INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

SC

An Implementation Of The Discrete Portion Of The Combined Digital Simulation Language GEST

By

Ladislav J. Bleha

Submitted To The School Of Graduate Studies
In Partial Fulfillment For The Requirements Of
The Degree Of Master Of Applied Science.

Department Of Electrical Engineering
Faculty Of Science And Engineering
University Of Ottawa
Ottawa, Ontario

April, 1977

ACKNOWLEDGEMENTS

I wish to express my appreciatiation to the entire faculty and staff of the Department of Computer Science and Electrical Engineering, The University of Ottawa.

In particular, to Dr. Tuncer Oren, as my supervisor, for his patience, counseling and everpresent assistance, invaluable suggestions and encouragement during the course of this project.

To Dr. Louis Birta for his advice during the development of the translator.

Finally I must acknowledge that without the moral support of my family I would not be able to complete this work.

Table Of Contents

Extended Table Of Contents :

CHAPTER 7

CHAPTER 8

ABSTRACT

An implementation of Discrete Portion of a combined digital simulation language, called the General System Theory Implementor, referred to by the acronym GEST, is presented.

A GEST translator, accepting a GEST source program generates the corresponding FORTRAN IV program and executes it.

This implementation handles a subset of GEST statements with a syntax defined for the Discrete Portion of the simulation language.

A general review of simulations and simulation languages with a respect to simulation requirements of large systems is given.

The implementation of this translator is presented with recommendations for the future implementation and enhancements based on the experience gained during this study.

CHAPTER 1

1.0 Concept of simulation languages

Simulation programming languages are designed to assist analysts in the design, programming, and analysis of simulation models. Ten particularly important simulation programming language features are identified: Modeling a system's static state, modeling system dynamics, statistical sampling, data collection, analysis and display, monitoring and debugging, initialization and language usability. The future development of simulation programming languages will be dependent on advances in the fields of computer languages, computer graphics and time sharing.

To be able to understand the specific requirements which are expected to be covered by the simulation programming languages the following will define what is understood by a simulation.

-SIMULATION (first definition):

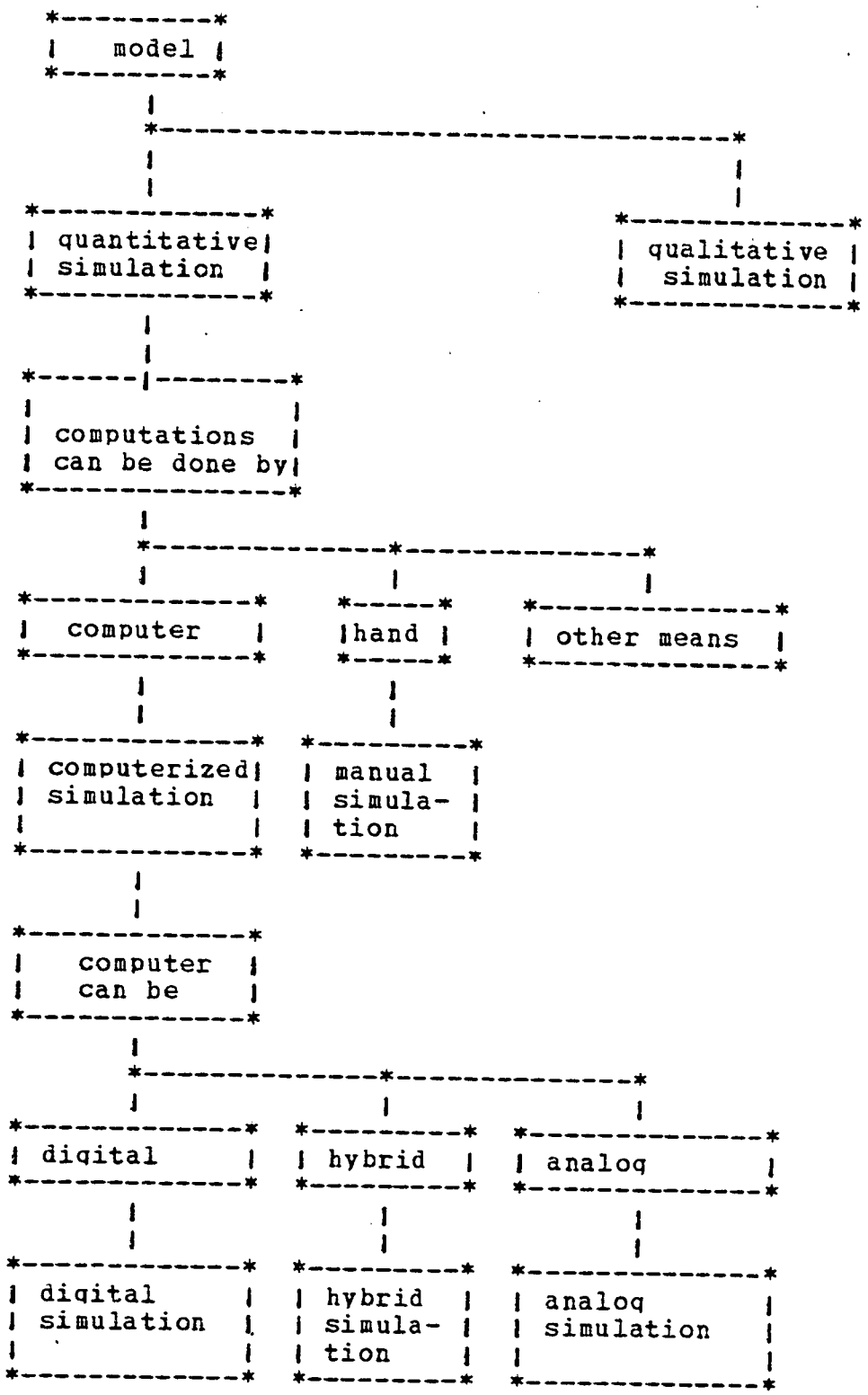
Is a performance of experiments upon a model of a system for the purpose of drawing inferences concerning the properties of the system under investigation from the behavior of its model.

-SIMULATION (second definition):

Is the replacement of a given system by a substitute system, or simulator, which responds to stimuli in a similar way and hence has the salient characteristics of the given system.

Model may be defined in the following way: To an observer O, an object A is a model of an object B to the extent that O can use A to answer questions that interest him about B.

Model can be:


model
quantitative simulation
qualitative simulation
computations can be done by
computer
hand
other means
computerized simulation
manual simula- tion
computer can be
digital
hybrid
analog
digital simulation
hybrid simula- tion
analog simulation

A simulation programming language (SPL) is a problem oriented language with special features. Simulation being a problem- solving activity with its own needs, programming languages have been written to make special features available to simulation programmers at a problem oriented language level.

Historically, Simulation Programming Languages have developed gradually from assembly language programs with special features, through extended commercially available Problem Oriented Languages, to sophisticated, special-purpose Simulation Programming Languages.

1.1 Principal features of simulation languages

FEATURES of the simulation programming languages that are not normally found in most other programming languages have the following characteristics:

1) They provide data representation that permit efficient and straighforward modeling.

2) They permit the reproduction of dynamics within a modeled system.

3) They are oriented to the study of stochastic systems, that is they contain procedures for the generation and analysis of random variables, and time series.

The first feature calls for data structures more elaborate than the typical unsubscripted-subscripted variable organisations found in the FORTRAN programming language.

DATA STRUCTURES must be richer in two ways: They must be capable of complex organisation, as in tree structures, lists, and sets and they must be able to store varieties of data, such as numbers, both integer and real, double precision and complex, character strings of both fixed and data structure references. As data structures exist only so that they can be manipulated, statements must be available that (i) assists in initializing a

system data base (as we can call the collection of data that describe a system); (ii) permit manipulation on the data, such as adding elements, changing data values, altering data structures, and monitoring data flows; and (iii) enable communication between the modeler and the data.

The second feature deals with modelling formalism, both definitional and executable, that permit the simulation of dynamic, interactive systems. Statements that deal with time dependent descriptions of changes in system state, and mechanisms that organize the execution of various system-state change programs so that the dynamics of a system are represented correctly, are an integral part of every Simulation Programming Language.

The third feature stems from the fact that the whole world is changing in a Stochastic manner. Because things do not happen regularly and deterministically, but randomly and with variations, procedures are needed to generate pseudorandom variates from different statistical distributions and from empirical sampling distributions, so the real world variability can be presented.

The three most frequently cited Reasons For Having Simulation Programming Languages are (i) programming convenience, (ii) concept articulation and (iii) communication and documentation.

Every Simulation Programming Language has a number of its own special simulation-oriented features which distinguish that simulation programming language from other simulation languages. As certain features of simulation programming languages become widely used their definitions will evolve and become more standardized. This leads us to establish the basic simulation programming language (SPL) requirements.

An SPL must provide:

- Facilities for performing experiments, for presenting experimental results, for prescribing experimental conditions etc. (To study the behavior.)

- Facilities for describing the structure of a great variety of systems. Representations are needed for describing the objects found in systems, their qualities and properties, and the relationships between them. (A mathematical model of a system).

- Facilities for describing dynamic relationships within systems and for operating the system representation in such a way that the dynamic aspects of system behavior are reproduced. (Operates over time.)

The static structure of a simulation model is a time

-independent frame work within which system states are defined. System states are possible configurations in which a system can be. Dynamic systems act and interact within a static data structure, changing data values and, thereby, changing system states. A system is an interacting collection of objects in a closed environment, the boundaries of which are clearly stated. Every system contains indentifiable classes of objects, which can vary in number, have varying numbers of identifying characteristics, and are related to one another and to the environment in changeable although prescribed ways.

Simulation programming languages may be divided into several groups by the following criteria:

1) Application area: General simulation languages, special simulation languages.

2) Type of the Computer: Digital or hybrid.

3) Time Set of the model: Discrete, continuous or combined.

4) Time Structure: Continuous or discrete (activity oriented(time 'slicing'), event oriented (exogenous, endogenous, process oriented).

5) Type of Access to the computer: Interactive (on-line) or non - interactive (off-line).

6) Type of the Language Processor:compiler, translator, macro - generator or interpreter.

When To Use Simulations ?

Criteria, for deciding on the use of a computerised simulation study are AVAILABILITY, CONVENIENCE and ECONOMY or COST.

- Availability: A real system may not exist, for example hardware simulation in developing software for the next generation of computers, or the real system is not available (simulation of lunar landing).

- Convenience: Experimentation on the real system may be impractical (simulation of social systems, business simulation). Or for example analytical solutions can be too difficult to find

- ECONOMY or COST: The cost of experimentation on the model including the cost of building and maintaining the model should be less than the cost of experimenting on the real system for the total number of experimentations required to get the necessary information (assuming experimentation on the real system is possible).

1.2 Need for a new simulation language.

The general features of simulation languages are not sufficient for modelling of large scale systems. To meet the following requirements the new simulation language GEST has been developed:

1) To handle large scale systems GEST permits the definition of a given system as a set of component systems having interactions. The modelling "independence" of the component systems is very useful because more then one team can work and develop the model at the same time separately. The user is asked to distinguish input, state variables and state transition functions of each component system.

2) The interfaces between several mathematical models of a system may be expressed by a description how all component systems are interrelated if these are available one can easily take already existing simulation models with or even without modifying them and develop mathematical models of a new system. The large scale system will consists all these simulation models and it is not always necessary to start from scratch. In GEST this facility is given by the Coupling Declaration.

3) Some of the component systems may operate over discrete and/or continuous time sets. Because of the impossibility of restricting the simulation model and its component systems on

discrete or continuous simulations only, there is a need to have combined discrete and continuous time digital simulation languages.

4) One needs to be able to describe how component systems and their interrelationships can vary with time. This could be of particular importance in a simulation study of long range international politics or financial models.

5) The parallel execution of programs is handled in GEST at two levels: at the level of the definition of the component systems and at the level of the state transition functions for each of the component systems.

6) Another requirements, very handy in modelling recurrence is to give a user the option to declare synchronous variables. The detailed description of synchronous variables in GEST and the implementation algorithm is given in Chapter 4.

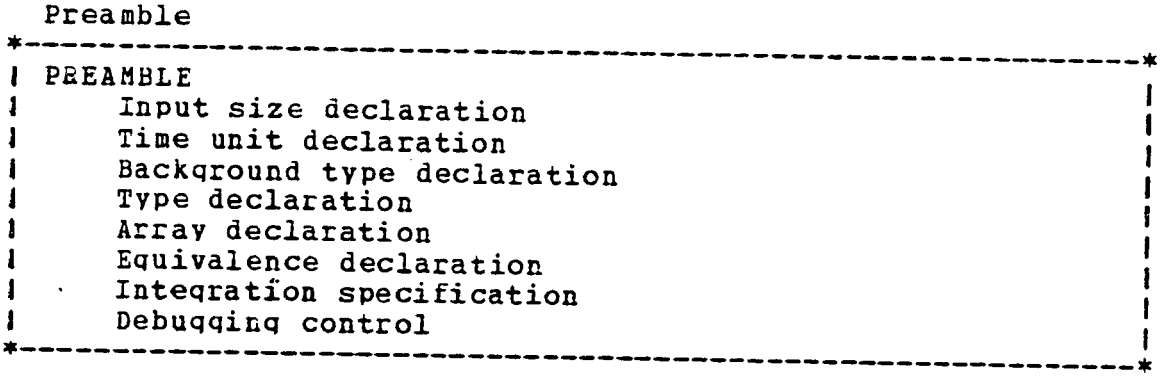
1.3 Block representation of GEST programs:

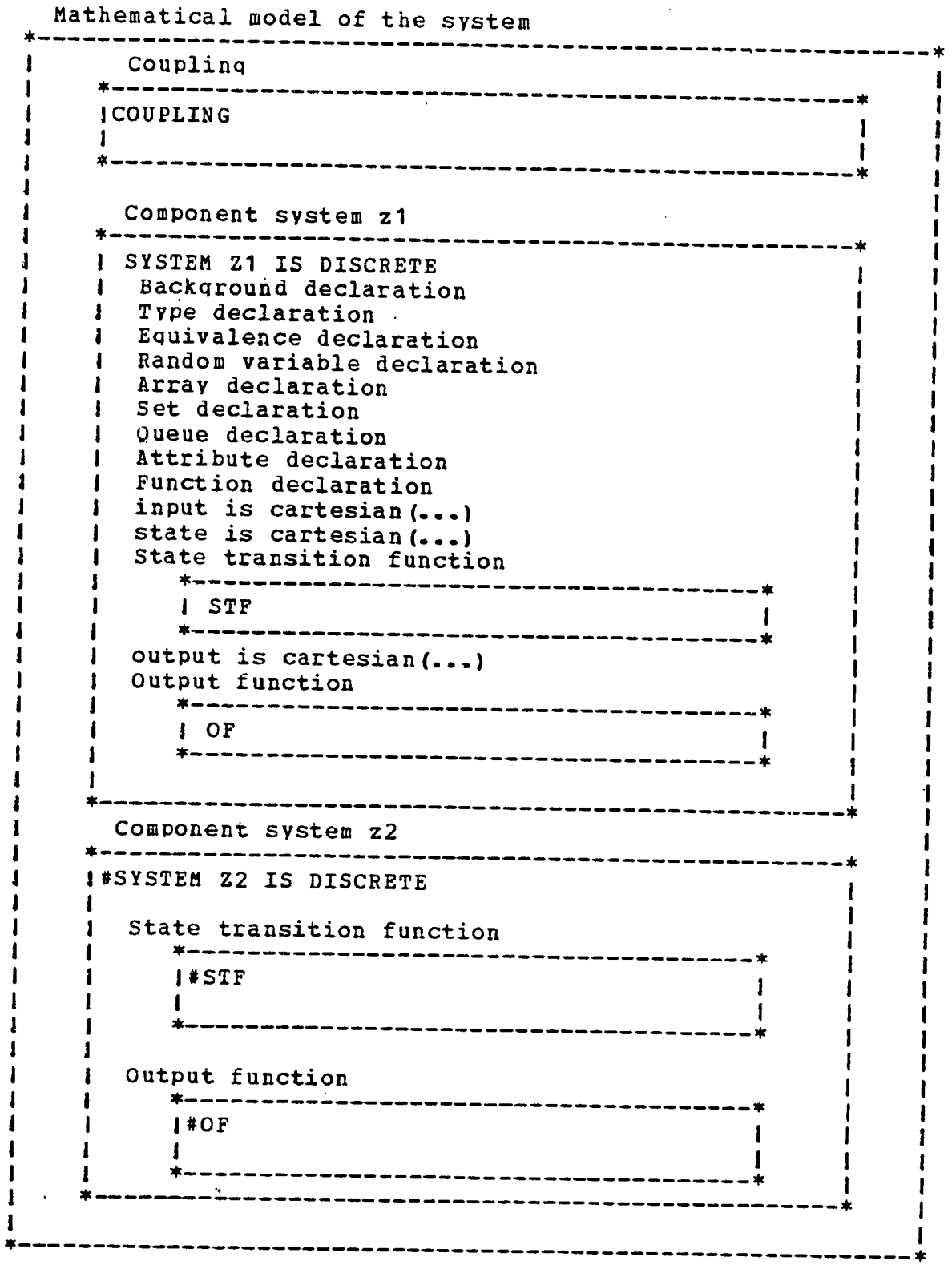
A GEST simulation model consists of two parts:

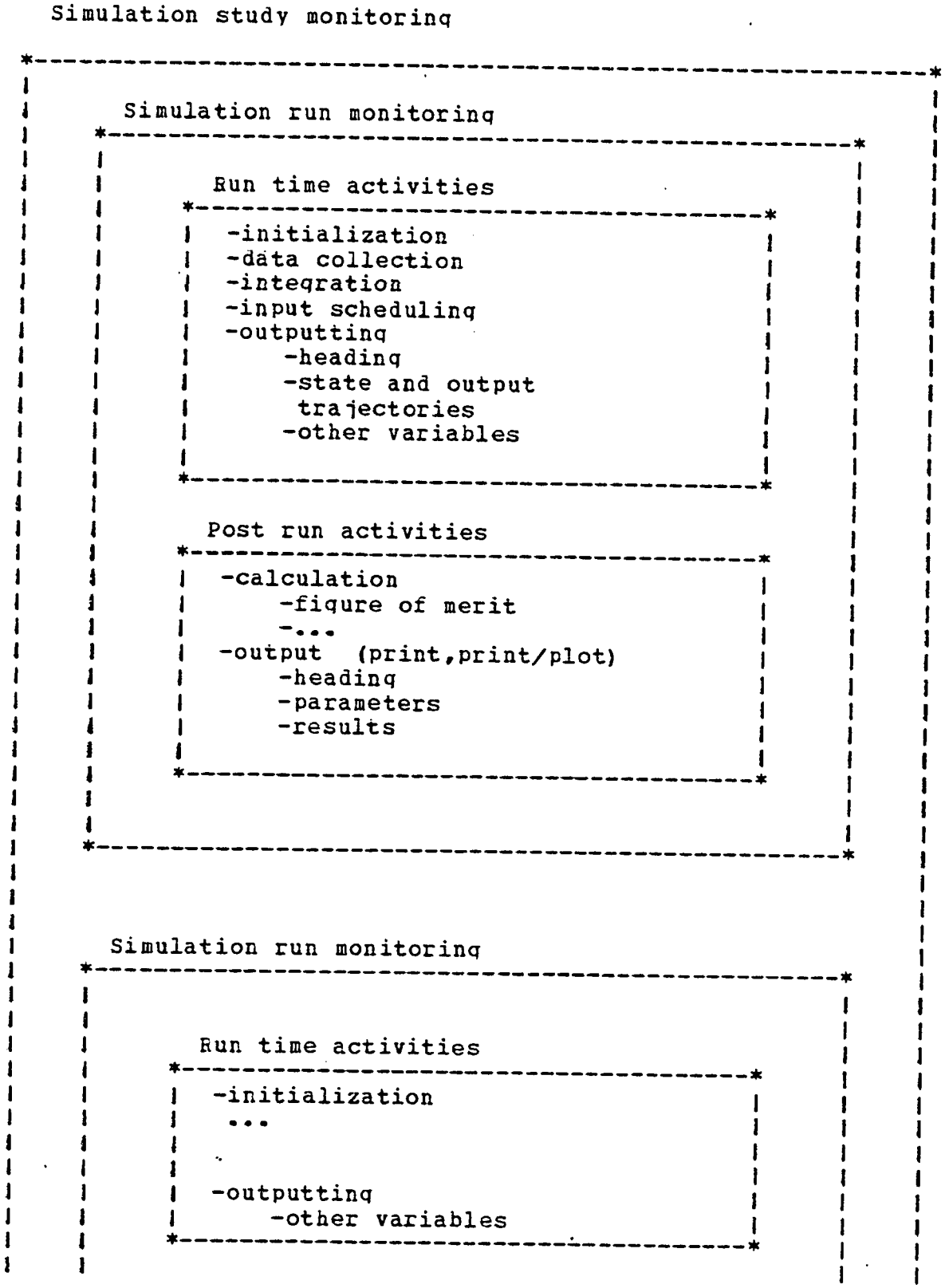
1) the first part is a mathematical model of the system. It may done by defining models of component systems and their interrelationships. Every such component system operates on sets of input values, state values, output values using state transition and output functions. The component systems and a definition of how they interact with each other describe the total system.

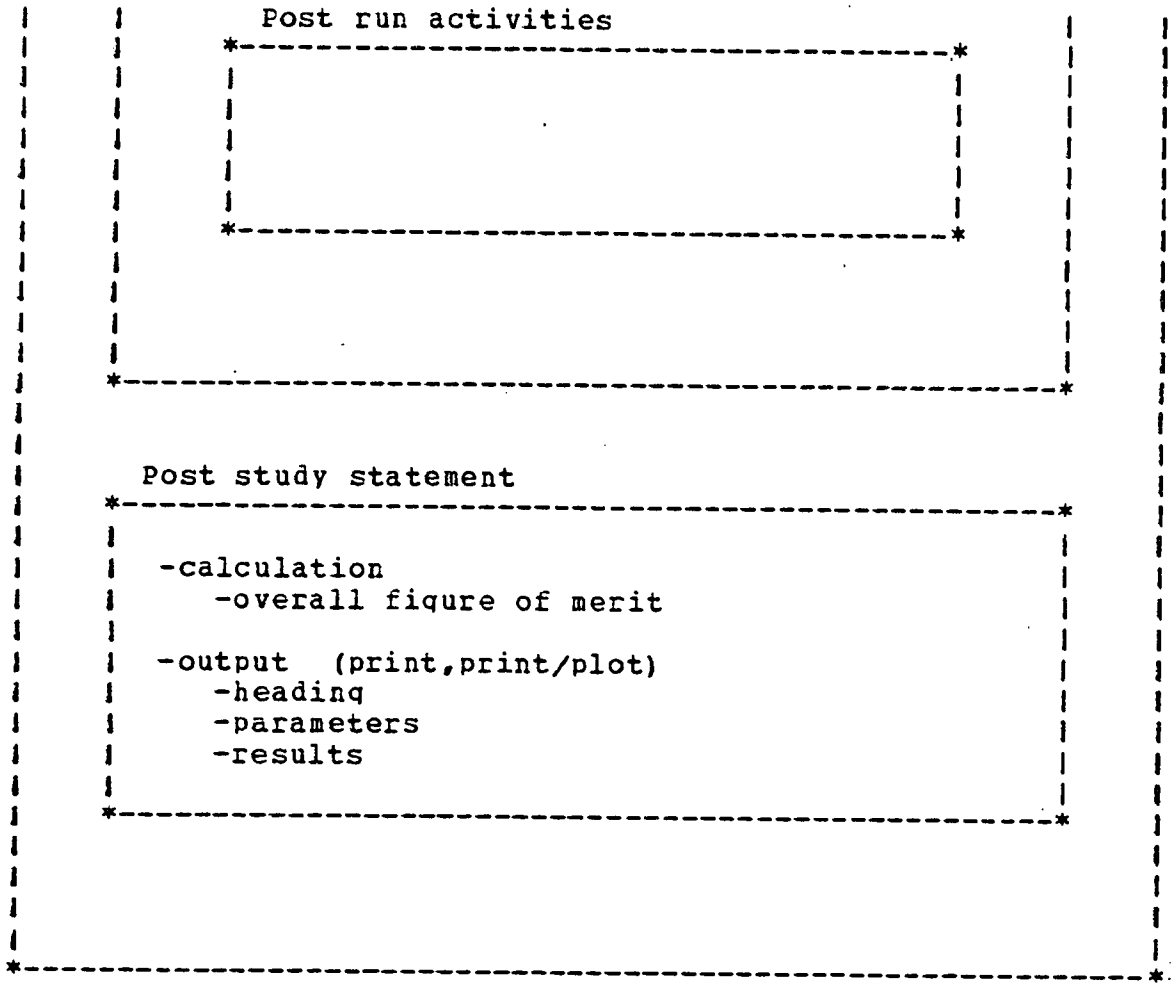
2) the second part consists of the statements to monitor the simulation study and to perform all desired experimentations for a simulation run. For example, statements to initialize and reinitialize the model, specification of an external input or several types of output.

A typical sequence of statements in GEST and a description of both parts are shown in the following block diagram.




Mathematical model of the system
Coupling
COUPLING
Component system z1
SYSTEM Z1 IS DISCRETE
Background declaration
Type declaration
Equivalence declaration
Random variable declaration
Array declaration
Set declaration
Queue declaration
Attribute declaration
Function declaration
input is cartesian(...)
state is cartesian(...)
State transition function
STF
output is cartesian(...)
Output function
OF
Component system z2
#SYSTEM Z2 IS DISCRETE
State transition function
#STF
Output function
#OF


Simulation study monitoring
Simulation run monitoring
Run time activities
-initialization
-data collection
-integration
-input scheduling
-outputting
-heading
-state and output trajectories
-other variables
Post run activities
-calculation
-figure of merit
-...
-output (print,print/plot)
-heading
-parameters
-results
Simulation run monitoring
Run time activities
-initialization
...
-outputting
-other variables


Post run activities
Post study statement
-calculation
-overall figure of merit
-output (print,print/plot)
-heading
-parameters
-results

CHAPTER 2

2.0 Implementation of high level languages

The success of a programming effort depends not only on the hardware on which the application is to be run, but also on the available programming systems. The programming systems generally include assemblers, compilers, translators, interpretors, report generators, program generators, various utility programs, etc. The purpose of programming systems is to increase the productivity of a programmer by providing functions that make it easier to program an application for operation on a computer.

In programming an application, the major problem confronting the programmer is the communication of the problem to the computer in a format acceptable to the machine circuitry. The set of representations, formats and rules used to communicate the problem is a programming language. Specifically, a programming language is a language used to prepare computer programs.

The excessive amount of time required to write programs in machine language, the clerical work needed to keep track of storage positions and assignments, the difficulty of renumbering all the various operation codes, etc., led to the development of

symbolic languages that use mnemonic (meaningful) terms.

In many cases, a specific language may be appropriate for the description of a given application but inappropriate for execution on a computer or for the optimization of subsequent execution time. The process of translating from one language form to another is of interest, therefore, particularly when it involves translation programs from source language to machine language. A computer program which operates on programs as data is called a language translator.

Three main types of translators can be identified: Compilers, assemblers and interpreters. A compiler accepts programs expressed in a given language, i.e., the argument language, and produces corresponding programs expressed in a second language, i.e., the function language.

An assembler is a special case of a compiler. The statements of the source program are primitive, in that they need not be analyzed into component statements. Since a fixed correspondence usually exists between symbolic instructions and actual machine instructions and between symbolic locations and addresses, translation essentially consists of a substitution of numeric values for symbolic operations or operands.

An interpreter is a special case of a translator. The object program is normally generated as an intermediate result and is discarded after it has been executed by the interpreter. The execution of an interpreter usually proceeds as follows:

each statement in the argument language is first translated to an intermediate function language, is then interpreted in that language, and is finally executed. Only the results of the executed statement are retained. It follows that statemants from the argument program are selected for processing by the interpreter in a sequence determined by the execution of preceeding statements.

The objective of an assembler program is to produce machine language function programs. Although the majority of compiler programs also produce machine language function programs, they are not so restricted by definition. In fact many compilers do produce assembler language function programs.

Most computer applications require mainly a combination of mathematical, file processing and retrieval techniques. Accordingly, these facilities are well ingrained in most procedure-oriented languages. Translators, tend to involve character manipulation, table construction and searching and various types of error analysis.

The processing programs consist of language translators, system service programs and user written problem programs. The programmer uses these programs to specify the work that the computing system is to perform and to aid in program preparation.

CHAPTER 3

3.0 Implementation strategy used for the GEST translator

The GEST processor, written in FORTRAN IV has been gradually developed from a program, called Synchronous Variables Analyser. The algorithm of this program is described with more details in Chapter 4. Initially the program was able to handle single words only, with no more then 4 symbols in an identifier. It was then extended to a program which can work with up to 16 characters in an identifier.

To start with the program analysing synchronous variables was not choosen accidentialy only, but because of the importance and the basic feature of GEST - to handle synchronous variables and their essential occurrence in simulation of large complex systems.

The processor analyses input GEST program, with its syntax rules as Appendix A, and translates the GEST program into a FORTRAN IV program.

The object FORTRAN program is not only a translation of the GEST source program, but it also includes all necessary statements to build, run and monitor a simulation run.

The GEST language was defined and developed as a (Oren,

1971a) Ph.D. Dissertation, for simulation of large scale complex systems.

The process of translating GEST source program into a FORTRAN program has two steps. The first step builds the main FORTRAN program and tables of identifiers, the second step creates one subroutine for every component system of a given GEST input program. Names of the generated subroutines correspond to the names of the component systems. Thus the maximum length of the name of a component system is 6 characters, where the first one must be alphabetic, and the remainder alphanumeric. This feature imposes the requirement for a unique name for every component system.

Monitoring of the execution of subroutines is controlled by CALL statements in the generated FORTRAN main program.

For all searches in the tables, created by the GEST processor, a hashing technique is used to reduce time. Subroutine TS725 generates hash codes for input strings and returns all other necessary information like number of single computer words to store such a string and a number of characters in the string.

An input GEST program is read and its statements are analysed. Subroutine TS729 (service routine TS72.9) - comment analyser is called to check if there is a GEST comment card. All GEST comment cards are converted to the FORTRAN format. Two

listings are produced from the GEST program. The first listing is the original GEST program, the second one is the complete generated FORTRAN IV program.

Subroutine CHECK uses a hash code of the first identifier of the GEST source to execute a statement of the processor, which will take proper action corresponding to the input identifier. For instance if the first identifier is the GEST keyword SYSTEM, the appropriate segment of the main processor is invoked and a pattern for subsequent call statements is established.

This structure of modular solution, based upon a path keyword/hash-code/ action taken, through the subroutine CHECK gives an easy way of updating the processor. By the updating is meant a process of adding new modules to handle additional GEST statements.

In general, every statement of a GEST program has a corresponding ENTRY and its module in the processor. The action or series of actions to be performed varies. If the structure of a GEST source statement does not conform to the GEST grammar, an error message is produced or if a fatal error is detected, the processing is terminated.

To give the user more freedom, the length of state variables names and other identifiers in GEST (except component system names) may have up to 16 characters. Source GEST program can contain an underscore as a symbol to express a concatination

of two words to create an identifier. Maximum length of such an identifier is again 16 characters (including the underscore). When the underscore or a longer name (7-16 characters) is used, the processor generates a new temporary name of the form TTTI ((I=1,2...). These new generated names (identifiers) are used in the generated FORTRAN program.

The table of identifiers is generated and every time a new identifier is recognized by the processor, the table is automatically updated. The most pertinent information about variables is stored (name, hash code, flag for state variables, pointer where the state variable has been declared, where used, and a flag if a remembered value (of the form ZZZI) has already been generated, another flag for identifiers which generated new temporary names (of the form TTTI).

The translation of a GEST source program into a FORTRAN program starts with reading in the GEST source program. The processor analyses the contents of every card and subroutine TS725 is called to get the first identifier on the statement currently being processed. The identifier may be a constant, a variable or a keyword. To distinguish to which of the above categories it belongs, the same subroutine generates a hash code. The hash code than gives an entry to the table of identifiers. content of this table entry is compared against the identifier and if there is a match - in the part of the GEST keywords - switch to execute the corresponding module of the main program is

done throuqht subroutine CHECK. If there was not a match against GEST keywords, the entries of the table of identifiers are searched next. If the table entry of that identifier is empty, corresponding values are inserted.

Because most of the GEST program statements must be placed in a certain sequence (not because of a GEST restriction but the logical aspect of passing the information to the processor) we may assume that all statements necessary to describe a component system are present and then the following will be a description how the GEST processor, using different subroutines handles the process of the translation.

At the beginning of every GEST program there may be PREAMBLE with the following statements:

- INPUT SIZE DECLARATION - if the input size is not declared, it is assumed to be 72. If this card is present it specifies the number of columns of the GEST program cards used for coding. The processor will set maximum range for the corresponding DO - statements.

- TIME UNIT DECLARATION - specifies unit of time used for the given simulation and it overrides the default time unit (HOUR).

- BACKGROUND TYPE DECLARATION - the occurence of the keyword BACKGROUND causes the subroutine TYPE to be called to analyse the BACKGROUND statement. There may be a common

BACKGROUND declaration for all component systems, defining all identifiers to be of one given mode- overall BACKGROUND declaration. Or the BACKGROUND declaration is given for one component system only, LOCAL BACKGROUND DECLARATION, in this case the component system name is given.

If the BACKGROUND declaration is not present the processor generates an IMPLICIT REAL statement for all identifiers of the system description.

- TYPE DECLARATION - when the GEST processor detects in the GEST source program the TYPE declaration, subroutine ARAVEC is called to analyse the declaration and to generate for instance a statement REAL C(5) for GEST Type Declaration REAL ARRAY C(5) or INTEGER Z(5),V(7),Q(10) for INTEGER VECTORS (5),V(7),Q(10), or DIMENSION Y(10),Z(50),W(2,10) for GEST source statement ARRAYS ARE Y(10), Z(50), W(2,10).

- The synonyms to designate a given variable in the EQUIVALENCE DECLARATION. The Equivalence Declaration must appear at the beginning of a system declaration. All variables placed at the same equivalence group are assumed to be of the same mode. When EQUIVALENTS ARE... Is recognised, subroutine EQUIV will generate the corresponding FORTRAN EQUIVALENCE statement(s).

After the PREAMBLE statements there are statements describing a mathematical model of the system. First statement

identifying a component system is a SYSTEM DECLARATION. To handle all tasks concerning the GEST SYSTEM declaration, routine SYSTEM is called. This subroutine checks the syntax of the statement, separates the component system name and stores it in a vector of component system names. If the system name is not unique, a fatal error message is printed and the process of translation is terminated. Similarly if the component system name exceeds the maximum length a warning error message is produced. Because of the ability of the present processor to handle only discrete systems, the keyword DISCRETE must be detected in the SYSTEM declaration statement, otherwise the process of translation is terminated (CONTINUOUS systems).

After the System Declaration there may be another BACKGROUND, TYPE, EQUIVALENCE DECLARATIONS for the given component system.

Syntax of the GEST INPUT DECLARATION is analysed by subroutine INPUT.

The next statement of the component system is a STATE VARIABLE DECLARATION. This declaration gives the names of the State Variables used in state transition functions. Subroutine STATE checks the correct occurence of GEST keywords, separates names of the state variables and returns information on them to the main program. The main program generates entries for the state variable names in the table of identifiers. A flag distinguishes them from other identifiers of the component system

The STATE TRANSITION FUNCTION DECLARATION signals that one or more State Transition Functions follow. Of course the system (the component system) should have the same number of state transition functions as there are state variables specified in the state variable declaration of that component system.

Every state transition function is analysed and the table of state variables is updated with line numbers where every state variable is declared and where it is used. If the state variable needs to be replaced by a temporary variable to hold its initial value, the GEST processor generates the new remembered value, updates the table of state variables with a flag, showing that the new remembered value was already generated and a new arithmetic assignment statement of the form

State Variable Name = Remembered Value

is added to the state transition functions of the source GEST program.

The state transition functions may be followed by output functions which describe the output from the GEST component system. The conversion of these output functions into FORTRAN is done almost without any changes to the original source GEST code. Between the last state transition function and GEST OUTPUT FUNCTION there may be GEST OUTPUT STATEMENT to print values of the state variables. Detailed description of this conversion is given in Chapter 5.

One component system description may be followed by others and the whole system is composed of the interconnection of the component systems.

The Simulation Study Monitoring in GEST is done by two groups of statements. First group of statements (Run Time Activities) is used for specifying the number of simulation runs to be performed, initialization, data collection, integration input schedulling and outputting. The second group of statements (Post Run Activities) serves to calculate figure of merit and to generate output - print, print/plot. The present GEST processor has implemented only the first group of statements.

CHAPTER 4

4.0 Implementation of synchronous variables in GEST

The state variables used to represent the state of a system are the synchronous variables and provide the basic parallel computational of GEST.

A simple example of the usage of synchronous variables (lagged variables) is given below.

Suppose one wants to calculate the values of a and b at each time t, where t [1,n]. The initial values are a and b. And the recurrence relationship giving the values of a and b (at time t) are given by the following very simple relation:

a =a *b

b =a +b

To be able to calculate the time series for a and b, the user can define a dummy variable Z and can write for example the following statement:

```
      A=A0
      B=B0
    1 Z=A*B
      B=A+B
```

```
A=Z
....        print statements
....        control statements
GO TO 1
```

If A and B are declared as synchronous variables with initial values A0 and B0 respectively, the statements

```
A=A*B
B=A+B
```

will be sufficient to define the recurrence relationships between the two variables.

From the above example it becomes obvious that the variable Z is used as a temporary variable to hold the remembered values of the A. The introduction of such (a) remembered variable(s) is necessary to be able to update the current values of the variables used in the state transition functions.

To eliminate the problem of redefining and rewriting the original state transition functions, one has in GEST a facility which will handle the synchronous variables. The user must define in the GEST State Variable Declaration of each component system the names of the state variables. GEST processor analyzes the State Variable Declaration and the State Transition Functions (STF) and all the necessary changes will be generated automatically. This gives the user a great flexibility. The user may have for example n state transition functions for n

state variables, placed in any order without being obliged to take care of the temporary variables to hold the remembered values of the synchronous variables. Any additional changes like inserting, deleting or modifying of the state transition functions can be done without a modification of any other state transition function(s).

To determine the next state of a component system from its current input and the current state, the state transition function(s) are used. This basic feature of the state transition function makes it possible to define synchronization of the input / output relationships. For simulation of large complex systems synchronous variables are essential. (In GEST the state variables of a component system are by definition synchronous variables.)

Within a mathematical model of a component system the state transition functions may be placed in any order and even any variable of any component system can be referred to in any other component system.

The algorithm used to implement the synchronous variables in GEST is described in the following flowchart.

At the beginning, the algorithm reads a list of state variables of a component system. If the component system has only one state variable there is no need to have a temporary variable. For this case the work area called work area 1 will

store the state transition function.

If the state variable list of a component system has more then one state variable, then the algorithm will analyse the state transition functions. The number of state transition functions should correspond to the number of state variables of the component system.

To decide where the remembered value should be used is based on the rule: If a reference is made to a state variable in a state transition function after its appearence at the left hand side of the state transition function, there is a need to generate a remembered variable to hold its remembered value. The replacement of a state variable is done at the left hand side of a state transition function, where it was first defined. The GEST processor generates for temporary (remembered) variables names of the form ZZZI, where i is a sequence number.

When all state transition functions of a component system are processed and placed into the work area 1, the processor generates assignment statements, one for every generated remembered variable. The assignment statement is of the form:

'replaced state variable' = ZZZI .

The number of remembered variables, generated by the GEST processor, may be reduced changing the order of state transition functions. If we consider again equation (1) and (2), where A

and B are the synchronous variables:

```
A=A*2+COS(2.5*A)                    (1)
B=A+B+SIN(.72*B)                    (2)
```

To be able to evaluate equation (2) properly we need to replace the state variable A at the left hand side of the equation (1) by a remembered value, in this case by ZZZ1.

The state transition functions (1) and (2) will be interpreted by the GEST processor, the following way:

```
ZZZ1=A*2+COS(2.5*A)
B   =A+B+SIN(.72*B)
A   =ZZZ1
```

if however equation (2) preceeds equation (1) there is no need for having a remembered value for A. The rearranging of the order of the state transition functions can be performed by a topological sort. The topological sort in conjunction with the algorithm of synchronous variables will minimize the number of remembered values generated by the GEST processor.

CHAPTER 5

5.0. Definition of run time outputting in GEST

A mechanism used for exchanging information with the outside world from the point of view of a computer may be a program or input data for a program or results of the process of a computation. In terms of a given simulation language we may be interested in processing pieces of information of the segment of a compiled program and to follow the flow of information.

5.1 General description and implementation

Two kinds of run time outputting are handled in GEST. Firstly it may be GEST own PRINT statement of the form:

1) PRINT FOR FIRST PAGE n HEADING LINE(S)

or

2) PRINT <variable list>

Or FORTRAN IV type of the WRITE(a,b)<list> statement. Their implementation in GEST is handled in the following manner.

The first PRINT statement may be used at two forms:
-case 1(i):

PRINT n HEADING LINE(S)

This format assumes that the n cards consisting of the heading line(s) will follow the PRINT statement. Heading lines will be printed on the top of every new page.

Implementation of the case (i) was done throught pre-assigning an internal storage to accomodate the given Heading Lines. Usage of this solution is then restricted and number of heading lines must be less or equal to nine. In the case of allocating a temporary disk file the number of the heading lines may be unlimited. It is a matter of a practical consideration if there is a need to have a power of storing of a larger number of

such heading lines.

-case 1 (ii) :

PRINT FOR FIRST PAGE n HEADING LINE(s)

The GEST processor will generate FORTRAN format WRITE statements. The generated FORMAT statement has as Hollerith field the given heading lines. A temporary storage for 80 characters at time is needed. The heading lines of this case are printed ONCE only on the top of first page.

-case 2(i) :

PRINT <VARIABLE LIST>

Where number of variable in the ''variable list'' does not exceed eight. Assuming to have variables ID1, ID2, ID3, ID4,... , ID8 in the variable list, the GEST processor will generate following output:

TIME	ID1	ID2	ID3	
....	....	....	....	....
....	....	....	....	....
....	....	....	....	....
....	....	....	....	....
....	....	....	....	....
....	....	....	....	....
....	....	....	....	....

For the above variables (TIME, ID1, ID2,... , ID8) the maximum of 11 printing positions for each is used. (FORTRAN Format E11.4 is used for real variables)

-case 2 (ii) :

PRINT <variable list>

Will handle GEST PRINT statements with more then 8 variables within the <variable list>. In this case the processor generates output having 5 variables printed in one line. The first variable of each such an output is always TIME, followed by all variables from the <variable list>.

CHAPTER 6

6.0 . Implementation of simulation monitoring

A simulation study consists of a sequence of one or more simulation runs, where a simulation run is an uninterrupted recording of the system's performance under a specified combination of controllable variables. Another definition of a simulation run (Oren, 1974): Simulation run is the act of observing the behavior of a system for a given period of time. The simulated time starts at t initial and ends at t final.

The simulation study monitoring is the specification of a sequence of simulation runs and the post study activities. The simulation study monitoring in GEST consists of two parts: (1) simulation run monitoring and (2) post study statement.

The simulation run monitoring is specification of run time time activities and post run activities. Run time activities consists of the initialization of the model, data collection, integration specification, input scheduling, and specification to output the state trajectory and the output trajectory of the system. When a simulation run is finished we may be interested to perform some extra calculations. During this period the simulation time is not taken into account. If required to calculate figure of merit it is based on the behavior of the

system within the simulation run. To display post run calculations (mainly the figure of merit of the system) there are in GEST statements to generate the outputting. Both, the post run calculations and the output statements are called post run activities.. Another term used in the same sence is post run statements.

When one simulation run is completed (run time activities and post run activities), the same mathematical model of the system can be used to define another simulation run. There may be any number of simulation runs.

After completing a simulation study (one or more simulation runs) we may be interested to describe post study activities. The post study activities may include calculations of the overall figure of merit of the system, based on the figure of merits calculated after each simulation run and statements to display the calculated results.

The figure of merit of a system is a symbol which summarizes how well a given system satisfies the goal of its existence in a given scenario. For complex systems the figure of merit will be a multi dimensional array. Each simulation run will have its own figure of merit so more than one figure of merit is calculated in terms of all simulation runs. An overall figure of merit is calculated based on the figure of merits calculated for each of the simulation run.

The following is a description and an example of a

simulation run monitoring in GEST. The statement ''RUN UNTIL <label>'' starts the description of the run time activities. The <label> delimits the range of the run time activities. Between these two statements may be placed any necessary system initialization (or reinitialization), data collection statements, input scheduling and so on. After the <label> will be placed all statements of the post run activities (calculation of the figure merit, output statements). Every simulation run monitoring ends with ''ENDRUN'' statement.

CHAPTER 7

7.0 GEST Example

To illustrate the process of translating a GEST source program into FORTRAN IV program, the next pages provide an example.

The System Z3 is being defined as a discrete system with input I33. TRANSITION Function (S.t.f.) section. Output from the system is represented by three variables O31, O32, O33 and are calculated for the given time frame (starting at time 0 and incremented by 1 until its value reaches 12). The values of input I33 are given in the GEST source program.

The FORTRAN IV program resulting from the translator has main program which is providing all the houskeeping (according to the GEST source) and the necessary initialization. For every system there will be one subroutine.

Values of the state transition functions and the values of the output variables are calculated in the SUBROUTINE Z3.

Last page of this chapter illustrates the behaviour of the system over the specified time frame.

```
      SYSTEM Z3 IS DISCRETE
      INPUT IS CARTESIAN(I33)
      STATE IS CARTESIAN (G,H)
      S.T.F.
      G=H-(G/100)+I33
      H=-(G/100)+(I33/3.)
      OUTPUT IS CARTESIAN(O31,O32,O33)
      O31=G+H
      O32=G*2.
      O33=7.6*H+3.2
      RUN UNTIL 1973
      PRINT 2 HEADING LINES
      OUTPUT FOR SYSTEM ''Z3''
      STATE VARIABLES ARE : G, H    INPUT IS I33
      PRINT G,H,I33,O31,O32,O33
      INITIALIZE (Z3)
      DATA INPUT (Z3) AT EVERY DAY START AT TIME=0. UNTIL TIME=11.
      I33 12. 9. 7.3 5.1 22.4 31.5 2. 16.1 19.32 15. 21.2 33.03;
1973:ENDRUN
```

```
FORTRAN IV G LEVEL  21                    MAIN                DATE = 77105          13/26/58

            C     MAIN PROGRAM
 0001             REAL * 4 I33
 0002             DIMENSION ZZZ2(  13)
 0003             DATA ZZZ2/12.,9.,7.3,5.1,22.4,31.5,2.,16.1,19.32.15.,21.2,33.03,
                 1'NULL'/
 0004             COMMON C31,032,033
            C     RUN 1
 0005             TIME=0.
            C     EXTERNAL INITIALIZATION OF SYSTEM Z3
 0006             G=7.1
 0007             H=36.2
            C     INTERNAL INITIALIZATION OF SYSTEM Z3
 0008             IS=0
 0009             CALL Z3  (H , G , I33 ,IS)
 0010             IS=1
            C     PRINT HEADING LINE(S) AND PRINT WITH LIST
 0011             WRITE(6,1)
 0012           1 FORMAT(1H1,1X,
                 1'        OUTPUT FOR SYSTEM  ''Z3''
                 1                 '/1H ,1X,
                 2'        STATE VARIABLES ARE : G, H  INPUT IS I33
                 2                 '///)
 0013             WRITE(6,2)
 0014           2 FORMAT(1H ,7X,
                 1'TIME          G               H               I33             031             032
                 2             033
                 3')
 0015             WRITE(6,3)TIME,G,H,ZZZ2(1),031,032,033
 0016           3 FORMAT(1H0,7(2X,F11.4))
 0017             DO    11     KKK2=1, 12
 0018             I33            =ZZZ2(KKK2)
 0019             CALL Z3  (H , G , I33,IS)
 0020             TIME           =KKK2
 0021             WRITE(6,3)TIME,G,H,ZZZ2(KKK2+1),031,032,033
 0022          11 CONTINUE
 0023             STOP
 0024             END
```

```
FORTRAN IV G LEVEL  21                    Z3                    DATE = 77105

0001                  SUBROUTINE Z3 (H , G , I33 ,IS)
0002                  REAL * 4 I33
0003                  COMMON O31,O32,O33
0004                  IF(IS)1,2,1
                    C SYSTEM Z3 IS DISCRETE
                    C INPUT IS CARTESIAN(I33)
                    C STATE IS CARTESIAN (G,H)
                    C S.T.F.
0005                1 ZZZ1=H-(G/100)+I33
0006                  H=-(G/100 )*H +(I33/3.)
0007                  G=ZZZ1
                    C OUTPUT IS CARTESIAN(O31,O32,O33)
0008                2 O31=G+H
0009                  O32=G*2.
0010                  O33=7.6*H+3.32
0011                  RETURN
0012                  END
```

OUTPUT FOR SYSTEM 'Z3'
STATE VARIABLES ARE : G, H INPUT IS I33

TIME	G	H	I33	O31	O32	O33
0.0	7.1000	36.2000	12.0000	43.3000	14.2000	278.4397
1.0000	48.1290	1.4298	9.0000	49.5588	96.2580	14.1865
2.0000	9.9485	2.3119	7.3000	12.2604	19.8970	20.8900
3.0000	9.5124	2.2033	5.1000	11.7157	19.0247	20.0654
4.0000	7.2082	1.4904	22.4000	8.6986	14.4164	14.6471
5.0000	23.8183	7.3592	31.5000	31.1775	47.6365	59.2501
6.0000	38.6210	8.7472	2.0000	47.3682	77.2421	69.7994
7.0000	10.3609	-2.7116	16.1000	7.6494	20.7219	-17.2980
8.0000	13.2848	5.6476	19.3200	18.9324	26.5696	46.2418
9.0000	24.8347	5.6897	15.0000	30.5245	49.6695	46.5619
10.0000	20.4414	3.5870	21.2000	24.0283	40.8827	30.5810
11.0000	24.5826	6.3334	33.0300	30.9160	49.1651	51.4541
12.0000	39.1176	9.4531	**********	48.5707	78.2352	75.1634

CHAPTER 8

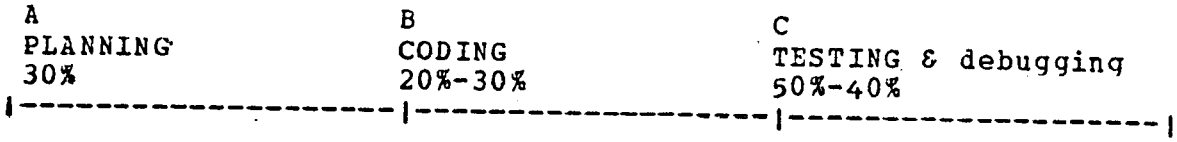
8.0 Notes On The Possible Implementation

First implementation of GEST provided the opportunity to collect ideas for a new GEST implementation. All points of this chapter have been chosen very carefuly, to help to overcome difficulties recognized during the first language implementation, to include those new ideas discovered too late to be considered and to satisfy new requirements of the language.

The notes on the possible implementation will cover the process of the initial language description - the preparation of all information for a successful coding, testing and GEST processor debugging, so at the end there will be available a system - including all documentation and an installation manual and users guide.

It is assumed that during the implementation, at every level of the project enough documentation will be generated to help to go back if necessary and to be able to redefine a part of the system without being obliged to start from scratch.

The process of the implementation is divided into three phases A, B and C.



8.1 The PHASE A , Planning And System Description

The phase A, called PLANNING And SYSTEM DESCRIPTION should be used to specify very clearly the main objectives and, generally speaking, to define all project requirements. To satisfy this goal at least ten steps can be identified:

8.1.1 System Description.

8.1.2 Specification Of The GEST Components.

8.1.3 System Syntax Description.

8.1.5 Interaction Of Algorithms Concerning The Functional Flowchart .

8.1.6 Detailed System Specification And Limitations, Based On The System Syntax Description .

8.1.7 Selection Of A Language To Suite The System Needs.

8.1.8 Specification Of Test Input Streams.

8.1.9 Documentation And Evaluation Of The System Description To Achieve The Goal Of The Defined System .

8.1.10 Personnel And Software - Hardware Requirements.

8.1.1 System Description. DEFINING the objectives of the system - GEST processor and drawing of a functional flowchart of the whole system.

The GEST processor will accept a description of a simulation system which is described in the GEST simulation language. The system can operate on discrete, continuous or combined (DC) time frame, so the GEST processor must be capable of handling discrete, continuous or combined simulations.

Input - source GEST program - as the description of a simulation can be placed on any valid input media - cards, tape, disk or drum file.

The next implementation is expected to handle not only BATCH processing, where the user has no control after the job is submitted to the system for execution, but to have in addition an on-line interactive capability. This would greatly help the user in developing a description of a simulation model. Perhaps, to follow the idea of Time Sharing Option a user could be allowed to call for GEST PROMPTER which could create a skeleton of a GEST source program, and on an interactive basis to provide information on the syntactical structure of the language.

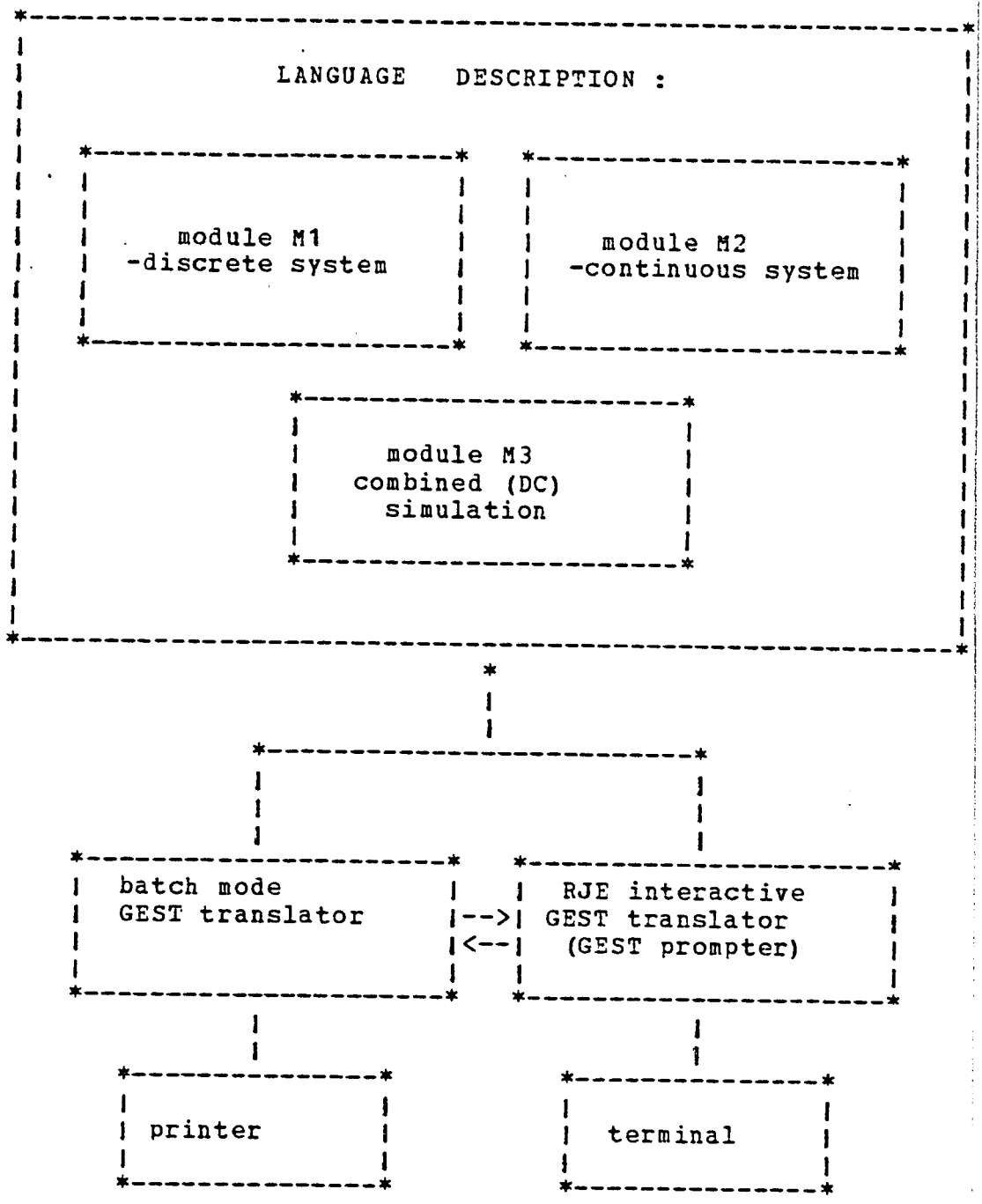


FIG. A 1)

General System Description

8.1.2 Specification Of The GEST Components.

The GEST translator should be composed of programs of two categories. Control programs and Processing Programs.

The Control Programs are called Initial Task Handling Routine (ITHR), Job Execution Routine (JER) and Input Output Routine (IOR).

The Initial Task Handling Routine (ITHR) will perform the following functions:

1) To analyse options of the GEST source program and to produce codes corresponding to the options. Two possibilities are essential. The default options will apply, if there is no GEST option card or if the option card is present, the given parameters will be used.

The parameters of the GEST option card may specify requirements on the GEST processor or Fortran compiler and or both. The last case will have some mutually exclusive parameters which should be listed as a note on the parameters of the GEST option card of the language description.

2) To verify a validity of the given parameters of the GEST option card and if necessary to distiguish the mutually exclusive parameters which will cause an abnormal termination of the job and to produce error massages from a group of the fatal error messages).

3) To produce a code for the valid options of the GEST processor and Fortran compiler. The code will be used by the Job Execution Routine (JER) to invoke the processor and compiler and to pass the selected options as correct keywords into the JER.

The user orientation of the GEST language should be accomplished by providing a way to allow a user to specify in an easy English-like form maximum time for execution of the job, maximum number of lines of the output to be generated and in the case of Remote Job Entry to specify where to direct the output. Most of these values and parameters are machine and installation dependent. The different approach will be in a case of a Service bureau and an in house computer.

To allow a user to direct the effectiveness of the program code he should be allowed to specify in a suitable manner execution of Fortran G or Fortran H compiler. The flexibility of a compiler version should be kept in mind even if another high level language will be used.

At least the following options should be available to GEST user:

On Generated Output :

1) Print option: (Direct output on printer and or create an output file on specified device)

-list GEST source only

-list generated Fortran program only

-list GEST source and generated Fortran program

2) Create a Permanent Fortran program file on specified device

3) Punch generated Fortran program

On GEST Input (Source) Program:

1) Mode of the input

-punched in EBCDIC

-punched in BCD

-mixed mode input (BCD and EBCDIC)

-user supplied routines in BCD

2) GEST source program with external references

Specification Of Job Execution Conditions:

-maximum time

-maximum number of lines of the GEST output

-selection of the compiler version

Generated Messages:

1) -List all diagnostic messages

-suppress all messages

2) -Produce list of options in effect

-suppress the list of options in effect

-list all available options

-print information on the current version of GEST processor

-print a skeleton of GEST program

For an environment with Remote Job Entries -CRT's or hard copy terminals a user should have a chance to ask for a syntax description of a GEST statement as an additional feature (an extention of the skeleton of GEST program).

The Job Execution Routine (JER) is used to submit the generated FORTRAN program for execution using the information passed from the Initial Task Handling Routine (ITHR). This step includes the process of generating Job Card, EXEC card, DD cards with values of parameters and keywords corresponding to GEST option card. All parameters are being passed from the Initial Task Handling Routine (ITHR).

The independence of this routine to GEST language translator gives a possibility to make an easy adjustment and modification for any given hardware system through a change or rewriting this routine only and to suite the requirements of the specific installation.

The INPUT / OUTPUT Routine (IOR) creates files and/or references to them, used directly or indirectly by the GEST language processor. Those files may be GEST input program files, output files generated by the language processor (request for permanent output FORTRAN files), user program files referred to in the GEST source program and data file(s) to be used for the input to the simulation program.

This routine and its output is controlled by the Initial

Task Handling Routine.

The second category of the GEST translator are Processing Programs, having two groups of routines. The language translator and service programs and programs supplied by a user of GEST.

(I) Language Translator and Service Programs:

GEST language processor itself will analyse the source GEST program and produce a corresponding FORTRAN program file.

The processor should be capable of checking SYNTAX of the source program, to create the corresponding tables, generate FORTRAN subroutine for every component system of the GEST program make the required conversion from GEST identifiers to valid FORTRAN identifiers (optionally for Fortran H Extended), to invoke GEST algorithms and if required to insert CALL statements to get SERVICE PROGRAMS.

The service programs will perform a function of the GEST source to produce a documentation of the GEST source program, to generate PRINT - PLOT, to initialize debugging routine and if necessary to produce error diagnostic messages.

(II) Users Programs:

Under the users Programs are understood FORTRAN IV programs inserted into the GEST source programs or references

made to FORTRAN source programs stored as single files or members of partitioned data sets. In the case of the references to programs, they must consist of all necessary information in the correct syntax of the GEST language.

The following figure gives a clear structure of the GEST Translator.

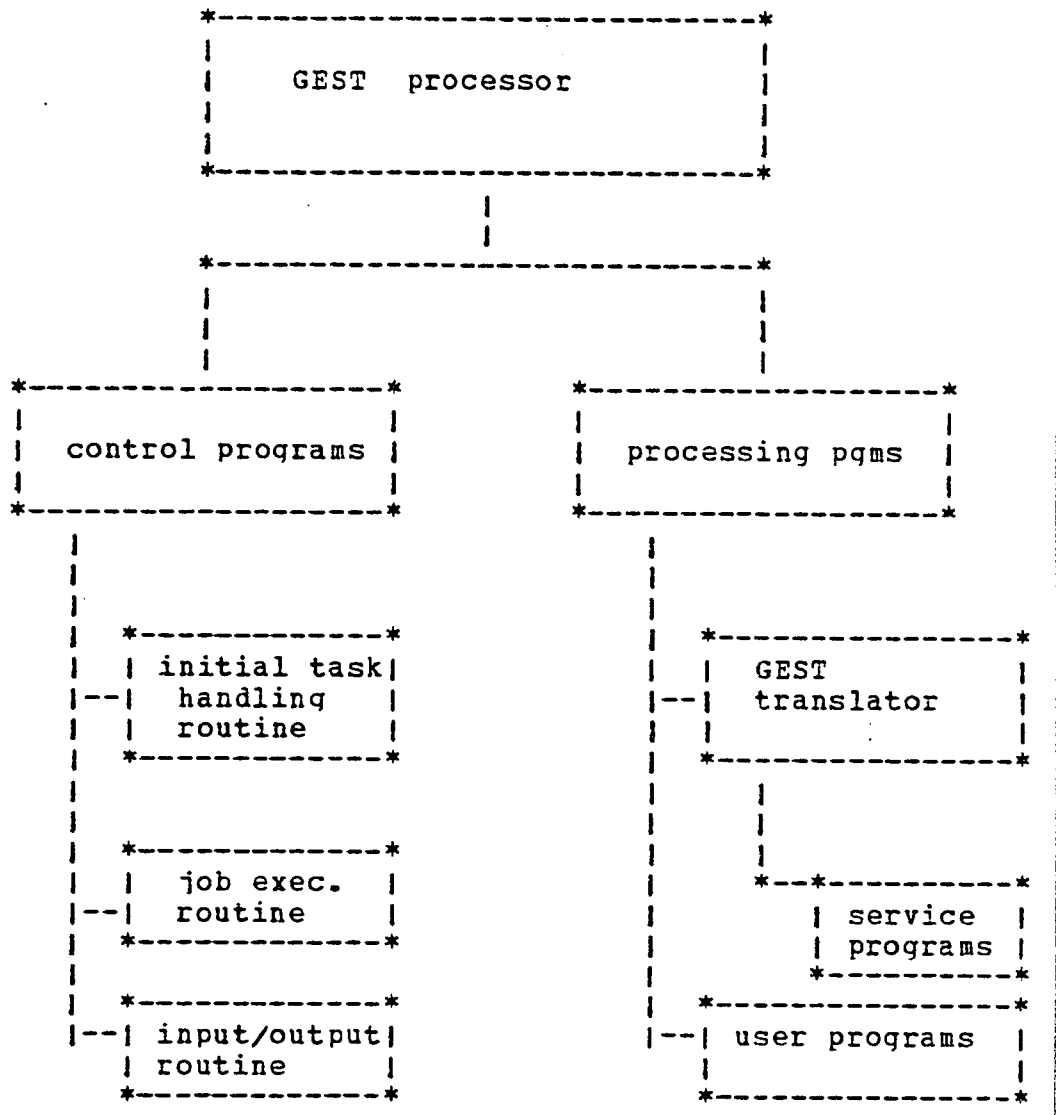


FIG. A 2)

Specification Of The System - GEST TRANSLATOR

8.1.3 System Syntax Description, general overview.

The GRAMMAR and the SYNTAX of the GEST language will be given. An extension should be made concerning the additional features for GEST PROMPTER. This phase should specify all possible combination of the occurence of GEST statements as acceptable for the GEST translator and restrictions causing error messages or termination of the job processing (fatal errors).

8.1.4 Development And Documentation Of Algorithms Used Within The Defined System.

Each algorithm will be clearly described and documented including a detailed flowchart. The documentation will state clearly input and output parameters, their modes (REAL, INTEGER, DOUBLE PRECISION and COMPLEX), length of every parameter in bytes and number of bytes used if less then the defined length.

8.1.5 Interaction Of Algorithms Concerning The Functional Flowchart.

The results of the previous step interfaced with the functional flowchart and the possible combination of the occurence of GEST source statements (as step number 3) will give an interrelationship of all accepted algorithms of the GEST translator. Based on the detailed parameter description one

would specify common blocks of variables for modules of GEST translator.

8.1.6 Detailed System Specification And Limitations, Based On The System Syntax Description, which will be used as an input for the phase B - CODING.

8.1.7 Selection Of A Language To Suit The System Needs.

With the assumption the language translator will be used, then the high level language should be selected. The leading idea for the language selection would be not only the programming needs and convenience but also machine independence criteria.

8.1.8 Specification Of Test Input Streams. The test programs should exercise all configurations of GEST in order to establish its correct implementation. The interface with the operating system should also be tested. Finally, the various user options should be tested as well as the relevance of the various diagnostic messages. This means not only correct (syntacticaly and grammatically) programs should be used for the testing but also programs, producing error diagnostic messages will be a part of the test streams.

In general the following areas should be covered by the test streams:

1) Work capability of the GEST processor

2) Software interface with an operating system

3) The correct diagnostic messages

8.1.9 Documentation And Evaluation Of The System Description To Achieve The Goal Of The Defined System.

8.1.10 Personnel And Software - Hardware Requirements.

8.2 The Process Of Coding - PHASE B

The phase B, generating all necessary coding as required using detailed specification from phase A, should not begin before the completion of the phase of planning and System Description.

The process of CODING - Phase B, may be approached at the following levels:

8.2.1 A Selection Of a Coding Technique And The Programming Strategy.

The primary requirement to be met in software development has always been to perform the function specified for the software. But, where at one time secondary emphasis was placed only on software efficiency, that is core and time required, today three other factors are recognized as requiring special emphasis. These factors are reliability, maintainability and extensibility.

To achieve the factors it is highly recommended to use structured programming. The essence of the Structured Programming is modular organization, with the enforcement of certain standards for the structure of modules and for creating the links between them, using a limited set of special structural and linking statements.

The structured programming is gaining acceptance as a means of:

(I) improving the quality of the code

(II) improving the project control

(III) reducing the debugging phase of the project

(IV) forcing good software documentation as an integral part of the system design

Hints of the Structured Programming:

-divide program into modules, where a MODULE is a set of one or more contiguous program statements having a name by which other parts of the system can invoke it and preferably having its own distinct set of variable names

-usage of labeled COMMON blocks rather then unlabeled ones

-minimize number of GO TO statements

-having ONE ENTRY and ONE EXIT per module

-readability and uderstandability - implies modules with 30-50 lines of program code

-top-down development - a methodology which structures the design processes of program development. The uniqueness of this structured approach is in the strict top to bottom implementation

of program modules (versus the generally used top-down design and bottom up implementation approach). The program is developed as levels of design in which modules are designed, tested and integrated before proceeding to the next design level.

8.2.2 Writing Of The Code Using The Selected Language And The Coding Technique.

The time dedicated for this level of the system coding should not exceed maximum 30 % of the total time of the project.

8.2.3 Creating A Sufficient BACKUP Copy For Every Completed Part Of The Project.

At the end of every phase of the project a copy of all software programs should be obtained and stored with a sufficient documentation. When modifications to the finished part are supposed to be done a copy of the original version will be kept. All back up copies should have READ only status (ring out for tapes) to prevent an accidential destruction.

8.2.4 The System SETUP For Testing And Debugging.

This level includes setting up the system library, linkage of modules and an availability of the system through simple EXEC statement on IBM installation (establishing all necessary cataloqued procedures), so the test streams can be executed.

8.2.5 Summary Of Specific Hardware - Software Dependent

Features. Invariably, the software system relies heavily on hardware features characteristic of the computing system on which it was developed. These should be identified.

8.2.6 Writing Of An Installation Manual, in which it is stated how install the system and how to run test programs. Complete set of JCL statements should be the most important part of the whole installation manual.

8.3 The PHASE C, Testing And Debugging

The final phase of the system implementation and development is the Phase C - TESTING and DEBUGGING.

Based on the succesful completion of the previous phase (B coding), one will be able to begin the fine portion of the whole system:

8.3.1 Creating And Running Of The Test Streams, and to start debugging. According to paragraph 8.1.8 the test programs (on different input media, options etc.) Will be created and executed.

8.3.2 Improvement Of The System Efficiency:

-optimizing logic of the system

-optimizing the program code

8.3.3 Creating a User Guide as a clear overview and the usage of the whole system. This should be accompanied by as many examples as possible to minimize the process of learning for new users.

8.3.4 Notes Toward a Future Implementation. To summarize the current implementation and to give ideas to improve the system for the simulation community.

8.4 Machine dependent features of the current implementation

The current implementation of the GEST processor was done on IBM/ System 360, model 65 (MVT) at Computing Centre University of Ottawa.

-(i) REREAD routine, Alias SETB99, RESB99. To allow to reread GEST source statements and to write and to reread records of any required length, the SETB99 routine is used. The routine itself is written in IBM 360/370 assembler language and it is listed in Appendix B of the Thesis.

-(II) Generated FORTRAN program, corresponding to GEST source, is passed on INTERNAL READER and submitted for execution.

Any FORTRAN unit can be used for the creation of the internal reader file. Data definition (DD statement) of the internal reader unit will have specified in data control block macro (DCB) information:
UNIT=INTRDR, BLKSIZE=80, LRECL=80, RECFM=FB, BUFNO=1 .

The feature of the internal reader unit is widely used on IBM 360/370 series computers, where a monitor program generates complete jobs as a function of an input data.

Appendix A Syntactical Rules Of GEST

The Appendix A has two parts. First part introduces an extended version of the Backus-Naur Form, a metalanquage, which is used to describe GEST simulation proqramminq languaqe.

The second part describes syntax and qrammar of GEST simulation lanquaqe.

To describe SYNTAX and GRAMMAR of the GEST simulation proqramminq lanquaqe, the BACKUS - NAUR form is used.

In 1959, BACKUS proposed a formal notation for representinq the syntax of proqramminq lanquages which was simple, straiqht forward and relatively concise. His notation was expanded by NAUR in 1960, and was utilized for the formal description of the ALGOL 60 proqramminq lanquaqe.

The followinq are the basical rules used to describe GEST

1. The <non-terminal-name> bounded by the meta brackets < and >.

2. The terminal characters are printed in CAPITAL letters.

3. Looks like symbol is '::='.

Repetitor factor '6' in the statement

<IDENTIFIER>

::= <ALPHABETIC>6<ALPHANUMERIC>

Means: One alphabetic character immediatelly followed by from zero to six alphanumeric symbols.

5. The metasyntactic symbol '' <metaexpression> '' denotes repetition of the <metaexpression> within the curly brackets.

Definition Of GEST

(Partially Used In This Processor)

For an updated version of the language definition see Oren (1977).

Basic Elements

(1) <basic element>::=

<letter>|<digit>|<blank>

(2) <blank>::=

Symbol

(3) <symbol>::=

<letter>|<digit>|<continuation symbol>|<delimiter>

|<special symbol>

Letter

(4) <letter>::=

A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

Digit

(5) <digit>::=

0|1|2|3|4|5|6|7|8|9

Continuation Symbol

(6) <continuation symbol>::= &

Any character standing at the right of the continuation symbol on a given card is ignored. In IF statements which need

more than one card, one can omit the continuation symbol, since IF statements have semicolumns as special delimiters. No symbol, identifier, or constant can be broken by continuation symbol.

Special Symbol

(7) <special symbol>::=

,|.|:|::|'|+|-|*|/|_|=|&|(|)|<|>

Delimiter

(8) <delimiter>::=

<operator>|<bracket>|<separator>

(9) <operator>::=

<arithmetic operator>|

<relational operator>|

<declaration operator>|

<logical operator>

(10) <arithmetic operator>::= +|-|*|/|**

If A and B are vectors of the same size, then A+B, A-B, and A*B denote vector addition, subtraction, and multiplication. If in addition, k is a scalar constant or variable, then k*A denotes multiplication of the elements of the vector A by the scalar k.

If A and B are matrices conformable for addition and multiplication, then A+B, A-B, and A*B denote matrix addition, subtraction, and multiplication, respectively. If in addition, k is a scalar constant or variable, then k*A denotes multiplication of a matrix by a scalar.

(11) <relational operator>::=

LT|LE|GT|GE|EQ|NE|=|>|<

(12) <declaration operator>::= <is>

(13) <is>::= IS|ARE|=

The equality sign "=" is used as a declaration operator when it is used in program preamble or in system input, state, and output declaration.

(14) <logical operator>::= AND|NOT|OR|EOR

(15) <separator>::= ,|<blank>

To increase the readability of the program, "WHERE" can be used freely.

(16) <bracket>::= (|)|IF|THEN|ELSE|;

Comment

(17) <comment>::= ::{<basic element>}...

A comment card has the symbol "::" as a first nonblank character. Comment cards are not continued. But one can have any number of consecutive comment cards.

Examples:

:: THIS IS A COMMENT CARD

A=B*C :: THE REST OF THIS CARD IS A COMMENT

Constant

(18) <constant>::=

<integer>|<real>

<logical constant>|<literal constant>

(19) <integer>::= [+|-]<unsigned integer>

(20) <unsigned integer>::=

<digit>|<unsigned integer><digit>

(21) <real>::= [+|-]{<real without exponent>|

<real with exponent>}

(22) <real without exponent>::=

<unsigned integer>|

.<unsigned integer>|

<unsigned integer>.<unsigned integer>

(23) <real with exponent>::=

{<unsigned integer>|<real without exponent>}E<integer>

(24) <logical constant>::= TRUE|FALSE

TRUE is a constant that always has "true" value. FALSE is a constant that always has "false" value.

(25) <literal constant>::= '{<basic element>}...'

Since single quotation marks are used as delimiters of literal constants, any single quotation mark in a literal constant is represented by two consecutive quotation marks. Examples:

'THIS IS A LITERAL CONSTANT'

'IT''S A LITERAL CONSTANT'

Identifier

(26) <identifier>::=

<letter>|

<identifier>_|

<identifier><letter>|

<identifier><digit>

Examples:

INVENTORY_LEVEL

A_27

A27

Identifiers serve to identify variables, arrays, sets, labels and systems. The underscore symbol is used for readability of long identifiers. Identifiers may be up to sixteen characters long. If an identifier has more than six characters and/or if it has underscore, then the identifier is translated into a FORTRAN acceptable unique name.

(27) <label>::=

<label identifier>:

(28) <label identifier>::=

<identifier>|

<unsigned integer>

Variable

(29) <variable>::=

<scalar variable>|

<subscripted variable>|

<variable of a component system>|

<collected variable>|

<variable of a simulation run>

(30) <variable list>::=

<variable>|

<variable list>,<variable>|

(<variable list>)

(31) <scalar variable>::= <identifier>

(32) <subscripted variable>::=

<array identifier>[(<subscript list>)]

(33) <array identifier>::= <identifier>

(34) <subscript list>::=

<subscript>[,<subscript>]...

(35) <subscript>::= <arithmetic expression>

Subscript usage in GEST conforms to the subscripts in FORTRAN IV up to seven subscripts are allowed. A subscript has always a positive value. A subscript may be an unsigned integer or a variable.

Variable Of A Component System

(36) <variable of a component system>::=

<identifier> OF <identifier>

First identifier is the name of a variable, the second is the name of the component system.

Example:

A OF Z1

The same name may be used in different component systems to name different entities. Variable of a component system can only be used in simulation monitoring section of GEST programs and has always the same mode. There are two possible ways to refer to a variable of a component system in the simulation monitoring section. In the long form, one can say C OF Z1, for example. To use a shorter form, say t instead of C OF Z1, an equivalence declaration has to be made in the preamble of the program of the form:

EQUIVALENTS ARE (T, C OF Z1)

In this case, T is taken as a synonym of C OF Z1 in every simulation run.

Collected Variable

(37) <collected variable>::= <identifier>

collected variables are used with get operators to collect statistics. Basically, state, output, and input variables are collected. Get statements are declared in run time activities portion of GEST programs. In the following example, A, B, and C are collected variables:

GET FOR SYSTEM Z1 MAX(M), MIN(M), RANGE(M) AS A, B, C

The declared get statement generates appropriate statements to calculate maximum,minimum, and the range of the variable M and to store them in A, B, and C respectively. One can refer to a collected variable in the post run and post study portion of a simulation experiment.

Variable Of A Simulation Run

(38) <variable of a simulation run>::=

{<scalar variable>|

<subscripted variable>|

<collected variable>}

(<run identifier>)

(39) <run identifier>::= RUN ={<unsigned integer>|<variable>}

The use of run identifier is very helpful in referring to the results of several simulation runs.

Examples:

A(RUN=I)

A(J)(RUN=3)

A(J)(RUN=I)

Variable Of A Simulation Run

Declarations

(40) <declaration>::=

<input size declaration>|

<time unit declaration>|

<background type declaration>|

<type declaration>|

<equivalence declaration>|

<array declaration>

Input Size Declaration

(41) <input size declaration>::=

CARD IS <unsigned integer>[COLUMNS]

If the input size is not declared, it is assumed to be 72. If the specified number is less than 1 or greater than 80, 72 is assumed and a comment is printed. The GEST statements can be written starting any column of the input card.

Any card used to describe the system and/or to specify the simulation (i.e., cards other than the control cards for specific implementations, and data cards) can be continued by using the symbol "&". The symbol "&" need not be put at the last available position. The columns left after & are ignored.

Time Unit Declaration

(42) <time unit declaration>::=

TIME UNIT<is><time unit>[,<time unit>]...

[WHERE<time unit relationship>

[,<time unit relationship>]...]

(43) <time unit>::=

YEAR|MONTH|DAY|SHIFT|MINUTE|SECOND|MSECOND|USECOND|

<time unit>S|TIME UNIT

MSECOND and USECOND stand for millisecond and microsecond respectively.

Default value of time unit declaration is hour.

(44) <time unit relationship>::=

<time unit><is>

{<unsigned integer>|<real without exponent>}*<time unit>

Example:

DAY = 8.*HOURS

If time unit relationships are not declared the following assumptions are made:

1 YEAR = 12 MONTHS

1 MONTH = 30 DAYS

1 DAY = 24 HOURS

1 DAY = 3 SHIFTS

1 SHIFT = 8 HOURS

1 HOUR = 60 MINUTES

1 MINUTE = 60 SECONDS

1 SECOND = 1000 MSECONDS

1 SECOND = 1000000 USECONDS

Examples:

TIME UNIT IS DAY

TIME UNIT IS SECOND

TIME UNITS ARE DAY, HOUR WHERE DAY=24.*HOURS

Background Type Declaration

(45) <background type declaration>::=

<overall background type declaration>|

<local background type declaration>

(46) <overall background type declaration>::=

BACKGROUND [<is>]<type>

(47) <type>::=

INTEGER|REAL

(48) <local background type declaration>::=

BACKGROUND(OF SYSTEM<system name>)[<is>]<type>

If there is no explicit background type declaration, then the background mode is assumed to be real.

Logical variable used as a real cariable has the value 1., if it is TRUE, and the value 0., if it FALSE.

Type Declaration

(49) <type declaration>::=

<type>[<is>]<scalar variable list>|

<type>OF SYSTEM<system name>[<is>]<scalar variable list>

(50) <scalar variable list>::=

<scalar variable>|

<scalar variable list>,<scalar variable>

Examples:

INTEGERS ARE A,B,C

INTEGERS OF SYSTEM Z1 ARE S,T,U

REALS ARE D,K

Type declaration can be done for each component system separately.

Equivalence Declaration

The synonyms to designate a given variable are declared in equivalence declarations. The equivalence declaration may be in the preamble of a GEST program or at the beginning of a system declaration. All variables in the same equivalence group are of the same type. Namely, they have the background mode, or they are declared to have the same mode.

(51) <equivalence declaration>::=
<equivalence declaration>
EQUIVALENTS[ARE]<equivalence groups>

(52) <equivalence group>::=
<variable>,<variable>|
<equivalence group>,<variable>

(53) <equivalence groups>::=
(<equivalence group>)|
<equivalence groups>,(<equivalence group>)

Array Declaration

(54) <array declaration>::=
<fixed size array declaration>

Fixed Size Array Declaration

(55) <fixed size array declaration>::=
[<type>]<array><is><dimensioned variable list>

(56) <array>::= ARRAY|VECTOR|<array>S

(57) <dimensioned variable>::=

<array identifier>(<size declarator list>)

(58) <size declarator list>::=

<unsigned integer>|

<size declarator list>,<unsigned integer>

(59) <dimensioned variable list>::=

<dimensioned variable>|

<dimensioned variable list>,<dimensioned variable>

VECTOR can be used for a one dimensional array. The number of elements in the size declarator list of an array gives the dimension of an array. The unsigned integers declare the size of corresponding dimension(s).

Examples:

```
INTEGER VECTORS ARE A(10), B(12)
ARRAYS ARE 2(12,7), F(2,3,6)
REAL VECTOR VELOCITY(23)
```

Expressions

(60) <expression>::=

<arithmetic expression>|

<logical expression>|

<relational expression>

Statements

(61) <statement>::=

<control statement>|

<assignment statement>|

<data collection statement>|

<input statement>|

<output statement>|

<format statement>|

<initialization statement>|

<input scheduling statement>

Control Statement

(62) <control statement>::=

<if statement>|

<for statement>|

<call statement>|

<end statement>

If Statement

(63) <if statement>::=

<if clause>,THEN<assignment statement list>

[,ELSE<assignment statement list>];|

<if clause>,THEN<if statement>

[,ELSE<if statement>];

(64) <if clause>::= [EXCEPT]IF(<relational expression>)

An IF statement always terminates by a semi-column. More than one card can be used to punch an IF statement, without the continuation symbol. Instead of "EXCEPT IF", "UNLESS" can be used.

Examples:

IF ---, THEN ---;

IF(A > 0),THEN ---, ELSE ---;

For Statement

(65) <for statement>::=

[<if clause>]<for clause><assignment statement list>;|

[<if clause>]<for clause><if statement>

Examples:

```
IF(A>0, AND B=TRUE) FOR I = 1 TO 17
K(I) = 3 * I + SQRT(B(I));
FOR I = 1 TO N BY .25 UNTIL ABC
  ...
ABC:  ...;
```

(66) <for clause>::=

FOR<identifier>=<arithmetic expression>

[TO<arithmetic expression>][BACKWARDS]

[BY<arithmetic expression>][UNTIL<label>]

If the optional BY<arithmetic expression> is not given, then it is assumed to be 1.

Examples:

```
FOR I = 1 TO N BY .25
FOR I = 1 TO N
```

Call Statements

(67) <call statement>::=

CALL<subroutine name>[(<dummy argument list>)]

End Statement

(68) <end statement>::= END

Assignment Statement

(69) <assignment statement>::=

<arithmetic assignment statement>|

<logical assignment statement>|

<multiple assignment statement>

(70) <arithmetic assignment statement>::=

[<label>]<variable>=<arithmetic expression>|

[<label>]<variable>=<matrix expression>

(71) <logical assignment statement>::=

[<label>]<logical variable>=<logical expression>

(72) <multiple assignment statement>::=

[<label>][<variable>=]...<expression>

(73) <assignment statement list>::=

<assignment statement>|

<assignment statement list>,<assignment statement>

Data Collection Statements

(74) <data collection statement>::=

<get statement>|

<save statement>

Get Statement

Get statement is specified in run time activities section of a program. See explanation given for collected variables.

(75) <get statement>::=

GET<system identification>

{<get operator>(<identifier list>)}...

AS<identifier list>

[FROM<lower time limit>[<time unit>]]

[UNTIL<upper time limit>[<time unit>]]<if clause>

(76) <system identification>::=

(<system name>)|FOR SYSTEM<system name>

(77) <identifier list>::=

<identifier>|<identifier list>,<identifier>

(78) <get operator>::=

COUNT|MIN|MAX|RANGE|SUM|MEAN|STDEV|VARIANCE|SUMSQ|

HISTO(<lower limit>,<interval size>,<number of interval>)

(79) <lower limit>::= <integer>|<real>

(80) <number of interval>::= <integer>

(81) <lower time limit>::= <integer>|<real>

(82) <upper time limit>::= <integer>|<real>

Examples:

```
GET FOR SYSTEM Z1 MEAN(M) AS A
GET(Z1) MEAN(X,Y,Z) STDEV(X) AS A, B, C, D
GET(Z1) MEAN(M) AS A IF TIME > 25
GET(Z1) MEAN(M) AS A FROM 25 HOURS
```

Save Statements

Save statement is specified in the run time activities of a simulation run. It causes the collection of input, state, and output trajectories of component systems. It may be used to collect the trajectories of some specified variables of a given component system. Data collected by save statement can be printed or print-plotted.

(83) <save statement>::=

SAVE<system identification list>

{<trajectory>|<iso variable list>}

AS{<identifier list>}...

[<save specification>]

(84) <system identification list>::=

(<system name>[,<system name>]...)|

FOR SYSTEM<system name>[,<system name>]...

(85) <trajectory>::= TTRAJ|OTRAJ|ITRAJ|TRAJ

TTRAJ, OTRAJ, ITRAJ stand for time trajectory, output trajectory, and input trajectory, respectively. Time trajectory is the trajectory of the state variable. TRAJ which represents all three trajectories of a system and stands therefor for TTRAJ, OTRAJ, and ITRAJ.

(86) <iso variable list>::=

<iso variable>|<iso variable list>,<iso variable>

(87) <iso variable>::=

<input variable>|

<state variable>|

<output variable>

(88) <save specification>::=

⌈FROM<lower time limit>⌈<time unit>⌉⌉

⌈UNTIL<upper time limit>⌈<time unit>⌉⌉

(AT EVERY<sampling time specification><time unit>N

⌈<if clause>⌉

(89) <sampling time specification>::=

<unsigned integer>|<unsigned real>|<scalar variable>

Examples:

```
SAVE FOR SYSTEM Z1 TTRAJ AS A
SAVE(Z1) TTRAJ AS A
SAVE(Z1, Z2) TTRAJ AS A, B
SAVE FOR SYSTEM Z1 X, Y, Z AS A, B, C
SAVE(Z1) X, Y, Z AS A, B, C
SAVE FOR SYSTEM Z1 TTRAJ AS A EXCEPT IF C OF Z1 < 0
SAVE FOR SYSTEM Z1 TTRAJ AS A UNTIL 25
```

Initialization Statement

Only external input and state variables of a system need to be initialized by the user. The initial values of the output variables are calculated by the GEST object program by referring to the system output functions.

(90) <initialization statement>::=

INITIALIZE SYSTEMS TO ZERO|

INITIALIZE [<iso>](<system name list>) TO ZERO|

INITIALIZE (<system name>)<initialization list>;|

TO INITIALIZE [<iso>](<system name list>)<input statement>

SAVE LAST [<iso>][VALUES OF](<system name list>)

(91) <iso>::= {INPUT[S]|STATE[S]} [OF]

(92) <system name list>::=

<system name>|

<system name list>,<system name>|

<system name>(<identifier>)

FOR<identifier>=<unsigned integer>

TO<unsigned integer>)

(93) <initialization list>::=

<variable>[=<variable>]...=<constant>|

<initialization list>,<initialization list>

Statements starting with SAVE LAST cannot be used to initialize the systems for the first run. "FOR NEXT RUN" is specified for readability. Its declaration can be omitted. THE first example is to initialize all external inputs and state variables of all the systems.

Examples:

```
INITIALIZE SYSTEMS TO ZERO

INITIALIZE INPUT (Z1) TO ZERO

INITIALIZE INPUT OF (Z1) TO ZERO

INITIALIZE INPUT OF (Z(I) FOR I = 1 TO 5) TO ZERO

INITIALIZE STATE (Z1,Z2,Z4) TO ZERO

INITIALIZE (Z1)

A=5.  B=3.;

TO INITIALIZE STATE(Z5) READ

SAVE LAST STATE VALUES FOR NEXT RUN

SAVE LAST (Z1)

SAVE LAST (Z(I) FOR I = 1 TO5)

SAVE LAST (Z1,Z3,Z7) FOR NEXT RUN
```

Input Statement

(94) <input statement>::=

<input with format>|

<input without format>|

<free format input>

Input With Format

(95) <input with format>::=

READ (<device number>,<format identifier>)

<variable list>

(96) <device number>::=

<unsigned integer>|

<scalar variable>

An input with format statement has a corresponding format statement.

Input Without Format

(97) <input without format>::=

READ (<device number>) <variable list>

The variable list may have up to ten numeric variables. eight columns are spared for each variable. The assumed formats are e8 and i8.

Free Format Input

(98) <free format input>::=

READ DATA

Input Scheduling Statement

(99) <input scheduling statement>::=

<input schedule read statement>|

<input schedule generate statement>|

<input schedule data statement>

(100) <schedule clause>::=

[AT] EVERY <time list>

[START AT TIME = <time list>]

[UNTIL TIME = <time list>]

]TERMINATE <if clause>]

(101) <time list>::=

<integer>|<real>|

<time list>[<time unit>]|

<time list>,<time list>

Input Schedule Read Statement

(102) <input schedule read statement>::=

<input statement><schedule clause>;

Input Schedule Generate Statement

(103) <input schedule generate statement>::=

GENERATE INPUT(<system name>)<schedule clause>

In this case input function(s) have to be specified as part of the mathematical model of the appropriate component system.

Input Schedule Data Statement

(104) <input schedule data statement>::=

DATA INPUT(<system name>)<schedule clause>

{,<identifier>{,<constant>}...;}...ENDATA

Example:

```
DATA INPUT(Z1) EVERY HOUR START AT TIME=1 UNTIL TIME=5,
    ORDER,3,2,2,25,1;  BACKLG,,0,3,8,7,2; ENDATA
```

GEST Programs

(105) <gest program>::=

[<preamble>]

<mathematical model of the system>

<simulation study monitoring>

Preamble

(106) <preamble>::=

PREAMBLE

[<input size declaration>]

[<time unit declaration>]

[<background type declaration>]

[<type declaration>]

[<equivalence declaration>]

[<integration monitoring>]

The keywords COUPLING or SYSTEM used in a mathematical model of the system terminate the preamble. Integration monitoring is for the continuous component systems.

(107) <mathematical model of the system>::=

[<mathematical model of the coupling>]

{<mathematical model of a component system>}...

(108) <simulation study monitoring>::=

{<simulation run monitoring>}...

[<post study statements>]

(109) <simple system declaration>::=

SYSTEM<system name>[VERSION<version identifier>][<is>]

DISCRETE

[<backqround type declaration>]

[<type declaration>]

[<equivalence declaration>]

[<random variable declaration>]

[<array declaration>]

[<function declaration>]

[<data collection statement>]

<system input declaration>

<system state declaration>

<system state transition function declaration>

[<system output declaration>

<system output function declaration>]

(110) <system input declaration>::=

INPUT[S][<is>][CARTESIAN](<system input port list>)

(111) <system state declaration>::=

STATE[<is>][CARTESIAN](<system state list>)

(112) <system state transition function declaration>::=

STF{<system state transition function>}...

(113) <system state transition function>::=

<assignment statement>|

<if statement>|

<for statement>

(114) <system output declaration>::=

OUTPUT[S][<is>][CARTESIAN](<system output port list>)

(115) <system output function declaration>::=

OF {<system output function>}...

(116) <system output function>::=

<assignment statement>|

<if statement>|

<for statement>|

<system output function><sampling specification>

(117) <sampling specification>::=

AT EVERY<sampling time specification><time unit>

State variables can be subscripted variables. They can be referred to in state transition and output functions which may or not contain if or for clauses.

Output variables can be subscripted variables.

Appendix B Main Processor's Subroutines

The main purpose of the Appendix A is to document all subroutines and then qive the basical material for the later implementation.

The subroutines are listed twice. The first list is a brief subroutines description in alphabetical order.

The second time the GEST subroutines are qrouped into several categories by problem areas.

The detailed description of all subroutines is qiven by usinq a skeleton so the information on every subroutine is placed in the identical order.

B.1 LIST OF SUBROUTINES :

Subroutine ATRA :- Conversion And Storage Routine

Subroutine CHECK :- Initiator Of The Main Program Segments

Subroutine COPY :- Copying Routine

Subroutine DALIST :- Arithmetic Assignment Statements Generator

Subroutine ENDRUN :- FORTRAN Statements Generator

Subroutine GETFSC :- FORTRAN Subroutines Generator

Subroutine IEXCTN :- Program Execution Initiator

Subroutine INPUT :- Syntax Analyser For GEST Input Statement

Subroutine NMBR :- Numbers Analyser

Subroutine PRINT :- FORTRAN Output Generator

Subroutine PUNCH :- Punched Output Generator

Subroutine REREAD :- Re-read And Re-write Subroutine

Subroutine RUN :- Syntax Checker For Run Until Statement

Subroutine SINITL :- Syntax Checker For System Initialization Stmt.

Subroutine START :- Initiator Of Arrays

Subroutine STATE :- Syntax Analyser For System State Declaration

Subroutine SYSTEM :- Syntax Analyser For System Declaration

Subroutine TRANS :- Conversion Routine

Subroutine TS725 :- Randomizer

Subroutine TS728 :- Number Convertor

Subroutine TS729 :- Comment Analyser

Subroutine TS7210 :- Algorithm F72.8

Subroutine TS7311 :- Data Input Sequencer

Subroutine TS7313 :- GEST Print Statement Analyser

B.2 LIST OF SUBROUTINES BY PROBLEM AREAS :

Analysing Routines:

Subroutine INPUT :- Syntax Analyser For GEST Input Statement

Subroutine NMBR :- Numbers Analyser

Subroutine RUN :- Syntax Checker For Run Until Statement

Subroutine SINITL :- Syntax Checker For System Initialization Stmt.

Subroutine STATE :- Syntax Analyser For System State Declaration

Subroutine SYSTEM :- Syntax Analyser For System Declaration

Subroutine TS729 :- Comment Analyser

Subroutine TS7311 :- Data Input Sequencer

Subroutine TS7313 :- GEST Print Statement Analyser

Array Initialisation Routines :

Subroutine START :- Initiator Of Arrays

Conversion Routines :

Subroutine ATRA :- Conversion And Storage Routine

Subroutine TRANS :- Conversion Routine

Subroutine TS728 :- Number Convertor

Copying Routines :

Subroutine COPY :- Copying Routine

FORTRAN Statement Generators :

Subroutine DALIST :- Arithmetic Assignment Statements Generator

Subroutine ENDRUN :- FORTRAN Statements Generator

Subroutine GETFSC :- FORTRAN Subroutines Generator

Subroutine PRINT :- FORTRAN Output Generator

Program Initialisation Routines :

Subroutine CHECK :- Initiator Of The Main Program Segments

Subroutine IEXCTN :- Program Execution Initiator

Punched Output Generators :

Subroutine PUNCH :- Punched Output Generator

Randomizers :

Subroutine TS725 :- Randomizer

Re - Read And Re - Write Routines :

Subroutine REREAD :- Re-read And Re-write Routine

Entry RESB99 :- Restore The Internal Buffer

Entry SETB99 :- Set The Internal Buffer For Re-read/Rewrite

B.3 DETAILED SUBROUTINES DESCRIPTION :

Subroutine ATRA - Conversion And Storage Routine

Calling Sequence : CALL ATRA (ITAB,AB,LIA,ITA)

ITAB - counter of new names

LIA - pointer in the output array AB

ITA - number of significant digits to be stored

AB - output array of 80 words

Common Statement :

COMMON IDMI(80),IMDI(50,80)

IMDI - dummy array for this subroutine

Inputs : ITAB , LIA

Outputs : AB , ITA

Abstract :

The subroutine ATRA is used for storing of remembered values counter in the left hand side of a corresponding state transition function.

Restrictions :

- maximum number of generated remembered names is 9999

Error/warning Messages :

None

Calling Programs :

Main Program

Subroutine CHECK - Initiator Of The Main Program Segments

Calling Sequence : CALL CHECK(HASH,&147,&140,&148,&49,&52,&118,
&132,&273,&393,&93,&164)

Common Statement :

COMMON IDM(80),IDUM(50,80)

IDM - a dummy variable for this subroutine

IDUM - a dummy variable for this subroutine

Inputs : IHASH

Outputs : RETURN

Abstract :

The subroutine calls for execution the part of main program consisting the action corresponding to the GEST keyword. The decision is based on values of generated hash codes of the GEST keywords. If not match is found, an unlabeled RETURN statement is executed.

Restrictions :

-The number of GEST keywords included in the subroutine and their hash codes give a limited number of generated return codes

Error/warning Messages :

None

Subroutines Called :

None

Calling Programs :

Main Program

Subroutine COPY - Copying Routine

Calling Sequence : CALL COPY

Common Statement :

```
COMMON A,IDUMMY(50,80)
COMMON /BLOCK1/N1,MSO,SOURCE
```

A - vector of the source statement

N - pointer to the first nonblank column of the source statement

MSO - pointer to the GEST source statement being analysed

SOURCE - output array

Inputs : A , N , MSO

Outputs : SOURCE

Abstract :

The subroutine COPY is used to copy GEST source into the array SOURCE.

Restrictions :

-Logical length of 80 bytes per record

Error/warning Messages :

None

Subroutines Called :

None

Calling Programs :

Main Program

Subroutine DALIST - Arithmetic Assignment Statements Generator

Calling Sequence : CALL DALIST(&100)

Common Statement :

```
COMMON A,IOB(50,80)
COMMON /BLOCK0/LETTER,DIGIT,OPER,RELOP,BRACKT,DPOINT,COLON,COMMA,EQSIGN,
              UNDSC
COMMON /BLOCK5/MMSO,INSRCE(20,40)
```

A - vector of the source statement

IOB - a dummy variable for this subroutine

LETTER - letters A-Z,$

DIGIT - digits 1-0

OPER - arithmetic operators : +, -, *, /, **

RELOP - relationship operator >, <

BRACKT left and right hand parenthesis (,)

DPOINT - period .

COMMA - comma ,

EQSIGN - equal sign =

UNDSC - underscore punch card code 0-5-8

MMSO - pointer of a current line within INSRCE

INSRCE - array of a generated output (Subroutine DALIST)

Inputs : A

Outputs : MMSO , INSRCE

Abstract :

The subroutine will generate FORTRAN format arithmetic

assignment statements to initialize variables and constants of the component system.

Restrictions :

None

Error/warning Messages :

None

Subroutines Called :

None

Calling Programs :

Main Program

Subroutine ENDRUN - FORTRAN Statements Generator

Calling Sequence : CALL ENDRUN(IF,&147,&100,&501)

Common Statement :

```
COMMON A,OBJECT
COMMON /BLOCK0/LETTER,DIGIT,OPER,RELOP,BRACKT,DPOINT,COLON,COMMA,EQSIGN,
              UNDSC
COMMON /BLOCK9/IPOSTP,IPOSTR,JPOSTP,JPOSTR,IENDRN
```

A - vector of the source statement

LETTER - letters A-Z,$

DIGIT - digits 1-0

OPER - arithmetic operators : +, -, *, /, **

RELOP - relationship operator >, <

BRACKT left and right hand parenthesis (,)

DPOINT - period .

COLON - colon :

COMMA - comma ,

EQSIGN - equal sign =

UNDSC - underscore punch card code 0-5-8

IPOSTP - array of generated FORTRAN statements

JPOSTP - array of generated FORTRAN statements

IENDRN - a flag to indicate END of the RUN

Inputs : A

Outputs : IPOSTR, IENDRN

Abstract :

The program generates FORTRAN statements corresponding to the GEST POST RUN ACTIVITIES. Under the Post Run Activities are understood arithmetic assignment statements and/or GEST PRINT statements.

Restrictions :

- Postrun Activities are :
  - arithmetic assignment statements and/or
  - GEST PRINT statements

Error/warning Messages :

None

Subroutines Called :

Subroutine TS725 :- Randomizer

Subroutine TS7311 :- Data Input Sequencer

Calling Programs :

Main Program

Subroutine GETFSC - FORTRAN Subroutines Generator

Calling Sequence : CALL GETFSC(SWITCH)

Common Statement :

```
COMMON A,IOBJ(50,80)
COMMON /BLOCK0/LETTER,DIGIT,OPER,RELOP,BRACKT,DPOINT,COLON,COMMA,EQSIGN,
              UNDSC
COMMON /BLOCK3/VSN,IVSN/BLOCK4/TTABLE,TSTOCK
COMMON /BLOCK6/IDIMAR(66),INSQST(50,72)
COMMON /BLOCK7/IPRNFG,IPRNT(20,72),JPRINT,LSTDL
COMMON /BLOCK8/PHDING,I2B
COMMON /BLOCK9/IPOSTP,JPOSTP,IPOSTR,IENDRN
COMMON /BLCK10/IPUNCH,IEXCUT,ITHIS,IJOB
```

A - vector of the source statement

LETTER - letters A-Z,$

DIGIT - digits 1-0

OPER - arithmetic operators : +, -, *, /, **

RELOP - relationship operator >, <

BRACKT left and right hand parenthesis (,)

DPOINT - period .

COLON - colon :

COMMA - comma ,

EQSIGN - equal sign =

UNDSC - underscore punch card code 0-5-8

VSN - vector of the system names

IVSN - pointer to the last character inserted into the vector VSN

TTABLE - table of identifiers

TSTOCK - vector to store the original identifiers

IDIMAR - vector of the qenerated output - DIMENSION statement

INSQST - array to store INPUT SEQUENCING Statement

IPRNFG - a flaq to distiquish different PRINT statements of GEST

IPRINT - array of output to be printed

JPRINT - pointer of the headinq lines

LSTDL - pointer to the current line

PHDING - array of the headinq lines

IPOSTP - array of qenerated FORTRAN statements

JPOSTP - array of qenerated FORTRAN statements

IENDRN - a flaq to indicate END of the RUN

IPUNCH - a flaq for a request of the punched qenerated object deck

IEXCUT - a flaq to indicate the request to execute the qenerated progra

ITHIS - a number specifyinq a job to be executed

IJOB - initialization of the JOB CARD (University of Ottawa)

Inputs : SWITCH, A , PHDING, IJOBS , IPUNCH

Outputs : IPOSTP, IPOSTR, IPRNFG

Abstract :

The subroutine qenerates for each component system, just beinq processed, FORTRAN type subroutine. The subroutine's name is the GEST component system name with input variables, state variables and constants as the subroutine parameters.

Restrictions :

-A component system name does not exceed 6 characters

Error/warning Messages :

None

Subroutines Called :

Subroutine IEXCTN :- Program Execution Initiator

Subroutine PUNCH :- Punched Output Generator

Entry SETB99 :- Set The Internal Buffer For Re-read/Rewrite

Calling Programs :

Main Program

Subroutine IEXCTN - Program Execution Initiator.

Calling Sequence : CALL IEXCTN(SWITCH,AB)

Common Statement :

```
COMMON A,IOBJ(50,80)
COMMON /BLOCK3/VSN,IVSN/BLOCK4/TTABLE,TSTOCK
COMMON /BLOCK6/IDIMAR(66),INSQST(50,72)
COMMON /BLOCK7/IPRNFG,IPRNT(20,72),JPRINT,LSTDL
COMMON /BLOCK8/PHDING,I2B
COMMON /BLOCK9/IPOSTP,JPOSTP,IPOSTR,IENDRN
```

IOBJ - a dummy variable for this program

IDIMAR - vector of the generated output - DIMENSION statement

INSQST - array to store INPUT SEQUENCING Statement

IPRINT - array of output to be printed

JPRINT - pointer of the heading lines

LSTDL - pointer to the current line

PHDING - array of the heading lines

IPOSTP - array of generated FORTRAN statements

JPOSTP - array of generated FORTRAN statements

IENDRN - a flag to indicate END of the RUN

IPUNCH - a flag for a request of the punched generated object deck

Inputs : A , SWITCH, AB

Outputs : IPOSTP, IPOSTR

Abstract :

The subroutine is used to compile and execute the

generated object program. According to the option specification all jobs of the output or only one with a given sequence number is (are) executed.

Restrictions :

None

Error/warning Messages :

None

Subroutines Called :

Entry SETB99 :- Set The Internal Buffer For Re-read/Rewrite

Calling Programs :

Main Program

Subroutine INPUT :- Syntax Analyser For GEST Input Statement

Calling Sequence : CALL INPUT (&100)

Common Statement :

COMMON A,IDUM(50,80),/BLOCK1/N,MSO,SOURCE

A - vector of the source statement

SOURCE - array of GEST input statements

MSO - pointer to the GEST source statement which is being analyze

N - pointer to the first nonblank column of the source statemen

Inputs : A , SOURCE, N

Outputs : SOURCE

Abstract :

The subroutine checks the GEST input declaration for keyword INPUT. When the INPUT statement is recognized, letter C is inserted in column one in the GEST source program.

Restrictions :

None

Error/warning Messages :

None

Subroutines Called :

None

Calling Programs :

Main Program

Subroutine NMBR - Numbers Analyser

Calling Sequence : CALL NMBR(I,RN,INDATA)

I - pointer to the first nonblank symbol

RN - qenerated output

INDATA - a flaq to indicate integer or real number

Common Statement :

COMMON A, OBJECT(50,80)

A - vector of the source statement

Inputs : A , I

Outputs : RN

Abstract :

When during a statement scanning a number has been recoqnized the subroutine is called to reread that number and to assiqn its numerical value to a variable named RN. The numerical value should be unsiqned real (sinqle word) or integer. The subroutine every time when called ,qenerates a correspoding format(e.q. F**.*).

Restrictions :

-Sinqle precision is assumed

Error/warninq Messaqes : None

Subroutines Called :

Entry SETB99 :- Set The Internal Buffer For Re-read/Rewrite

Callinq Proqrams :

Main Proqram

Subroutine TS7311 :- Data Input Sequencer

Subroutine PRINT - FORTRAN Output Generator

Calling Sequence : CALL PRINT

Entry : CASE1A(NOFHLN,IF)

CASE2A(I,IF)

NOFHLN - number of heading lines

Common Statement :

```
COMMON A,OBJECT
COMMON /BLOCK7/IPRNFG,IPRINT(20,72),JPRINT,LSTDL
COMMON /BLOCK8/PHDING,I2B
COMMON /BLOCK9/IPOSTP,IPOSTR,JPOSTP,JPOSTR,IENDRN
```

A - vector of the source statement

IPRNFG - a flag to distiquish different PRINT statements of GEST

IPRINT - array of output to be printed

JPRINT - pointer of the heading lines

LSTDL - pointer to the current line

PHDING - array of the heading lines

IPOSTP - array of generated FORTRAN statements

JPOSTP - array of generated FORTRAN statements

IENDRN - a flag to indicate END of the RUN

Inputs : A

Outputs : IPRINT, IPOSTP, JPOSTP, PHDING

Abstract :

The program generates FORTRAN output statements corresponding to the source GEST PRINT statement. This

subroutine is devided into four parts, according the format of the GEST PRINT statement.

Restrictions :

-Maximum number of heading lines is nine (NMAX)

Error/warning Messages :

None

Subroutines Called :

Entry SETB99 :- Set The Internal Buffer For Re-read/Rewrite

Calling Programs :

Main Program

Subroutine PUNCH - Punched Output Generator

Calling Sequence : CALL PUNCH(SWITCH,AB)

Common Statement :

```
COMMON A,IOBJ(50,80)
COMMON /BLOCK3/VSN,IVSN/BLOCK4/TTABLE,TSTOCK
COMMON /BLOCK6/IDIMAR(66),INSQST(50,72)
COMMON /BLOCK7/IPRNFG,IPRNT(20,72),JPRINT,LSTDL
COMMON /BLOCK8/PHDING,I2B
COMMON /BLOCK9/IPOSTP,JPOSTP,IPOSTR,IENDRN
```

IOBJ - a dummy variable for this program

IDIMAR - vector of the generated output - DIMENSION statement

INSQST - array to store INPUT SEQUENCING Statement

IPRINT - array of output to be printed

JPRINT - pointer of the heading lines

LSTDL - pointer to the current line

PHDING - array of the heading lines

IPOSTP - array of generated FORTRAN statements

JPOSTP - array of generated FORTRAN statements

IENDRN - a flag to indicate END of the RUN

IPUNCH - a flag for a request of the punched generated object deck

Inputs : A , SWITCH, AB

Outputs : IPOSTP, IPOSTR, INSQST

Abstract :

The subroutine PUNCH is used to punch the object program

generated by the processor (FORTRAN IV). The flag to activate this subroutine is IPUNCH = 1 (user's option).

Restrictions :

None

Error/warning Messages :

None

Subroutines Called :

Entry SETB99 :- Set The Internal Buffer For Re-read/Rewrite

Calling Programs :

Main Program

Subroutine RUN :- Syntax Checker For Run Until Statement

Calling Sequence : CALL RUN (LABEL,IRUN,&100)

LABEL - alphanumeric string of 1-6 characters

IRUN - sequence number of the simulation run

Common Statement :

```
COMMON A,IDUMMY(50,80)
COMMON /BLOCK0/LETTER,DIGIT,OPER,RELOP,BRACKT,DPOINT,COLON,COMMA,EQSIGN,
              UNDSC
```

IDUMMY - a dummy variable for this subroutine

A - vector of the source statement

LETTER - letters A-Z,$

DIGIT - digits 1-0

OPER - arithmetic operators : +, -, *, /, **

RELOP - relationship operator >, <

BRACKT - left and right hand parenthesis (,)

DPOINT - period .

COLON - colon :

COMMA - comma ,

EQSIGN - equal sign =

UNDSC - underscore punch card code 0-5-8

Inputs : A

Outputs : LABEL

Abstract :

The subroutine analyses RUN UNTIL statement of the GEST

source program:

- for correct syntax of the RUN UNTIL statement
- recognizes label after the keyword RUN UNTIL

Restrictions :

-Maximum 6 symbols of a label after the RUN UNTIL

Error/warning Messages :

Error :

-Keyword RUN was not found

-Keyword UNTIL was not found

-Label exceeds 6 characters

Stops :

-STOP 132 : - see error messages

Subroutines Called :

None

Calling Programs :

Main Program

Subroutine SINITL :- Syntax Checker For System Initialization Stmt.

Calling Sequence : CALL SINITL(&100)

Common Statement :

```
COMMON A,IOB(50,80)
COMMON /BLOCK0/LETTER,DIGIT,OPER,RELOP,BRACKT,DPOINT,COLON,COMMA,EQSIGN,
              UNDSC
COMMON /BLOCK3/VSN,IVSN
```

A - vector of the source statement

IOB - a dummy variable for this subroutine

LETTER - letters A-Z,$

DIGIT - digits 1-0

OPER - arithmetic operators : +, -, *, /, **

RELOP - relationship operator >, <

BRACKT left and right hand parenthesis (,)

DPOINT - period .

COLON - colon :

COMMA - comma ,

EQSIGN - equal sign =

UNDSC - underscore punch card code 0-5-8

VSN - vector of the system names

IVSN - pointer to the last character inserted into the vector VSN

Inputs : A , VSN , IVSN

Outputs : NONE

Abstract :

The subroutine checks the syntax of GEST INITIALIZE SYSTEM .. Statement. A check is made if the system has been already defined.

Restrictions :

None

Error/warning Messages :

Error :

-Keyword INITIALIZE was not found

-System name is missing

-The system was not defined

Stops :

-STOP 273 : - see error messages

Subroutines Called :

None

Calling Programs :

Main Program

Subroutine START - Initiator Of Arrays

Calling Sequence : CALL START

Common Statement :

COMMON IA,OBJECT

IA - vector of the source statement

Inputs : OBJECT

Outputs : OBJECT

Abstract :

Subroutine START is called to initialize values to variables and arrays in unlabeled COMMON statements of the GEST processor.

Restrictions :

None

Error/warning Messages :

None

Subroutines Called :

None

Calling Programs :

Main Program

Subroutine STATE :- Syntax Analyser For System State Declaration

Calling Sequence : CALL STATE

Common Statement :

```
COMMON A,IDUM(50,80)/BLOCK1/ N,MSO,SOURCE
COMMON /BLOCK2/PSV,ISV
```

A - vector of the source statements

IDUM - a dummy variable for this subroutine

N - pointer to the first nonblank column of the source statement

MSO - pointer to the GEST source statement which is being analysed

SOURCE - array of GEST input statements

PSV - vector of state variables declared in the state variable declaration

Inputs : SOURCE, N , MSO

Outputs : SOURCE, PSV , ISV

Abstract :

The subroutine STATE checks the state variable declaration statement of a GEST source program:

- for the occurence of GEST keyword STATE
- for the occurence of GEST keyword CARTESIAN
- finds delimiters between names of the state variables

When the GEST State Variable Declaration is processed, the subroutine returns number of state variables declared (ISV) and pointers of PSV vector give locations of the state variables.

Restrictions :

-Maximum 100 pointers (PSV)

-State variable statement limited on one card only

Error/warning Messages :

Error :

-Error in the State Variable Declaration

Subroutines Called :

None

Calling Programs :

Main Program

Subroutine SYSTEM :- Syntax Analyser For System Declaration

Calling Sequence : CALL SYSTEM (&100)

Common Statement :

COMMON A,IDMM(50,80)

COMMON /BLOCK3/VSN,IVSN

COMMON /BLOCK1/N,MSO,SOURCE

A - vector of the source statement

IDMM - dummy array for this subroutine

VSN - vector of the system name

IVSN - pointer to the last character inserted into the VSN

N - pointer to the first nonblank column of the source statemen

MSO - pointer to the GEST source statemwnt which is being analyze

SOURCE - array of GEST input statements

VSNACS - array of all component system names

Inputs : A

Outputs : VSN , VSNACS

Abstract :

The subroutine checks the GEST source program for correct syntax of the keyword SYSTEM. The subroutine separates a component system name and stores it in the vector VSN. It checks if the source GEST system declaration has a keyword DISCRETE. If the keyword is missing an error message is printed and the processing is terminated.

Restrictions :

-Maximum 6 symbols for every component system name

-Maximum 50 component system names for the system

-Keyword DISCRETE must be present

Error/warning Messages :

Error :

-Keyword SYSTEM was not found

-Keyword IS missing

-Fatal error - keyword DISCRETE was not found

-Component system name exceeds 6 characters

Stops :

-STOP 172 : - missing keyword SYSTEM

- keyword DISCRETE was not found

- component system name exceeds 6 characters

Subroutines Called :

None

Calling Programs :

Main Program

Subroutine TRANS - Conversion Routine

Calling Sequence : CALL TRANS(ITAB,LI,STA)

ITAB - counter of new names

LI - pointer (first dimension) of the array OBJECT

STA - pointer (second dimension) of the array OBJECT

Common Statement :

COMMON IDM(80),OBJECT

IDM - a dummy variable for this subroutine

Inputs : ITAB , LI , STA

Outputs : OBJECT

Abstract :

This subroutine checks the value of the new names counter (ITAB=TTABLE(IHASH,5)) and if its value is greater then 9, the value is breaken down by digits and the digits are stored.

Restrictions :

-Maximum number (value of the counter of new names - ITAB) of new names 9999.

Error/warning Messages :

Error :

-Number of new names generated exceeds 9999

Stops :

-STOP 3 : - number of new names exceeds 9999

Calling Programs :

Main Program

Subroutine TS725 - Randomizer

Calling Sequence : CALL TS725 (HASH,NW,LS,WORD,NIS)

Entry : REP725(HASH,NW,LS,WORD,NIS)

HASH - qenerated hash code for a qiven input string

NW - number of sinqle (4 bytes) computer words to hold the input string in STOCK

LS - location in the table where the input string is stored

WORD - vector of 4 sinqle words

NIS - number of characters of the input string

Common Statement :

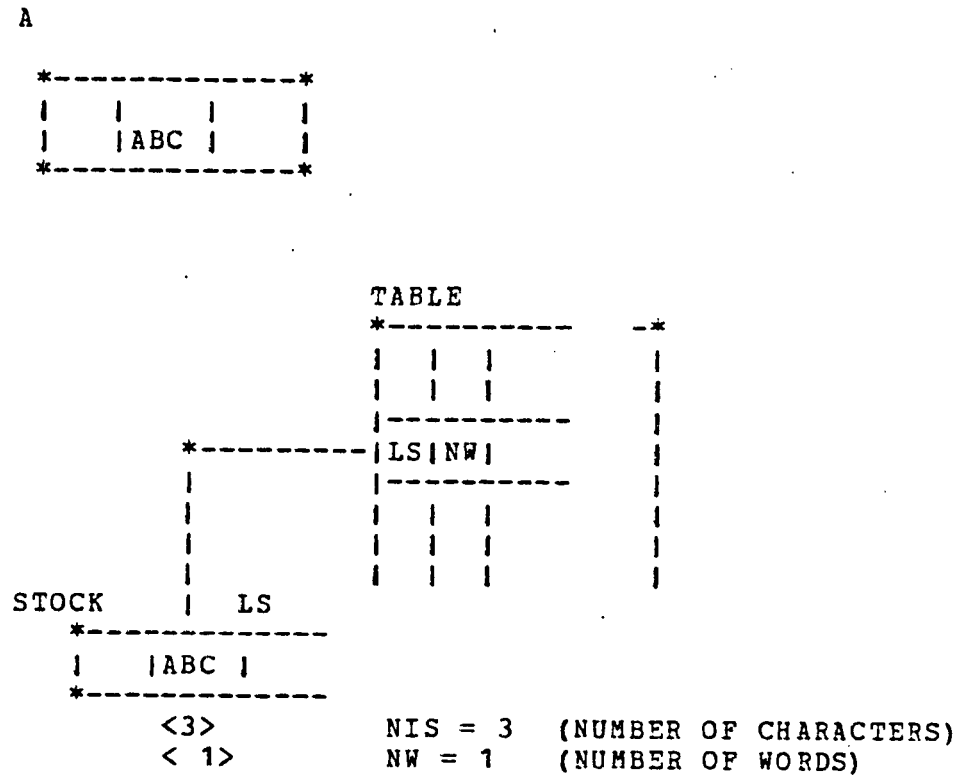
COMMON A,IDUM(50,80)

A - vector of the source statement

IDUM - dummy array for this subroutine

Inputs : A

Outputs : HASH , NW , NIS , WORD



Abstract :

For a given input string A (passed throught COMMON statement) the program generates a hash code. The hash code is calculated as follows: The numerical equivalent of the input is divided by TS - table size, which is the maximum number allowed for hash codes. Then the hash code = remainder + 1.

Restrictions :

-Input string is up to 16 characters

Error/warning Messages :

Error :

-The following input exceeds the maximum length 16 characters

Stops :

-STOP 3 : - the input exceeds 16 characters

Subroutines Called :

Entry SETB99 :- Set The Internal Buffer For Re-read/Rewrite

Calling Programs :

Main Program

Subroutine ENDRUN :- FORTRAN Statements Generator

Subroutine TS728 - Number Conversion

Calling Sequence : CALL TS728 (N1,N2)

Common Statement :

COMMON A,IDUM(50,80)

A - vector of the source statement

IDUM - a dummy variable for this subroutine

Inputs : A , N1 , N2

Outputs :

Abstract :

This subroutine detects integer numbers and floating point numbers with or without exponent E.

Restrictions :

-The subroutine accepts single precision only

Error/warning Messages :

Error :

-See listing of the subroutine

Stops :

- STOP 5 : -maximum precision is exceeded

Subroutines Called :

Entry SETB99 :- Set The Internal Buffer For Re-read/Rewrite

Calling Programs :

Main Program

Subroutine TS729 - Comment Analyser

Calling Sequence : CALL TS729(ITSF,*,*)

Common Statement :

COMMON A,IDUMY/BLOCK1/N,MSO,SOURCE

IDUMY - A DUMMY VARIABLE FOR THIS SUBROUTINE

A - vector of the source statement

N - pointer to the first nonblank column of the source statement

MSO - pointer to the GEST source statement being analysed

SOURCE - output array

ISTF - a flag of state transition functions

Inputs : A , N , MSO , SOURCE, ISTF

Outputs : SOURCE

Abstract :

The program analyses GEST source program for an occurence of GEST comment symbol (::) and replaces it by letter C in the first column of an object program. In the addition the source program is checked for blank card.

Restrictions :

-See definition of GEST comment statements

Error/warning Messages :

Error :

-See listing of the subroutine

Stops :

-STOP 729 : - incorrect GEST comment statement

Subroutines Called :

None

Calling Programs :

Main Program

Subroutine TS7210 - Algorithm F72.8

Calling Sequence : CALL TS7210(N1,N2,N3,NIN)

Common Statement :

COMMON A,OBJECT(50,80)

A - vector of the source statement

Inputs : A , N1 , N2 , N3

Outputs : NIN

Abstract :

The program calculates number of input values for the INPUT SCHEDULE DATA statement of GEST:

- the values of N1, N2, N3 corresponding to the following format of the INPU SCHEDULE DATA statement are read :

... AT EVERY <N1> <time unit> /START AT TIME=<N2>/

UNTIL TIME=<N2> ...

Restrictions :

-Numbers N1, N2, N3 are unsigned integers or reals

- N1 should be positive

- N2 should be nonnegative

- N3 should be positive and greater then N2

-Number of input values must be the same for all identifiers of the same INPUT DATA SCHEDULE Statement

Error/warning Messages :

None

Subroutines Called :

None

Calling Programs :

Main Program

Subroutine TS7311 :- Data Input Sequencer

Subroutine TS7311 - Data Input Sequencer

Calling Sequence : CALL TS7311(IDN)

Common Statement :

```
COMMON A,OBJECT
COMMON /BLOCK0/LETTER,DIGIT,OPER,RELOP,BRACKT,DPOINT,COLON,COMMA,EQSIGN,
            UNDSC
COMMON /BLOCK6/IDIMAR(66),INSQST(50,72)
```

A - vector of the source statement

LETTER - letters A-Z,$

DIGIT - digits 1-0

OPER - arithmetic operators : +, -, *, /, **

RELOP - relationship operator >, <

BRACKT left and right hand parenthesis (,)

DPOINT - period .

COLON - colon :

EQSIGN - equal sign =

UNDSC - underscore punch card code 0-5-8

IDIMAR - vector of the generated output - DIMENSION statement

INSQST - array to store INPUT SEQUENCING Statement

Inputs : A

Outputs : INSQST, IDIMAR

Abstract :

The program analyses the INPUT DATA SCHEDULE Statement and calculates (using subroutine TS7210 number of the input values

correspoding to the given time parameter of that statement.

Restrictions :

-Numbers N1, N2, N3 are unsigned integers or reals

- N1 should be positive

- N2 should be nonnegative

- N3 should be positive and greater then N2

-Number of input values must be the same for all identifiers of the same INPUT DATA SCHEDULE Statement

Assumptions :

-If N1 is not given N1=1

-If N2 is not given N2=1

Error/warning Messages :

None

Subroutines Called :

Subroutine NMBR :- Numbers Analyser

Entry SETB99 :- Set The Internal Buffer For Re-read/Rewrite

Subroutine TS7210 :- Algorithm F72.8

Calling Programs :

Main Program

Subroutine TS7313 - GEST Print Statement Analyser

Calling Sequence : CALL TS7313(IF)

Common Statement :

```
COMMON A,OBJECT
COMMON /BLOCK0/LETTER,DIGIT,OPER,RELOP,BRACKT,DPOINT,COLON,COMMA,EQSIGN,
              UNDSC
```

A - vector of the source statement

LETTER - letters A-Z,$

DIGIT - digits 1-0

OPER - arithmetic operators : +, -, *, /, **

RELOP - relationship operator >, <

BRACKT left and right hand parenthesis (,)

DPOINT - period .

COLON - colon :

COMMA - comma ,

EQSIGN - equal sign =

UNDSC - underscore punch card code 0-5-8

Inputs : A

Outputs : OBJECT

Abstract :

The program TS7313 generates FORTRAN statements for GEST source PRINT statement:

- general form of the GEST PRINT statement :

  PRINT n HEADING LINE(s) ...1a

PRINT FOR FIRST PAGE n HEADING LINE(s) ...1b

PRINT <variable list> /number of variables.LE.8/ ...2a

PRINT <variable list> /number of variables .GT.8/ ...2b

- CASE 1A : The heading line(s) will be printed on top of every new page
- CASE 1B : The heading line(s) is(are) printed once only (top of the first page only)
- CASE 2A : TIME and names of variables are printed across the page (FORMAT E11.4/I10)
- CASE 2B : TIME and names of variables are printed together with their values (FORMAT E11.4/I10)
  their values (FORMAT E11.4/I10, VARIABLE=VALUE)

Restrictions :

-The length of a variable does not exceed 10 characters

-Maximum number of heading lines is nine

-Maximum number of variables in GEST PRINT statement is 100

Error/warning Messages :

None

Subroutines Called :

Entry SETB99 :- Set The Internal Buffer For Re-read/Rewrite

Entry CASE1A :- Routine To Egnerate Heading Lines

Entry CASE2A :- Routine To Print TIME And Variables

Calling Programs :

Main Program

Appendix C Bibliography

Abbreviations

ACM : Association For Computing Machinery

AICA: Association Internationale Pour Le Calcul Analogique

GEST: General System Theory Implementor

SCI : Simulation Councils, Inc.

Allen, F. E., "Program Optimization", Annual Review In Automatic Programming , Vol. 6, Part 5, 1970, Page 239-308.

Baker, F. T., And Mills, H. D., "Chief Programmer Teams", Datamation , December 1973.

Baker, F. T., System Quality Through Structured Programming , Proc. FJCC (1972), Page 339-343 .

Boies, S. J., "User Behavior On An Interactive Computer System", IBM System Journal , Vol. 13, No. 1, 1974, Page 2-18.

Booth, T. L., Sequential Machines And Automata Theory , John Wiley & Sons, New York, 1967.

Brinch, Hansen, P., "Concurrent Programming Concepts", ACM Computing Surveys , Vol. 5, No. 4, December 1973, Page 223-245.

Brown, G. D., System /360 Job Control Language , John Wiley & Sons, New York, 1970.

Brown, P. J., "Programming And Documenting Software Projects", ACM Computing Surveys, 6,4 (December 1974), Page 213-220.

Buxton, J. N., And Randell, B., "Software Engineering Techniques", NATO Science Committee , Rome, 1964.

Chapin, N., 360/370 Programming In Assembly Language , Mc Graw - Hill Book Company, New York, 1973.

Chau, C. K., Algorithms For Hash Coding And Document Classification , Doctoral Dissertation, University Of Illinois, 1972.

Cocke, J. And Schwartz, J. T., Programming Languages And Their Compilers ,Courant Institute Of Mathematicalsciences, New York University, 1970.

Cress, Dirksen, Graham, FORTRAN IV With WATFOR And WATFIV , Prentice - Hall, 1970.

Day, W. H. E., "Compiler Assignment Of Data Items To Registers", IBM Systems Journal , Vol.9, No. 4, 1970.

Denning, P. J., "Is It Not Time To Define Structured Programming", ACM Operating Systems Review , Vol.8, No. 1, Jan. 1974.

Dijkstra, E. W., "The Structure Of The Multiprogramming System", ACM 11 (1968) , Page 341-346.

Dijkstra, E. W., "Notes On Structured Programming", Technical University, Eindhove, The Netherlands, 1970.

Donaldson, J. R., "Structured Programing", Datamation ,

December 1973.

Doran, R. W., And Tate, G., An Approach To Structured Programming , Massey University, New Zealand, 1972.

Dutton, J. M. And Starbuck, W. H., Computer Simulation Of Human Behavior , John Wiley & Sons, New York, 1971.

Elson, M., And Rake, S. T., "Code -Generation Technique For Large-Language Compilers", IBM System Journal , Vol. 9, No. 3, 1970.

Emshoff, D. And W., Sisson, R. And J., Design And Use Of Computer Simulation Models , The Macmillan Company, New York, 1970.

Gaver, D. P., And Thompson, G. L., Programming And Probability Models In Operations Research , Brooks/ Cole Publishing Co., Monterey, California, 1973, Page 560-599.

Ginzburg, A., Algebraic Theory Of Automata , Academic Press, 1968.

Gordon, G., System Simulation , Prentice - Hall, 1969.

Gries, D., Compiler Construction For Digital Computers ,John Wiley & Sons Inc., Toronto, 1971.

Handford, K. V., "Automatic Generation Of Test Cases" , IBM System Journal 9, 4(1970), Page 242-257.

Harrison, M. C., Data-structures And Programming , The Courant Institute Of Mathematical Sciences, New York, Jan. 1971.

Horning, J. J., And Randell, B., "Process Structuring", ACM Computing Surveys , Vol. 5, No. 1, March 1973,page 5-30.

IBM, IBM System /360, Assembler Language , Form No. GC 28-6514.

IBM, Bibliography On Simulation ,Form Number 320-0924, 1964.

IBM, IBM 1403 Printer Component Description , GA24-3073-7, 1403-03.

IBM, IBM System /360, Concepts And Facilities , Form No. GC 28-6535.

IBM, IBM System /360 And System /370, FORTRAN IV Language , Form No. GC 28-6515-8.

IBM, IBM System /360 Operating System, FORTRAN IV (G And H) Programmer's Guide , Form No. GC 28-6817-2.

IBM, IBM System /360 Operating System: Job Control Language Reference Manual , Form No. GC 28-6704-3.

IBM, IBM System /360, Linkage Editor And Loader , Form No. GC 28-6538.

IBM, IBM System /360, Principles Of Operation , Form. No. GA 22-6821-8.

IBM, IBM System /360, Supervisor And Data Management Services , Form No. GC 28-6646.

IBM, IBM System /360, System Summary , Form No. A 22-6810.

Ingerman, P. Z., A Syntax Oriented Translator , Academic Press, New York, 1966.

Katzan, H., Jr., Advanced Programming , Van Nostrand Reinhold Co., New York, 1970.

Katzan, H., Jr., Operating Systems A Pragmatic Approach , Van Nostrand Reinhold Co., New York, 1973.

Katzenelson, J., "Documentation And The Management Of A Software Project", Software Practise And Experience , 1,2,

(April 1971), Page 147-157.

Kernighan, B. W., And Plauger, P. J., "Programming Style: Examples And Computer Examples", ACM Computing Surveys , 6,4 (December 1974), Page 303-319.

Kiviat, P. J., Villanueva, R., Markovitz, H. M., The SIMSCRIPT II , Prentice - Hall, Santa Monica, 1968.

Knuth, D. E., " The Art Of Computer Programming ,1,2,3 Addison-Wesley Publishing Company, New York, 1968-1973.

Knuth, D. E., "Structured Programming With Go To Statements", ACM Computing Surveys , 6,4(December 1974), Page 261-302.

Lee, J. A. N., Computer Semantics , Van Nostrand Reinhold Co., New York, 1972.

Liskov, B. H., "Guidelines For The Design And Implementation Of Reliable Software Systems", Mitre Technical Report Number 2345 .

London, K. R., Techniques For Direct Access , Auerbach, Philadelphia, 1973.

Macgovan, R. A., And Henderson, R., CDP Review Manual: A Data Processing Handbook , Auerbach Publishers Inc., 1972.

Mayers, G. J., "Composite Design: The Design Of Modular Programs", IBM Poughkeepsie Laboratory Report , 29 Jan. 1973.

Mc Cracken, D. D., "Revolution In Programming: An Overview", Datamation ,19,12(December 1973), Page 50-52.

Mc Keeman , W. M., A Compiler Generator , Prentice - Hall, 1970.

Miller, E. F., Jr., And Lidamood, G., "Structured Programming: Top-Down Approach", Datamation , December 1973.

Mills, H., "Top - Down Programming In Large Systems", IBM FSD Report .

Naur, P., "An Experiment In Program Development", BIT , 12,3(1972), Page 347-365.

Naylor, T. H., Balintfy, J. L., Burick, D. S., And Chu, K., Computer Simulation Techniques , John Wiley, New York, 1966.

Naylor, T. H., Computer Simulation Experiments With Models Of Economic Systems , John Wiley & Sons, Inc., Toronto, 1971.

Oren, T. I., Systems Approach In Ecological System Simulation , Proceedings Of The 4th Annual International Symposium Of The American Society For Cybernetics, Washington, D. C., October 8-9, 1970.

Oren, T. I., A Basis For The Taxonomy Of Simulation Languages , Proceedings Of The 1971 Summer Computer Simulation Conference, Boston, Mass., July 19-21, 1971.

Oren, T. I., GEST: General System Theory Implementor (A Combined Digital Simulation Language) , Doctoral Dissertation, The University Of Arizona, 1971.

Oren, T. I., GEST: A Combined Digital Simulation Language For Large Scale Systems , Proceedings Of The Tokyo 1971 AICA Symposium Of Complex Systems, Tokyo, Japan, Sept. 3-7, 1971, B-1/1-b1/4.

Oren, T. I., Simulation Of Time - Varying Systems , Proceedings Of The International Congress Of Cybernetics And Systems,

Oxford, England, August 28 Sept.1, 1972.

Oren, T. I., Implementation Of Synchronous Variables And Another Parallel Computation Features Of GEST , Proceedings Of The 1972 Summer Simulation Conference, San Diego, Calif., June 14-16, 1972.

Oren, T. I., Revised Report On GEST :
A Simulation Language For Large Scale Combined Discrete / Continuous Systems
Technical Report: 77-03, Computer Science Department, University Of Ottawa, Ottawa, Ontario, March 1977

Pollack, B. W., Compiler Techniques , Auerbach, New York, 1972.

Rain, M., "Two Unusual Methods For Debugging System Software", Software Practise And Experience ,3,1(January 1973), Page 61-63.

Stevens, W. P., Myers, G. J., And Constantine, L., "Structured Design", IBM Systems Journal , Vol. 13, No.2, 1974.

Stone, H., Introduction To Computer Organization And Data Structure , Mcgraw - Hill Book Co., New York, 1970.

Tassel, D., Computer Security Management , Prentice - Hall, Inc., Englewood Cliffs, New Jersey, 1972.

Thomas, A. Jr., System /360 Programming , Rinehart Press, San Francisco, 1971.

Ton, J. N., Applied Automata Theory , Academic Press, 1968.

Ulrich, E. G., "Serial/ Parallel Event Scheduling For The Simulation Of Large Systems", Proceedings-1968, ACM National Conference , Page 279-287.

Walsh, D. A., A Guide For Software Documentation ,

Mc Graw - Hill, New York, 1969.

Willis, R. R., "Structured Model Development Techniques", Simulleter , Volume 5, Number 4, July 1974, Page 41-52.

Wirth, N., "On The Composition Of Well- Structured Programs", ACM Computing Surveys , ,6,4 (December 1974), Page 247-260.

Yohe, J. M., "An Overview Of Programming Practices", ACM Computing Surveys , 6,4 (December 1974), Page 221-246.

VITA

Name: Ladislav Joseph Bleha

Born: Praque, Czechoslovakia

August 4, 1940

Education: Postqraduate Studies In Automation,
Faculty Of Electrical Enqineerinq,
Ceske Vysoke Uceni Technicke,
Praque, Czechoslovakia

Dipl. Enq.,
Faculty Of Mechanical Engineerinq,
Ceske Vysoke Uceni Technicke,
Praque, Czechoslovakia