TELSIM, A USER-ORIENTED LANGUAGE FOR SIMULATING CONTINUOUS SYSTEMS AT A REMOTE TERMINAL

Kenneth J. Busch

Bell Telephone Laboratories, Incorporated
Whippany, New Jersey

INTRODUCTION

TELSIM is a TELetypewriter SIMulation language designed for nonprogrammers and written for a moderate-size time-sharing system. It simulates continuous systems that can be described by a block diagram. Its input language is a natural engineering description of the boxes that comprise the diagram. The time-sharing system is offered commercially by the General Electric Company. This system uses a GE-235 computer shared by a number of remote model 33 or 35 teletypewriters. The programming language is a special version of FORTRAN II.

TELSIM is more than a problem-oriented language; it is also user-oriented. In the past many problem-oriented simulation languages have been written for a batch-processing environment. Many of these have been only a limited help to nonprogrammers in their use of digital computers. Telsim’s advantage lies in freeing the user from memorization of an instruction manual and from the requirements of specialized formats. The machine remembers the rules and helps the man. The man directs the computation; the machine does it. In this way a complex problem can be formulated and solved with Telsim in a relatively short interactive session.

TELSIM CHARACTERISTICS

During the problem definition stage, the boxes are specified one at a time in the order chosen by the user. He types the box number, its contents, and the inputs from other boxes. Each box is arbitrarily assigned a label from 1 to 99. A box may contain a signed or unsigned number, a symbolic constant, a function name, an algebraic operator, or an integration operator. All types are shown in box form in Fig. 1. When the block diagram is completely specified, Telsim produces (and types out upon request) a set of first-order differential equations that represent the dynamic behavior of the box graph. An auxiliary set of output equations is also compiled if required. These equations are valid Fortran expressions.

After seeing the equations, the user may recycle and edit the block diagram, deleting or adding boxes. In fact if he elects to save his input on a private file, he may resume this particular problem any time in the future. Once he is satisfied that his problem has been adequately described, the user directs Telsim to generate a Fortran simulation program that contains the equations for his system. He then requests the computer to compile the program and to run it with precompiled subroutines. The fixed portions are stored in the time-sharing system.
and available on call. The user may also permanently store any simulation program produced by Tel-sim.

During the problem simulation stage, the user directs the computation. He may make any number of simulation runs. The simulation consists of numerically integrating the set of differential equations over a given interval and printing output at a specified period. The dependent variables in this set will be called state variables. The independent variable is designated as "T." Before and after each simulation, the user may list the values of state, change the state initial conditions, and indicate which state variables should be included in the printout. He may also list and change the values of any symbolic constants. The state variables and constants are listed and change the values of any symbolic constants. A request may be made to list the user-assigned names corresponding to these index numbers.

The user can also change the integration control parameters prior to a simulation run. These parameters are the initial and final value of T, the integration step size (\(\Delta T\)), and the print period in units of T.

In addition, the simulation can be made to recycle automatically until a miss tolerance is met or a specified number of iterations has been completed. Both of these values are under the user's control. On each iteration a state initial condition will be incremented in a direction to minimize the miss. This increment(s) is supplied by the user before requesting a run in the optimization mode. If on a given rerun of the simulation the miss increases, the initial condition will be incremented in the opposite direction and the run repeated. Should both directions fail to decrease the miss, the initial condition will be reset to its original value. Hence any iteration may recycle the simulation one or more times.

A successful iteration causes an adjustment of the initial condition of a state variable. An unsuccessful one leaves the initial condition for a given variable unchanged. This process is continued using successive state variables with nonzero increments until the recycling is terminated. If all variables with nonzero state increments have been perturbed and a termination criterion has not been reached, the entire process is repeated. The first state variable adjusted is again changed in the optimal direction (if any) found on the previous attempt. The user can select this optimization mode only if he designated a particular box output as the miss variable during the problem definition phase. Before an optimization run is started, the user indicates how often throughout the run he wishes to be given a choice of continuing or terminating the recycling.

AN ILLUSTRATIVE EXAMPLE

As an elementary example, let us suppose a bombing plane, flying at a constant altitude of 1024 feet with a velocity of 240 ft/sec, is overtaking a surface ship traveling at 80 ft/sec in the same direction as the plane. At what distance astern of the ship should a bomb be released in order to hit the ship if the air resistance is neglected?

Suppose in addition that just as the bomb is released the captain of the ship spots the plane through his telescope. If the plane appears to be very low on the horizon, he is unconcerned and allows his ship to proceed at its current speed. However, if the plane is almost overhead he orders the engine room to accelerate at 1 ft/sec. Let us assume then, that in effect the captain of the ship continuously orders an acceleration equal to the sine of the angle that his telescope makes with the horizon. Under these conditions at what point should the plane drop the bomb?

This problem can be described in block diagram form as shown in Fig. 2. The blocks have been arbitrarily numbered. The user now calls the computer with the "hello" sequence shown in Fig. 3 and re-
requests an instruction manual on how to use the program.

Figure 4 shows the procedure for describing the block diagram to the computer. The sequence “BOX: :=” is supplied by the machine; the user types the rest of the line. In this sample the boxes were described randomly to emphasize this feature. After all the boxes are specified, Telsim asks the user to identify those boxes (a maximum of five including T) whose outputs should be printed during the simulation run. He gives the box number and a title for each output. Finally, if the user wishes to use the optimization mode in the simulation program, he must indicate which output variable is to be optimized. This variable is called the terminal miss. The above procedure is shown in Fig. 5.

Telsim then asks the user if he wishes to see the equations for his problem. In problems that are initially formulated in terms of a block diagram an examination of the equations is an important part of the system analysis. Indeed for those problems described originally as equations (and converted to a block diagram) a comparison of the equations derived by Telsim with the original ones is a valuable check. The user may wish to change the block diagram after inspection of the equations. Telsim provides this opportunity. Once the user is satisfied with his input, he directs Telsim to punch out the program. The equations and a partial listing of the punched program for this example are shown in Figs. 6 and 7, respectively.

This program is then loaded with fixed portions of Telsim and the simulation started. In Fig. 8 the user requested a manual describing how to use the simulation. Figure 9 shows the method of initializing the integrators and specifying constants. In this problem the initial conditions on state variables (ST.IC) have the following meanings:

<table>
<thead>
<tr>
<th>Index No.</th>
<th>Symbol</th>
<th>Definition</th>
<th>IC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X8</td>
<td>Horizontal Velocity of Ship 80 ft/sec</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X36</td>
<td>Vertical Velocity of Bomb 0 ft/sec</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BOMB</td>
<td>Horizontal Pos. of Bomb 0 ft</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>X62</td>
<td>Change in Alt. of Bomb 0 ft</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SHIP</td>
<td>Horizontal Pos. of Ship (Unknown)</td>
<td></td>
</tr>
</tbody>
</table>

"VP" is the only symbol in this problem. It represents the horizontal velocity of the plane, 240 ft/sec. The user’s strategy in finding the solution to this problem is to make the following two runs:

1. A standard simulation run in which the initial position of the ship is directly under the plane (0 ft).
2. An optimization run using the results of the first run.

The above strategy is arbitrary but it was chosen because it demonstrates an efficient use of interaction in order to converge quickly on a solution.

The first run, shown in Fig. 10, will yield the time it takes for the bomb to reach zero altitude. A good approximation to this time is found by hand interpolation between the last positive altitude and first negative altitude points. The second run is then made using this time as the simulation run time and the final separation distance near impact as the initial guess for the initial condition of state variable 5. State increment 5 is now set to 9 feet and an optimization run is made as shown in Fig. 11. The optimization proceeds until the final separation distance at impact is less than 10 feet. Listing the state initial conditions yields the answer that the plane should drop the bomb 1254 feet astern of the ship to score a hit.

TELSIM IMPLEMENTATION

A syntactic form is generated for each box as it is specified by the user. Once the block diagram is completed, a one-pass algorithm is used to compile from this information equations in infix notation with minimal parentheses. Some simplification in the form of the equations is achieved by using tables to control replacement of certain operator combinations. The algorithm permits inclusion of all Fortran functions without defining each function name as an operator. This is accomplished by introducing three special operators for functional notation. In fact this
SHELLO
3 USER NUMBER--WHPO51KJB
SYSTEM--$BFORTRAN
RUN TYPE-- $LOAD INPUT,GRAPH,OF,BOXES

LOAD LIMITS 10463 16857

DO YOU WANT AN INSTRUCTION MANUAL
   ANSWER YES OR NO: YES

TELSIM ACCEPTS A DESCRIPTION OF A BLOCK DIAGRAM.
Each box is inputted as follows:

   BOX: = NO TYPE INPUTS

WHERE
   BOX: = IS TYPED BY TELSIM
   NO = IS A DISTINCT LABEL FOR BOX BEING INPUTTED
   TYPE = IS DESCRIBED BELOW
   INPUTS = ARE LABELS FOR THE LEADS INTO GIVEN BOX OR THE SYMBOL FOR
            TIME, T. INPUTS ARE SEPARATED BY ONE OR MORE SPACES.

A LABEL IS AN INTEGER FROM 1-99.

BOX TYPES ARE SPECIFIED AS FOLLOWS:

+ = ALL INPUTS ARE SUMMED
- = ALL INPUTS ARE NEGATED, THEN SUMMED
* = ALL INPUTS ARE MULTIPLIED TOGETHER
/ = INPUT IS DIVIDED INTO 1
\ = INPUT IS INTEGRATED
CONSTANT = ANY SIGNED OR UNSIGNED NUMBER WITH OR WITHOUT A DECIMAL
            (OR FORTRAN FLOATING NOTATION)
FUNCTION = ANY ONE OF THE FOLLOWING FUNCTIONS: SINF COSF LOGF
           EXPF SGRTF ATANF ABSF INTF MODF SGNF DIMF
           MAXOF MINOF (SEE GREEN CARD FORTRAN MANUAL, PP 14-15
           FOR DETAILS)
SYMBOL = ANY STRING OF FROM 1 TO 5 NON-BLANK ALPHABETIC OR
         NUMERIC CHARACTERS, AT LEAST ONE OF WHICH IS NON-NUMERIC.
         A SYMBOL MAY BE SIGNED OR UNSIGNED.

If the leads into a box are too numerous to be typed on a single line
they may be continued on the next line by typing a $ as the last
non-blank character of the original line.

In editing a box diagram, a box may be deleted as follows:

   BOX: = NO DELETE

WHERE "NO" IS LABEL OF BOX TO BE DELETED.

To terminate block diagram description type:

   BOX: = END

Figure 3. Telsim instruction manual.
TO SAVE INPUT ON PRIVATE FILE OR RERUN USING BLOCK DIAGRAM DESCRIPTION
ON PRIVATE FILE, ANSWER THE FOLLOWING REQUEST:

OPEN FILE NO. 4:

WITH XXXXXX, NAME YYYY

WHERE XXXXXX = YOUR USER NUMBER (FIRST 6 CHARACTERS)
NAME = YOUR FIRST AND LAST INITIALS
YYYY = YOUR PHONE EXT. (4 DIGITS)

IF SYSTEM TYPES "RETRY:=" REPEAT ABOVE INFO.
IF SYSTEM TYPES "FILE NOT THERE, RETRY:=" YOUR OLD INPUT MAY BE LOST.

TELSIM ALLOWS THE USER TO SPECIFY UP TO 5 OUTPUTS.
EACH OUTPUT IS SPECIFIED AS FOLLOWS:

OUT:= NO NAME

WHERE OUT:= IS TYPED BY TELSIM
NO IS THE LABEL OF A BOX TO BE OUTPUTED
NAME IS A SYMBOL USED TO IDENTIFY THE OUTPUT (4 CHARACTERS MAX)

TO DESIGNATE TIME AS AN OUTPUT, TYPE

OUT:= T

TO DELETE THE LAST OUTPUT SPECIFIED, TYPE

OUT:= DELETE

TO TERMINATE OUTPUT SPECIFICATION, TYPE

OUT:= END

*** GOOD LUCK ***

Figure 3. (continued)

scheme can easily be extended to include any function as long as its skeletal form is available for embedding into the target code.

Syntactic Construction

After each input box specification is scanned for errors, the Polish suffix notation can readily be written. The Polish form for each type is given in Table I. All operators are either binary or unary. Unary operators are indicated with a subscript “u.” Note that Hamblin’s early-operator Reverse Polish form is used. In general, this should tend to reduce the total stack space required in the compiling algorithm. Function names, like symbols, are treated as operands. With this convention all functions are represented in terms of a binary and unary null operator (φ, φu respectively) and the comma operator. In tree notation, the function

<name> (arg1, arg2, ..., argn)

has the form indicated in Fig. 12. The null operators control the generation of parentheses as explained later.

The scanning and syntactic construction phase is shown schematically in Fig. 13. A pointer to the Polish information for each box defined by the user is placed in an index table. This box index table (LTB) consists of 99 locations, one for each possible box label. The information describing the Polish suffix form for a box is placed in the box Polish table (LTP). These two tables are used to feed a source stack as explained in the following section.

The final step in preparing the input for compilation is to set up two entries for each “output” box to be printed during the simulation. The output box number is entered into the output box table (LOB). In addition, a pointer to the user’s output name is placed in the output name table (LON) at a location corresponding to the LOB entry.
The State and Auxiliary Equations

The desired equations are produced by a stack compilation technique. Before discussing this algorithm, first consider which equations must be generated. The output of each integrator is a state variable. Let us designate these variables as \( X \)'s. The input to an integrator is \( dX/dT \), where \( T \) is the variable of integration. A set of state equations of the form

\[
dX_i/dT = f_i(X_1, X_2, \ldots, X_n)
\]

\( i = 1, 2, \ldots, n \)

Table I—Polish Suffix for Each Box Type

<table>
<thead>
<tr>
<th>Name</th>
<th>Designation</th>
<th>Polish Suffix Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summar</td>
<td>+</td>
<td>( R_1 ) ( R_2 ) + ( R_3 ) + \ldots + ( R_n ) ( + )</td>
</tr>
<tr>
<td>Negative Sum</td>
<td>-</td>
<td>( R_1 - R_2 - R_3 - \ldots - R_n ) ( - )</td>
</tr>
<tr>
<td>Multiplier</td>
<td>( \ast )</td>
<td>( R_1 \ast R_2 \ast R_3 \ast \ldots \ast R_n ) ( \ast )</td>
</tr>
<tr>
<td>Reciprocal</td>
<td>1/</td>
<td>( R_1 ) ( 1/ )</td>
</tr>
<tr>
<td>Integrator</td>
<td>1/</td>
<td>( R_1 ) ( 1/ )</td>
</tr>
<tr>
<td>Unsigned Value</td>
<td>X.XX</td>
<td>X.XX ( 0 ) ( + ) (with input)</td>
</tr>
<tr>
<td>Positive Value</td>
<td>X.XX</td>
<td>X.XX ( 0 ) ( + ) (no input)</td>
</tr>
<tr>
<td>Negative Value</td>
<td>X.XX</td>
<td>X.XX ( 0 ) ( + ) (with input)</td>
</tr>
<tr>
<td>Symbolic Constant</td>
<td>&lt;symbol&gt;</td>
<td>&lt;symbol&gt; ( R_1 ) ( + ) (with input)</td>
</tr>
<tr>
<td>Function</td>
<td>&lt;name&gt;</td>
<td>&lt;name&gt; ( R_1 ) ( + ) (with one input)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;name&gt; ( R_1 ) ( + ) (no input)</td>
</tr>
</tbody>
</table>

Legend

- \( B \) denotes a box and the subscript notation is as follows:
- \( 1, 2, \ldots, n \) represents \( n \) inputs to a box
- \( i \) represents the only input to a box
- \( j \) represents the given box
- A subscript "u" on an operator denotes a unary operator.
- "\( \ast \)" denotes a null operator.

From the collection of the Computer History Museum (www.computerhistory.org)
must be generated. Here \( i \) refers to successive integrators and is not the box number. There is a total of \( n \) integrators for a given problem. In addition to the state equations, auxiliary equations must be generated for each output variable that is not a state variable. These equations are of the form:

\[
<\text{output name}> = g_i(X_1, X_2, \ldots, X_n)
\]

Although not explicitly shown, the \( f_i \)'s and \( g_i \)'s may (and usually do) involve the values and symbols entered by the user. The equation for the input to an integrator (\( dx_i / dt \)) or an output variable can be constructed by threading backwards through the block diagram. This backward motion is always in opposition to the arrows of the diagram and is terminated on those branches that end in either a value, a symbol, or a state variable. For example, using \( Bn \) to represent the output of box \( n \), the state equations for Fig. 2 are:

\[
\begin{align*}
\frac{d(X36)}{dt} &= -32.16 \\
\frac{d(X62)}{dt} &= X36 \\
\frac{d(BOMB)}{dt} &= V_p \\
\frac{d(X8)}{dt} &= B50 &= \sin(B13) \\
&= \sin(\text{atan}(B28)) \\
&= \sin(\text{atan}(1024*B5)) \\
&= \sin(\text{atan}(1024*(1./B33))) \\
&= \sin(\text{atan}(1024*(1./(B73 + B54)))) \\
&= \sin(\text{atan}(1024*(1./(\text{SHIP} + (-BOMB))))) \\
\frac{d(\text{SHIP})}{dt} &= X8
\end{align*}
\]

A similar relation can be written for the auxiliary equation needed to compute \( \text{DIST} \). Note that if the user does not assign an output name to a state variable, the compiler must provide a distinct symbol to represent it (\( Xk \) was chosen, where \( k \) is the box number).

The Compiler Algorithm

To generate these equations the box index and box Polish tables feed a source stack. This procedure is designed so that as the source stack is popped the Polish suffix for an equation is scanned from right to left. Scanning in this order permits identification of required infix parentheses in one pass. In compiling a state equation the source stack is primed with the Polish for the corresponding integrator box (i.e., \( DX_i B \)). The first token entered is the left-hand token of the Polish for the box (\( DX_i \)). The source is then popped on a last-in first-out basis (LIFO). Tokens representing nonintegrator boxes cause the source stack to be replenished by the corresponding box Polish form, left to right. A box token for an integrator box causes the state symbol (\( Xk \) or user output name if assigned) to be pushed down on the source stack.

Tokens not representing boxes or operators will be called "symbols" for the purpose of this discussion. Operators popped from the source stack are held temporarily in an operator stack. When symbol tokens are popped, they are entered directly into the target stack, preceded by any necessary right parentheses, and followed by an operator from the opera-
$LOAD SAMPLE, USING, TELSIM, SYSTEM

LOAD LIMITS 11552 16132

DO YOU WANT THE INSTRUCTION MANUAL.
ANSWER YES OR NO: YES

TELSIM WILL SIMULATE YOUR BLOCK DIAGRAM SYSTEM. THE OUTPUT OF EACH INTEGRATOR (IYS) IS A STATE VARIABLE NAMED: XBB -- WHERE BB IS BOX NUMBER, OR ZZZ -- WHERE ZZZ IS YOUR OUTPUT NAME. IF YOU SPECIFIED SYMBOLIC CONSTANTS, EACH CONSTANT IS IDENTIFIED BY THE SYMBOL YOU GAVE.

LISTING NAMES AND VALUES: BEFORE AND AFTER A RUN, YOU CAN ASK FOR A LIST OF THE STATE VARIABLES AND SYMBOLS. IN LISTING NAMES AND VALUES AN INDEX NUMBER IS ALSO PRINTED. THIS INDEX NUMBER IS THE KEY THAT ASSOCIATES A VALUE WITH A STATE VARIABLE OR SYMBOL.

CHANGING VALUES: BEFORE A RUN, YOU CAN CHANGE ANY OF THE STATE VARIABLES OR SYMBOLS. AFTER A REQUEST FOR A CHANGE YOU MUST SUPPLY THE INDEX NO. IDENTIFYING THE ITEM TO BE CHANGED, A SPACE OR COMMA, AND THE NEW VALUE. AS MANY VALUES AS DESIRED MAY BE ENTERED ON SUCCEEDING LINES INCLUDING RETYPING A BAD ENTRY. WHEN FINISHED TYPE: 0 0

ASSOCIATED WITH EACH STATE VARIABLE ARE THE KEY WORDS:

ST.PRT -- PRINT FLAGS WHICH YOU CAN SET TO 1 IF YOU WANT THE INTEGRATOR VALUE PRINTED WITH THE SIMULATION OUTPUT.

SW,PRT -- SWITCH TO CONTROL PRINTING OF ABOVE INTEGRATORS. STATES ARE: OFF, NORMAL, INITIAL(ONLY), FINAL(ONLY).

ST,IC -- INITIALS CONDITION (I.E. THE VALUE) OF THE INTEGRATOR AT THE START OF THE SIMULATION.

ST,INC -- IF YOU SPECIFIED A TERMINAL MISS VARIABLE (TM) IN TELBOX, YOU CAN CHOOSE TO MAKE AN OPTIMIZATION (OPT) RUN IN WHICH THE SIMULATION WILL AUTOMATICALLY RECYCLE, AND THE ICS ARE SYSTEMATICALLY ADJUSTED UNTIL THE TM VALUE IS LESS THAN THE TOLERANCE YOU SPECIFY OR THE NUMBER OF ITERATIONS EQUALS THE MAXIMUM YOU SPECIFY.

ALL OF THE ABOVE CAN BE LISTED (INCLUDING NAMES), AND CHANGED AS DESCRIBED ABOVE. THE INDEX NO. OF THE CORRESPONDING STATE VARIABLE IS USED WHEN CHANGES ARE MADE.

Figure 8. Simulation instruction manual.
TELsim, a User-oriented Language

CONTROL OF INTEGRATION: You must supply (and can change) the initial and final values of t, the variable of integration. In addition, an integration step size and print interval in units of t must be given. If a printout is desired only at the beginning and end of a simulation make the print interval equal to or greater than the total change in t. When making an optimization run you must supply (and can change) the tolerance on the terminal miss, no. of iterations, and the no. of iter. cycles before inquiry. One iter. cycle consists of an iteration for each state variable whose st.inc is nonzero. After the no. of iter. cycles specified you will have the choice of continuing or terminating the optimization.

How to type numbers:

Index no. -- an unsigned integer, no decimal point

Value -- a floating point number of the following form:
+1.2345678, 123.45678E-2, +1.2345678E+24

Where the signs, when +, can be omitted, the decimal may occur anywhere, the number of digits may be less, and the exponent eswh may be omitted but when present represents the power of ten which multiplies the value.

St.prt flags -- an integer, 0 or 1

Leave a space between each number, hit return key when finished.

***************
ON COLD START (FIRST RUN) ALL ST.IC, SYMBOLS, ST.PRT, AND ST.INC HAVE BEEN CLEARED, I.E. SET TO ZERO. YOU MAY CHANGE THEM AS DESIRED.

GOOD LUCK***************

Figure 8. (continued)
TYPE VALUES FOR: INITIAL T, FINAL T, INTEGRATION STEP, PRINT INTERVAL
:t=0 10 .5
QUES: EXPLAIN LIST CHANGE RUN END
QUES: NAMES VALUES ST.PRT NONE
QUES: SYMBOLS STATE ST.IC ST.INC NONE

1 X8 2 X36 3 BOMB 4 X62 5 SHIP
QUES: EXPLAIN LIST CHANGE RUN END
QUES: VALUES ST.PRT SW.PRT CONTROL NONE
QUES: SYMBOLS STATE ST.IC ST.INC NONE

1 VP
QUES: EXPLAIN LIST CHANGE RUN END
QUES: VALUES ST.PRT SW.PRT CONTROL NONE
QUES: SYMBOLS STATE ST.IC ST.INC NONE

Figure 9. Method of identifying and changing variables and symbols.

QUES: EXPLAIN LIST CHANGE RUN END
QUES: STD OPT NONE

<table>
<thead>
<tr>
<th>T</th>
<th>SHIP</th>
<th>BOMB</th>
<th>ALT</th>
<th>DIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000000E-01</td>
<td>0.000000E-01</td>
<td>0.000000E-01</td>
<td>1.024000E+03</td>
<td>1.024000E+03</td>
</tr>
<tr>
<td>1.000000E+00</td>
<td>7.966702E+01</td>
<td>2.400000E+02</td>
<td>1.007920E+03</td>
<td>1.020953E+03</td>
</tr>
<tr>
<td>2.000000E+00</td>
<td>1.585492E+02</td>
<td>4.500000E+02</td>
<td>9.396000E+02</td>
<td>1.021149E+03</td>
</tr>
<tr>
<td>3.000000E+00</td>
<td>2.360816E+02</td>
<td>7.200000E+02</td>
<td>8.792000E+02</td>
<td>1.033505E+03</td>
</tr>
<tr>
<td>4.000000E+00</td>
<td>3.129833E+02</td>
<td>9.600000E+02</td>
<td>8.057000E+02</td>
<td>1.033925E+03</td>
</tr>
<tr>
<td>5.000000E+00</td>
<td>3.898839E+02</td>
<td>1.200000E+03</td>
<td>7.697000E+02</td>
<td>1.034215E+03</td>
</tr>
<tr>
<td>6.000000E+00</td>
<td>4.660002E+02</td>
<td>1.440000E+03</td>
<td>7.372000E+02</td>
<td>1.048235E+03</td>
</tr>
<tr>
<td>7.000000E+00</td>
<td>5.385323E+02</td>
<td>1.680000E+03</td>
<td>6.960000E+02</td>
<td>1.072638E+03</td>
</tr>
<tr>
<td>8.000000E+00</td>
<td>6.123362E+02</td>
<td>1.920000E+03</td>
<td>6.570000E+02</td>
<td>1.150405E+03</td>
</tr>
<tr>
<td>9.000000E+00</td>
<td>6.854310E+02</td>
<td>2.160000E+03</td>
<td>6.200000E+02</td>
<td>1.307504E+03</td>
</tr>
<tr>
<td>1.000000E+01</td>
<td>7.581592E+02</td>
<td>2.400000E+03</td>
<td>5.840000E+02</td>
<td>1.450525E+03</td>
</tr>
</tbody>
</table>

QUES: EXPLAIN LIST CHANGE RUN END
QUES: VALUES ST.PRT SW.PRT CONTROL NONE
QUES: SYMBOLS STATE ST.IC ST.INC NONE

Figure 10. Standard simulation run.
TELSIM, A USER-ORIENTED LANGUAGE

QUES: EXPLAIN LIST CHANGE RUN END
1=RUN
QUES: STD OPT NONE
1=OPT

TYPE VALUES FOR:

TOLERANCE ON T. TERMINAL MISS, NO. OF ITERATIONS, NO. OF
ITER. CYCLES BEFORE INQUIRY.

T
0.000000E-01 1.305000E+03 0.000000E-01 1.024000E+03 1.661156E+03
7.960000E+00 1.970540E+03 1.915200E+03 1.916218E-02 9.385381E+01
T
0.000000E-01 1.517000E+03 0.000000E-01 1.024000E+03 1.6618252E+03
7.950000E+00 1.979444E+03 1.915200E+03 1.916218E-02 6.424410E+01
T
0.000000E-01 1.290000E+03 0.000000E-01 1.024000E+03 1.6618252E+03
7.950000E+00 1.961636E+03 1.915200E+03 1.916218E-02 4.643556E+01
T
0.000000E-01 1.290000E+03 0.000000E-01 1.024000E+03 1.6618252E+03
7.950000E+00 1.957271E+03 1.915200E+03 1.916218E-02 3.755314E+01
T
0.000000E-01 1.251000E+03 0.000000E-01 1.024000E+03 1.639891E+03
7.950000E+00 1.943527E+03 1.915200E+03 1.916218E-02 2.862715E+01
T
0.000000E-01 1.272000E+03 0.000000E-01 1.024000E+03 1.639891E+03
7.950000E+00 1.934923E+03 1.915200E+03 1.916218E-02 1.972295E+01

DO YOU WANT TO CONTINUE

ANSWER YES OR NO := YES

T
0.000000E-01 1.263000E+03 0.000000E-01 1.024000E+03 1.625960E+03
7.950000E+00 1.926019E+03 1.915200E+03 1.916218E-02 1.081875E+01
T
0.000000E-01 1.254000E+03 0.000000E-01 1.024000E+03 1.618979E+03
7.950000E+00 1.917114E+03 1.915200E+03 1.916218E-02 1.914386E+00
QUES: EXPLAIN LIST CHANGE RUN END
1=LIST
QUES: NAMES VALUES ST.PRT NONE
1=VALUES
QUES: SYMBOLS STATE ST.IC ST.INC NONE
1=ST.IC

1 5.000000E+01 2 0.0000000E-01 3 0.0000000E-01 4 0.0000000E-01
5 1.2540000E+03
QUES: EXPLAIN LIST CHANGE RUN END
1=LIST
QUES: NAMES VALUES ST.PRT NONE
1=VALUES
QUES: SYMBOLS STATE ST.IC ST.INC NONE
1=STATE

1 5.6732114E+01 2 -2.5663680E+02 3 1.9152000E+03 4 -1.0239808E+03
5 1.9171145E+03
QUES: EXPLAIN LIST CHANGE RUN END
1=END

END OF RUN. GOOD-BYE.
*STOP 0 AT 0755

ELAPSED TIME IN HUNDREDTHS OF HOURS 078

Figure 11. Optimization run.
right-hand buds is found from Table II. For operators added to left buds, Table III is used. In these tables the added operator is the current operator; its parent in the tree ("*" in this case) is the indirect operator. The table look-up indicates parentheses are required. This is indicated as:

\[0,0\]

where the number on the left stands for a left parenthesis count; the right number, for a right parenthesis. The counts on the starting node are zero. The next token is then taken from the Polish string and added to the first available right bud:

\[0,0\]

As long as operators are being added the structure continues to grow. Each new operator is added to the lowest bud with a preference given to right buds. An operand, on the other hand, is a leaf of the tree. These leaves trigger the pruning of both operands and their associated branches. In pruning right branches the number of right parentheses indicated above the last operator is written into the target stack. The operand which triggered the pruning is written next, followed by the operator. The target stack resulting from the pruning is shown alongside the tree. The right side of this stack is the top of a LIFO stack. After pruning the right branch, the binary operator is marked with an L to indicate it has only a left bud remaining. This is necessary in order to prevent the operator from being written a second time when its left branch is pruned. The next token

Table II—Right-Hand Table

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Indirect Op</th>
<th>Current Op</th>
<th>(l/u)</th>
<th>(u)</th>
<th>(\phi)</th>
<th>(\beta)</th>
<th>(\gamma)</th>
<th>(\theta)</th>
<th>(\delta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(l/u)</td>
<td>(l/u)</td>
<td>(-1)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
</tr>
<tr>
<td>2</td>
<td>(\gamma)</td>
<td>(\gamma)</td>
<td>(0)</td>
<td>(3)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
</tr>
<tr>
<td>3</td>
<td>(\Phi)</td>
<td>(\Phi)</td>
<td>(-1)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
</tr>
<tr>
<td>4</td>
<td>(\phi)</td>
<td>(\phi)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
</tr>
<tr>
<td>5</td>
<td>(\epsilon)</td>
<td>(\epsilon)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
</tr>
</tbody>
</table>

Legend

NA: not applicable
0: no parentheses required
-1: parentheses required
>0: pointer to operator to replace indirect operator

Table III—Left-Hand Table

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Indirect Op</th>
<th>Current Op</th>
<th>(l/u)</th>
<th>(u)</th>
<th>(\phi)</th>
<th>(\beta)</th>
<th>(\gamma)</th>
<th>(\theta)</th>
<th>(\delta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(l/u)</td>
<td>(l/u)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
</tr>
<tr>
<td>2</td>
<td>(\gamma)</td>
<td>(\gamma)</td>
<td>(-1)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
</tr>
<tr>
<td>3</td>
<td>(\Phi)</td>
<td>(\Phi)</td>
<td>(-1)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
</tr>
<tr>
<td>4</td>
<td>(\phi)</td>
<td>(\phi)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
</tr>
<tr>
<td>5</td>
<td>(\epsilon)</td>
<td>(\epsilon)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
</tr>
</tbody>
</table>

Legend

NA: not applicable
0: no parentheses required
-1: parentheses required
>0: pointer to operator to replace indirect operator

Note: When current operator is unary, the indirect operator is its "right" parent.

Figure 12. Function tree.
C is added. In this case no right bud is available, so it is added to the lowest left bud of the structure.

In so doing, another branch is completed and pruning starts again. For a left branch the operand is written followed by the number of left parentheses indicated by the left count above the connecting node. The node is then discarded.

This also completes the right branch of the `*` node and causes it to be pruned and marked as described above. Now the structure grows again with the addition of a `+` node to the remaining left bud.

In this case the parenthesis count is determined from Table III. The addition of the last tokens, in turn, completes the tree and causes pruning of all the nodes. The target stack will then contain:

\[ )D - C(\star)B + A( \]

Unloading this on a LIFO basis gives the original terms in correct infix notation.

This simple example did not involve two very significant aspects in compiling infix equations. First, parentheses will accumulate as operators are nested within expressions. To accommodate this, the right-hand count is cumulative while the structure grows along successive right branches. This count is reset to zero when the growth changes to a left branch. Thereupon, the left count is accumulated as the structure continues to grow downward to the left.

Second, if the expression contains unary operators, slightly different rules are needed to accommodate them. If a unary term requires parentheses, for example in the expression \( A*(-B) \), the left parenthesis is always adjacent to the operator. No amount of nesting of terms within an expression will ever break this bond. For a binary operator it would be to the left of the left-hand symbol. The corresponding right parenthesis for a unary operator, however, may be considerably removed from the operator. This occurs if the unary node is a parent of other nodes as in the expression \( A*(-C*D/E) \).

This suggests that right parenthesis information for unary operators might be handled in the same fashion as for binary. But the left parenthesis count should be treated differently. If the unary operator requires parentheses, the right counter is incremented and the operator itself is marked with a P. This mark will indicate that when the operator is written into the target stack, it should be followed immediately by a left parenthesis. With this convention unary nodes need carry only one count, the right parenthesis.

The expressions given above that involve unary operators should suggest that such operators may lead to situations where operator replacement is desirable. For example \( A+(-B) \) is better written as...
**Data Structure**

A good data structure is important if compilation is to be fast. The structure used in Telsim permits quick identification of the three types of tokens: boxes, symbols (or function names or values), and operators. The entries in the box Polish table are actually pointers to the primitive information stored in a text table. The text table is appended to the box index table starting at location 100. Symbols and values are then distinguishable from boxes since the latter have pointers whose magnitude is less than 100. To separate this operand class from the operators, the box and symbol pointers are made negative. In processing the operators, the algorithm also needs to identify its type: either binary or unary. This is accomplished by constructing the operator entries to point to an operator table (LTO). The operator table entry then references the primitive in the text table. A positive pointer in the operator table indicates that the operator is binary; negative is unary. Furthermore, the operator table is ordered to

---

**Table IV—Compilation Example**

<table>
<thead>
<tr>
<th>Source Token</th>
<th>Counter</th>
<th>Stack Contents</th>
<th>Source</th>
<th>Token</th>
<th>Target</th>
<th>Stack Contents</th>
<th>Stack Contents After Each Token Is Analyzed</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Prime)</td>
<td>Op</td>
<td>L=</td>
<td>Target</td>
<td>0</td>
<td>0</td>
<td>L=</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>0</td>
<td>0</td>
<td>Op</td>
<td>Target</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>0</td>
<td>0</td>
<td>Op</td>
<td>Target</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0</td>
<td>Op</td>
<td>Target</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>0</td>
<td>0</td>
<td>Op</td>
<td>Target</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>Op</td>
<td>Target</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Op</td>
<td>Target</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>21T</td>
<td>0</td>
<td>0</td>
<td>Op</td>
<td>L=</td>
<td>Target</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>0</td>
<td>Op</td>
<td>L=</td>
<td>Target</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Sin</td>
<td>0</td>
<td>0</td>
<td>Op</td>
<td>L=</td>
<td>Target</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
<td>Op</td>
<td>L=</td>
<td>Target</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

---

A—B. In this case the unary term is a right-hand term of the "+" operator. Note that Table II indeed indicates this replacement. A similar situation may prevail when the unary term is a left-hand term of a binary operator as in A + (−B * C). Here (−B) is the left term of "*". But to ascertain that replacement is possible, one must consider not the immediate parent (the "*" in this case) but rather the closest "right" parent (the "+"). The left-hand table, Table III, shows this special consideration for these unary operators. Now consider the following equation:

Infix: \( X = \sin((T - T_I) \cdot 2 \pi * F) \)

Suffix: \( X \sin T T_I - u + 2 \pi * F * \phi_u \phi = \)

The unary minus operator was used, since the block diagram corresponding to this equation would cause it to be present. The dynamic growth and pruning of the tree that represents this equation is shown stage by stage in Fig. 15. The source and target stacks are shown at each stage.

The reader may want to follow the flow indicated in the successive parts in Fig. 15. Note that the nodes "+ −u" were placed by "−" in Figs. 15(c) and (d). The target stack in (g) is completed when "X" is added as the last item. Unloading the target stack on a LIFO basis and suppressing the null operators (squeezing out blanks) gives the desired infix equation, left to right.

The complete algorithm is flow-chARTed in Fig. 16. Before compilation of each equation, the indirect operator (OP) and left parenthesis counter (Lparen) stacks are primed with "L= =" and "O", respectively. The purpose of this priming is to provide an indirect operator for the first operator in the source stack (always =) that gives an initial count of 0, 0. Note that the marking of nodes is indicated by preceding the operator with the marking symbol. Let's step through the algorithm using the preceding suffix equation. This is shown in Table IV. The first line represents the priming operation. Succeeding lines give the results as successive tokens are taken from the source and processed. Popping the operator stack in the algorithm also implies discarding the Lparen entry. The last entry in Table IV shows the desired infix equation in the target stack.
Figure 15. Dynamic tree structure.
PRINING: "L*" TO OP STK,"O" TO LPAREN STK

NOTATION: OP AND LPARAREN ARE TOPS OF DUAL STACKS.
OP1 IS INDIRECT OPERATOR (LAST ITEM IN OP).
OP2 IS CURRENT (OPERATOR) TOKEN.
P AND L STAND FOR "MARKED" OPERATORS.

NOTE: POPPING OP STK. ALSO
IMPLIES POPPING LPARAREN
STACK.

Figure 16. The compiler algorithm.

Figure 17. Telsim data structure.
correspond to the entries of the right- and left-hand tables used in the algorithm. The details of this structure are shown in Fig. 17.

Referring back to Fig. 14, we see that during the generation of the suffix equation the box pointers are numbers in the range of $-1$ to $-99$. All symbol pointers are less than this, and operator pointers are positive. In the next phase, the algorithm proper, all symbol pointers are set positive when fed into the target stack. In pruning operators, the magnitude of the pointer in the operator table is placed in the target stack. This means that after an equation is compiled it will be a list of pointers to the primitives in the text table. Special conventions within the text table have been designed as seen in Fig. 17 to distinguish user symbols and output names from each other and from numbers and compiler-generated symbols. This permits symbol prefixing to be done on the fly in punching out the final program. No prefixing is done when they are printed for the user’s inspection. These details will not be discussed here.

Before leaving this subject it should be mentioned that entries for integrators in the box index table and output box table are set negative. Negative pointers in the box index table indicate when a state variable is encountered in popping the source stack. Output integrator boxes also are marked, since no auxiliary equation will be needed for them.

Structure of the Simulation Program

The state and auxiliary equations are embedded by the compiler in a segment of a Fortran program. The complete simulation program consists of this variable segment together with a fixed segment and precompiled subroutines. The structure of the simulation program is relatively straightforward and is shown in Fig. 18.

The variable segment contains the necessary bookkeeping (common, equivalence, etc.) that changes from problem to problem. These instructions are arranged so that, together with flags that give the number of integrators and symbols, the common layout is known by all subroutines. Routines such as those that print and read the contents of various arrays must be able to communicate with this common data region. Following the “bookkeeping,” a statement transfers control to the section that plays the “question game” with the user. The user is given a choice of actions for the program to carry out, such as list or change. Based upon his response, the next choice is funneled to a subset of such meaningful descriptors as names or values. This “funneling or steering” is a technique that has been employed in control of graphical routines. In this instance the user employs a light pen to point at the appropriate control word displayed on the cathode ray tube. Programming this interactive dialogue was facilitated by using a general purpose message routine.

The variable segment is completed by embedding the state and auxiliary equations immediately after the statement transferring control to the “question game.” The equations are executed, as indicated by the dotted lines in Fig. 18, by transferring out of (and then back to) the integration and print sections of the fixed segment. The integration method presently used is 4th order Runge-Kutta.

The unusual structure of this program deserves comment. The GE-235 computer has a machine cycle of six microseconds and no floating-point hardware. This together with the time-slicing required to time-share the system tends to make large blocks of computation relatively slow. The simulation program was designed to minimize transfers to subroutines once the integration computations were underway. In effect, the integration and equations which compute the derivatives are the main program in Telsim. This corresponds to turning the usual structure of a simulation program “inside-out” in order to make it as fast-running as possible.
SUMMARY

This version of Telsim is an initial effort to provide user-oriented languages for control system analysis and synthesis. A user-oriented language must be responsive. There should be a man-machine dialogue during both problem definition and solution. Telsim attempts to do this.

Telsim is a compiler not an interpreter. It is a language programmed in Fortran that produces Fortran code designed for efficient simulation. Telsim uses a one-pass table-driven compiler algorithm to produce the state (differential) and output equations. This is a departure from the existing "sequencing" method used in current simulation languages. "Sequencing" or "sorting" of boxes requires that an ordered list of Fortran expressions, one for each box, be generated such that the last expressions give the values for each of the derivatives. As a consequence of this new method there is no need to apportion storage space among the various types of boxes as done in the past. The user can specify any number of the different types as long as storage is not exhausted. He can then view the equations in meaningful form and thereby check his block diagram. These equations are valid Fortran expressions.* Telsim compiles efficient and fast-running Fortran code for either simple or complex diagrams. This results in a program comparable to one hand-coded by a good programmer.

By using the procedures presented in this paper, Telsim can be extended to allow the user to define and name functions by a block diagram composed of primitive boxes. In addition, Telsim has advanced language features such as symbolic labeling of constants, use of Fortran functions (including nesting), and relative freedom in input formats. The simulation program, itself, has an optimization mode that permits automatic adjustment of initial conditions in solving boundary value problems.

This effort has suggested a number of improvements. Three things will be done immediately. Little more can be, since this version of Telsim has taxed the available time-sharing system to its capacity. First, the optimization ability will be extended to include automatic adjustment of symbolic constants in an analogous fashion to the initial condition feature. Included with symbolic constants are the initial and final values of T, the variable of integration.

Second, Telsim will be improved to allow the user to specify a termination function (akin to the terminal miss variable) to control the final time in a simulation run. And finally, the user will be provided a choice of the method of numerical integration to be used during the simulation run.

There are many more desirable features that one would like in a general simulation language. During problem definition, the following should be considered:

1. The ability to accept equations as the specification for the whole system or part of a block diagram.
2. The ability to simulate composite systems with both continuous and discrete portions. (For discrete system simulation the reader is referred to the BLODI language.)
3. A larger subset of primitive boxes to specify nonlinear elements, transfer functions, random noise generator, and tables.
4. A simple method for the definition of new functional boxes thereby providing system growth.
5. Superblock specification by the user of those portions of a system that are repeated a number of times.
7. Diagnostics during the compilation algorithm and efficient handling of subexpressions found to be common.

During the simulation phase the user should have:

1. The ability to select from a set of optimization procedures.
2. The ability to specify events and criterion information.
3. Automatic determination of integration methods and step size.
4. More freedom in selecting printout (choosing any variable and nonuniform printing interval during a run).
5. Graphical output.

Needless to say implementation of these features will have to await the arrival of a larger time-sharing system. However, exploratory programming and development along these avenues will be started in a batch-processing environment.
ACKNOWLEDGMENT

The skillful programming done by Misses Mary Lou Flynn and Ruth L. Salmon, and Messrs. Robert E. Downes and Gardner C. Patton has made this version of Telsim possible. The helpful suggestions and comments by Dr. W. C. Ridgway, III during the preparation of this paper are appreciated.

REFERENCES
