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 queuing 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 affected 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 queuing disciplines, output formats, statistic gathering, etc. as reasons for set 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 algorithmically definable discipline. Differing queueing disciplines are easily applied to different entities in the model without recourse to chains or artificial 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 n-adic, 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 \( MC \), 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

\[ I \] - 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.

\[ E \] - 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.

\[ C \] - Clock
The current internal time of the
simulation.

\[ I \] - Index
This variable is the index of the column in
the transaction matrix, where the current
transaction is located.

\[ E \] - Exogenous Event Matrix
See the description of functions **EVENT**
and **NEXTEVENT**.

\[ B \] - Empty Slot Vector
This is a vector of free columns in the \[ I \]
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
Meta-language (continued)

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

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

DESTROY
A monadic function that accepts a vector argument of indices of transactions (columns) in T. These transactions are then removed from the T 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 (C) to its next value and the transaction index (A) to its next value. The line number that is returned is either selected from T or from the exogenous event matrix (EX) 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 prioritites tie, then the one with the lowest index is chosen.

the T matrix is selected. This effectively results in a pseudo-random choice for a warmed-up model.

EXEVENT
A monadic function that adds events to the exogenous event matrix (EX) 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 (E)), 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-1[4+ARG(1)],0$p0
E-1[11]-[E]+11,EK-2 1$pC-0
ES-1[2+ARG(2)],0$p0

CREATE ARG:NS
-FILLSLOTS*400NS-1[1$pARG]-pES
ES-ES-1[1$p1]+NS
T-1[1][pT]+NS
FILLSLOTS*1[1$pARG]ES-ARG
ES-1[1$pARG]+ES

LN-DESTROY ARG:I
T-1[1],ARG-ARG[ARG][ARG]]SI
TEST-DECX1[1$pARG]=11$pT
ES-ES-ARG
-EXIT
DEC:T-0 $11T
ARG-1[ARG]
-TESTX1$1pARG
EXIT:LN-NEXTEVENT

LN-NEXTEVENT:C
C-1[1]=0/11$pT
C-1[2:1]
LN-2[3:1]
-0+ES[E11][1]
EXIT:LN-NEXTEVENT
LN-ES[4][11]
E-0 $11E

EXIT EVENT ARG
ES-ES-ARG
ES-ES[E11][1]

LN-GET ARG:K:E;AM
E-1[ARG,AM-11][ARG]
-CESIZE*40=PK-1[T[1]=E]/$11$pT
-WAIT=T[4][K]<T[4][K]
-CESIZE-GET=ES[11][1][2:E]+AM
WAIT=T[1:1],E-2 E;E,11][EC
-0,LN-NEXTEVENT
GOT=E[2:E]E;E]+AM
LN-0

FREE ARG:K
K-1[2+ARG(1)]=E(2+ARG(1))ARG(2)
-0$1=PK-[ARG(1)1][1]/$11$pT
-SET-ES[1][ARG(1)]>1
K-1[1][4[K]=1[4[K]]/K
SET-E[1:1,E]=01[p,E],3,p0,C

LN-DELAY ARG
T[2:1],I-2 1$pC+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<k<1|LN-10
LN-DELAY T;1+1|LC

LN-SEIZE N
=0<k<1|LN-10
LN-GET N+1
=0<k<1

H(3 6 7;3|N)-3,1
```

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 WIT
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
```

<table>
<thead>
<tr>
<th>FACILITY</th>
<th>AVERAGE</th>
<th>AVERAGE</th>
<th>AVERAGE</th>
<th>TRANSACTION NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER</td>
<td>UTILIZATION</td>
<td>ENTRIES</td>
<td>TIME/TRAN</td>
<td>SEIZE</td>
</tr>
<tr>
<td>3</td>
<td>+658</td>
<td>8</td>
<td>772.33</td>
<td>486.25</td>
</tr>
<tr>
<td>4</td>
<td>+517</td>
<td>4</td>
<td>723.00</td>
<td>756.72</td>
</tr>
<tr>
<td>5</td>
<td>+644</td>
<td>8</td>
<td>519.04</td>
<td>58.01</td>
</tr>
<tr>
<td>6</td>
<td>+112</td>
<td>2</td>
<td>190.36</td>
<td>90.40</td>
</tr>
<tr>
<td>7</td>
<td>+431</td>
<td>5</td>
<td>100.00</td>
<td>90.40</td>
</tr>
<tr>
<td>8</td>
<td>+174</td>
<td>27</td>
<td>100.00</td>
<td>90.40</td>
</tr>
<tr>
<td>9</td>
<td>+866</td>
<td>77</td>
<td>100.00</td>
<td>90.40</td>
</tr>
<tr>
<td>10</td>
<td>+292</td>
<td>25</td>
<td>100.00</td>
<td>90.40</td>
</tr>
</tbody>
</table>

| STOR | CAPACITY | AVERAGE | AVERAGE | CURRENT | MAXIMUM | AVERAGE |
| NUMB | CONTENT | UTILIZE | CONTENT | TIME/TRAN | CONTENT | 598.00 |
| 2    | 10      | 2       | 173     | 26      | 1       | 4      |

| QUEUE | MAXIMUM | AVERAGE | TOTAL | AVERAGE | ZERO | CURRENT |
| NUMBER | CONTENT | CONTENT | ENTRIES | TIME/TRAN | ENTRIES | CONTENT |
| 3     | 2       | +37     | 5     | 409.97  | 4     | 0       |
| 4     | 2       | +11     | 4     | 251.39  | 2     | 0       |
| 5     | 2       | +37     | 10    | 334.02  | 4     | 2       |
| 6     | 1       | +80     | 2     | 00.00   | 0     | 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 (→).
PERT NETWORK SIMULATION

The metalinguage 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.

<table>
<thead>
<tr>
<th>NET, NETTIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 20 18 28 22</td>
</tr>
<tr>
<td>1 3 40 20 30 100</td>
</tr>
<tr>
<td>1 6 28 18 20 70</td>
</tr>
<tr>
<td>2 4 8 6 7 14</td>
</tr>
<tr>
<td>3 5 0 0 0 0</td>
</tr>
<tr>
<td>3 6 10 7 9 17</td>
</tr>
<tr>
<td>4 5 30 20 25 60</td>
</tr>
<tr>
<td>5 6 20 17 18 31</td>
</tr>
<tr>
<td>5 7 24 18 20 46</td>
</tr>
<tr>
<td>6 8 10 8 10 13</td>
</tr>
<tr>
<td>7 8 12 11 12 13</td>
</tr>
<tr>
<td>8 9 0 0 0 0</td>
</tr>
<tr>
<td>8 10 10 8 8 20</td>
</tr>
<tr>
<td>8 11 8 8 8 8</td>
</tr>
<tr>
<td>9 10 6 5 6 7</td>
</tr>
<tr>
<td>10 11 4 2 3 10</td>
</tr>
<tr>
<td>11 12 4 3 4 5</td>
</tr>
</tbody>
</table>

SIMNET 100

OBSERVATIONS 100

<table>
<thead>
<tr>
<th>REALIZATION TIME</th>
<th>1 2 3 4 5 6 7 8 9 10 11 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVERAGE TIME</td>
<td>0 21 42 29 61 82 86 99 99 108 114 118</td>
</tr>
<tr>
<td>MAXIMUM TIME</td>
<td>0 22 76 32 80 101 104 121 121 130 136 141</td>
</tr>
<tr>
<td>MINIMUM TIME</td>
<td>0 13 10 24 46 65 61 79 79 90 93 97</td>
</tr>
</tbody>
</table>

Winter Simulation Conference 565
While the solution of this problem is of no particular interest, the variety demonstrated in the output 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 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.

CONCLUSION

Our experience with the metalanguage so far has been as a 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 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 is 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


