARG
EXIT:LN←NEXTEVENT

LN←NEXTEVENT;C
C←(T[1;]=0)/⍳1↓⍴T
C←(⌊/T[2;C])=T[2;C])/C
→EXT×⍳0=I←1↑((⌈/T[4;C])=T[4;C])/C
C←T[2;I]
LN←T[3;I]
→0×⍳C≤EX[1;1]
EXT:C←EX[1;1]
LN←EX[2;1]
EX←0 1↓EX

EXEVENT ARG
EX←EX,ARG
EX←EX[;⍋EX[1;]]

LN←GET ARG;K;E;AM
E←1↑ARG,AM←1↑1↓ARG
→CKSIZE×⍳0=⍴K←(T[1;]=E)/⍳1↓⍴T
→WAIT×⍳T[4;I]<⌈/T[4;K]
CKSIZE:→GOT×⍳E[1;E]≥E[2;E]+AM
WAIT:T[1 3;,I]←2 1⍴E, 1↑⎕LC
→0,LN←NEXTEVENT
GOT:E[2;E]←E[2;E]+AM
LN←⍳0

FREE ARG;K
E[2;ARG[1]]←E[2;ARG[1]]−ARG[2]
→0×⍳0=⍴K←(ARG[1]=T[1;])/⍳1↓⍴T
→SET×⍳E[1;ARG[1]]>1
K←1↑((⌈/T[4;K])=T[4;K])/K
SET:T[1 2;,K]←⍉((⍴,K),2)⍴0,C

LN←DELAY ARG
T[2 3;,I]←2 1⍴(C+ARG[1]),ARG[2]
LN←NEXTEVENT
```

APPLICATION EXAMPLES

A GPSS-LIKE LANGUAGE

The metalanguage can be used to create *APL* functions with GPSS-like syntax. A function is written for each command desired in the language. Two such functions are shown as examples of using the metalanguage to construct an application language.

```
LN←ADVANCE T
→0×⍳I<1↑LN←⍳0
LN←DELAY T,1+¯1↑⎕LC

LN←SEIZE N
→0×⍳I<1↑LN←⍳0
LN←GET N,1
→0×LN
E[3 6 7;,N]←3 1⍴(E[3;N]+1),C,I
```

The user enters at the terminal an *APL* function composed of the GPSS-like commands. An example simulation function drawn from [3] might be

```
EXAMPLE N
SIMULATE
CORE:STORAGE 10
GENERATE 300+100×NRM
ASSIGN 1,UNIF 3 8
QUEUE P 1
→SEIZE P 1
DEPART P 1
→SEIZE 8
→ADVANCE 50
→ENTER CORE
→PREEMPT 9
→ADVANCE XMIT
RETURN 9
RELEASE 8
→SEIZE 9
→ADVANCE UNIF 150 450
RELEASE 9
→SEIZE 10
→PREEMPT 9
→ADVANCE 15
RETURN 9
LEAVE CORE
→ADVANCE 75
RELEASE 10
RELEASE P 1
→TERMINATE 1
→START N
END
```

The output of this system has been made to conform to the basic format of GPSS output with the exception of restricting line length to fit CRT terminals.

```
EXAMPLE 25
RELATIVE CLOCK      8980         ABSOLUTE CLOCK      8980
FACILITY    AVERAGE      ENTRIES     AVERAGE     TRANSACTION NUMBER
 NUMBER   UTILIZATION              TIME/TRAN    SEIZE       PREEMPT
   3          .688          8        772.33
   4          .217          4        486.25
   5          .644          8        723.00
   6          .116          2        519.04
   7          .421          5        756.72
   8          .174         27         58.01
   9          .866         77        100.96
  10          .252         25         90.40
STOR  CAPACITY  AVERAGE  AVERAGE   ENTRIES   CURRENT  MAXIMUM   AVERAGE
NUMB            CONTENT  UTILIZ              CONTENT  CONTENT  TIME/TRAN
  2       10       2       .173       26        1        4       598.00
 QUEUE    MAXIMUM   AVERAGE   TOTAL    AVERAGE    ZERO     CURRENT
NUMBER    CONTENT   CONTENT  ENTRIES  TIME/TRAN  ENTRIES   CONTENT
   3         2        .37       8      409.97       4         0
   4         2        .11       4      251.39       2         0
   5         2        .37      10      334.02       4         2
   6         1        .00       2         .00       2         0
   7         1        .04       5       71.81       4         0
SIMULATION COMPLETED
```

A few comments are in order as to the differences between GPSS and this GPSS-like language.

- The Branch Arrow – Those commands which represent activities which can block the progress of the current transaction or otherwise transfer control to a new transaction are preceded by a branch arrow (→).

- Interspersed *APL* Code – *APL* statements may be placed anywhere in the model so long as they are syntactically correct. The random arrival times are passed to the GENERATE command through execution of an *APL* function named *NRM*. The parameter of one of the ADVANCE commands is shown as a variable *XMIT* set outside the simulation function. Another of the ADVANCE commands takes a parameter computed by the *APL* function *UNIF*.

- Interrupting Execution – The *APL* trace and stop vectors could have been set to display flow or interrupt execution. Whenever execution is interrupted, the values of all metavariables are available for inspection.

- Parameter Lists – The parameter strings for commands are greatly simplified by use of interspersed *APL* code. The FUNCTION and VARIABLE commands are no longer necessary.

- Execution Speed – As each line is interactively interpreted, this GPSS-like language will execute at about one-tenth the speed of GPSS. Extremely long simulations could be debugged using an interactive application language but would better be executed with the terminal running disconnected from the system.

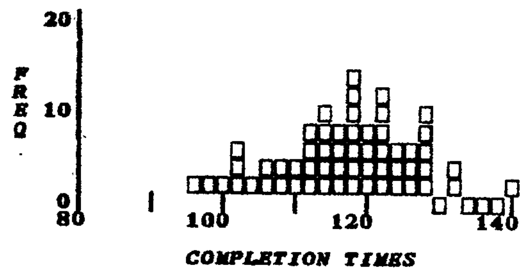
PERT NETWORK SIMULATION

The metalanguage is used here in a different context from that discussed previously. Rather than constructing an application language, it is used to assist in constructing a specific simulation model. This GASP-like use of interactive simulation in this context is discussed in chapter six of [3]. The particular example is drawn from chapter twelve of [4].
Two global variables, *NET* and *NETTIME*, must be set. *NET* is a N (the number of activities) by 3 matrix that holds the start node, end node, and the current activity duration. When running multiple simulations of the network this third column will be varied by the control function *SIMNET* using the activity duration statistics held in the second global variable, *NETTIME*. This variable is also an N by 3 matrix, it contains the optimistic, most likely, and pessimistic duration times for each activity. The two variables set for this problem are displayed below. *NET*[;3] is shown containing sample activity times.

```
      NET,NETTIME
 1  2 20 18 28  22
 1  3 40 20 30 100
 1  6 28 18 20  70
 2  4  8  6  7  14
 3  5  0  0  0   0
 3  6 10  7  9  17
 4  5 30 20 25  60
 5  6 20 17 18  31
 5  7 24 18 20  46
 6  8 10  8 10  12
 7  8 12 11 12  13
 8  9  0  0  0   0
 8 10 10  8  8  20
 8 11  8  8  8   8
 9 10  6  5  6   7
10 11  4  2  3  10
11 12  4  3  4   5
```

```
      SIMNET 100
     OBSERVATIONS 100
   REALIZATION TIME   1   2   3   4   5   6   7   8   9  10  11  12
   ________________________________________________________________
      AVERAGE TIME    0  21  42  29  61  82  86  99  99 109 114 118
      MAXIMUN TIME    0  22  76  32  80 101 108 121 121 130 136 141
      MINIMUM TIME    0  19  10  24  46  65  61  79  79  90  93  97
```

```
CRITICALITY INDEX
     ACTIVITY    1- 2       0.870
     ACTIVITY    1- 3       0.130
     ACTIVITY    1- 6       0.000
     ACTIVITY    2- 4       0.870
     ACTIVITY    3- 5       0.130
     ACTIVITY    3- 6       0.000
     ACTIVITY    4- 5       0.870
     ACTIVITY    5- 6       0.180
     ACTIVITY    5- 7       0.820
     ACTIVITY    6- 8       0.180
     ACTIVITY    7- 8       0.820
     ACTIVITY    8- 9       0.010
     ACTIVITY    8-10       0.980
     ACTIVITY    8-11       0.010
     ACTIVITY    9-10       0.010
     ACTIVITY   10-11       0.990
     ACTIVITY   11-12       1.000
```



While the solution of this problem is of no particular interest, the variety demonstrated in the ouput is easily achieved with an interactive language such as *APL*.

In building this example simulation model, the metalanguage functions have been used directly by the modeller to handle all of the basic housekeeping. The functions shown below were written by the modeller to control the flow through the network, and gather data. Additional functions used for analysis and output formatting are not shown. The master function is *SIMNET* which provides repetitive execution of the network by invoking *PERT* for each trial and then printing the results. *PERT* uses the information in in the variable *NET* to define flow paths and times and returns the network completion time as an explicit result. *GENERATE* is used by *PERT* to provide transactions to flow through the model.

```
SIMNET NRUN;K;NT
⍝ XEQ A PERT SIMULATION 'NRUN' TIMES
⍝ KEEP ACTIVITY 'CRITICALITY INDEX'
⍝ AND MIN, MAX, AND AVG COMPLETION
⍝ TIMES FOR THE NETWORK
NT←BETA∆MEAN NETTIME
NT←NT,[1.5]BETA∆SD NETTIME[;1 3]
MINTIME←(⌈/,NET[;1 2])⍴⌊/⍳0
AVGTIME←MAXTIME←(⌈/,NET[;1 2])⍴I←0
CTIMES←⍳1↑CRIT∆INDEX←(1↑⍴NET)⍴0
⍝ SET ACTIVITY TIMES FOR THIS ITERATION
LOOP:NET[;3]←⌈NT[;1]NORM∆DIST NT[;2]
CTIMES←CTIMES,PERT
⍝ UPDATE STATISTICS
K←CACTIV CPATH
CRIT∆INDEX[K]←CRIT∆INDEX[K]+1
AVGTIME←AVGTIME+E[5;]
MAXTIME←MAXTIME⌈E[5;]
MINTIME←MINTIME⌊E[5;]
⍝ LOOP IF NOT DONE
→LOOP×⍳NRUN≠I←I+1
PRTNETSTAT NRUN


CTIME←PERT;N;I;S;LN;COUNT
⍝ PERT NETWORK SIMULATION
INITIALIZE 2 0
COUNT←1↑⍴NET
S←1,0⍴,E←(5,⌈/,NET[;1 2])⍴0
⍝ START NETWORK
GENERATE ST
ST:N←''⍴T[5;I]
E[1 2;N]←1+E[1 2;N]
E[4 5;N]←(E[4;N]+E[1;N]×C−E[5;N]),C
→DELAY T[6;I],1+1↑⎕LC
N←''⍴T[5;I]
E[1 3;N]←(¯1+E[1;N]),⌈/E[3 1;N]
E[4 5;N]←(E[4;N]+E[1;N]×C−E[5;N]),C
I←I
→TERM×⍳(+/NET[;2]=N)>−/E[2 1;N←T[5;I]]
S←N
→TERM×⍳0=+/NET[;1]=S
GENERATE ST
TERM:LN←DESTROY I
→LN×⍳0≠COUNT←COUNT−1]
CTIME←, 1 ¯1↑E


GENERATE LN;K;I;T
K←(NET[;1]=S)/⍳1↑⍴NET
I←⍴K
I←1↑ES,1+1↓⍴T
T←NET[K;2]),[1]NET[K;3]
CREATE((⍉(I,4)⍴0,C,LN,0),[1]T
```

CONCLUSION

Our experience with the metalanguage so far has been as learning tool for ourselves and our students. Its use has permitted us to rapidly construct simulations and investigate the implications of alternative constructs. At the moment, the metalanguage contains no consistency checking or integrity protection other than than that provided by *APL* itself. Those who would use it directly must therefore have a moderate level of competence in *APL*. Of course, someone who constructs an application language could provide such features, and that may be the appropriate place for them.

The nature of the interactive environment has also raised the need for features not currently available. For example, the exogenous event function allows the modeller to schedule an interruption of the simulation at a specified time, but it may also be desirable to provide metafunctions to facilitate interrupting execution when the model attains a certain condition, e.g. a transaction of priority 10 is refused entry to an entity after 250 clock units. There is also some need for the metalanguage to optionally limit the computer resource consumption of a model in order to detect the runaway simulation which is often created during the development process.

In addition to new services which ought to be available to interactive simulation, there is the issue of identifying practices which are not appropriate. After creating the output routines for the GPSS-like application language, it was not clear that the automatic presentation of tables was desirable for an interactive user. It may be better to eliminate the predefined summary statistics altogether and provide a set of simpler selection functions to allow the user to rapidly scan large amounts of data or investigate some small subset of information. Perhaps even that is unnecessary since he already exists in a powerful computing environment.

On balance, it may prove to be wiser to leave the metalanguage simple and comprehensible. Its principal virtue may lie in minimizing the constraints faced by the model builder. Interactive computing itself is the major advantage presented to the user, and the goal of any such effort as this ought to be to enhance, not diminish, the texture of that environment.

REFERENCES

[1] Iverson, K.; An Introduction to *APL* for Scientists and Engineers; Technical Report No. 320-3019; IBM Philadelphia Scientific Center; Philadelphia, Pa; 1973

[2] ; General Purpose Systems Simulater III; IBM Corporation; White Plains, N.Y.; 1965

[3] Pritsker, P. and Kiviat, P.; Simulation with GASP: A FORTRAN Based Simulation Language; Prentice-Hall Inc.; Englewood Cliffs, N.J.; 1970

[4] Antill, J. and Woodhead, R.; Critical Path Methods in Construction Practice; John Wiley and Sons, Inc.; New York, N.Y.; 1970

[5] Kaufman, J.; *JACK'S APL* Computer Simulation System; Computer Center, SUNY-Binghamton; Binghamton, N.Y.; 1974

[6] Pakin, S.; *APL*\360 Reference Manual; Science Research Associates, Inc.; Chicago, Ill.; 1972