A SUPERVISOR TO MONITOR MULTIPLE SIMULATORS

by

ROBERT NORMAN McRAE

B.SC., University of British Columbia, 1970

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

M.Sc.

in the Department
of
Computer Science

We accept this thesis as conforming to the
required standard

THE UNIVERSITY OF BRITISH COLUMBIA
March 1972

Department of Computer Science

The University of British Columbia
Vancouver 8, Canada

Date April 13, 1972

ABSTRACT

The problem treated in this thesis is to create a system to monitor multiple interacting simulators. The problem is encountered in an attempt to simulate urban growth. The resulting system has a supervisor which controls all of the components of the system, namely the I/O routines, the graphics routine, the command routines, and the simulators. The main task of the supervisor is to display whichever data values the user desires. This involves executing some simulators, and extracting the data values from the data base. The extraction process first finds an association path between the files in the data base in order to relate the variables being displayed. Then using the association path the physical values are extracted from the data base. The data values are then passed to the graphics routine to be displayed for the user.

TABLE OF CONTENTS

chapter contents page

TABLE OF FIGURES

ACKNOWLEDGEMENTS

I am deeply indebted to Dr. J.L. Parker for his able assistance and direction in helping me to complete this thesis, and for spending generous portions of his time discussing its development. I would also like to take this opportunity to commend Brian Jervis, Doug Ash, Zafar Rashid and Joel Yan for their fine programming effort in various sections of the overall system. I would also like to thank Dr. F. Nake for his constructive criticisms.

CHAPTER I THE PROJECT - AN OVERVIEW

§ A. introduction

The problem treated in this thesis is to create a system to monitor multiple simulators. The system is given a set of simulators $\{s_1,..,s_n\}$, where simulation is defined as follows:

> A simulation of a system is the operation of a model or simulator which is a representation of the system. The model is amenable to manipulations which would be impossible, too expensive, or impractical to perform on the entity it portrays. The operation of the model can be studied and, from it, properties concerning the behaviour of the actual system can be infered [12, p.2].

The need for such a system arose out of the desire to simulate urban growth. Hence the involvement with the Inter-Institutional Policy Simulator (hereafter refered to as IIPS) group. The IIPS group consists of representatives from the City of Vancouver, the Vancouver Regional District, and the University of British Columbia. There is also some participation from the Municipal Affairs Department in Victoria, and the Urban Planning Department in Ottawa. The goal of the project is to create a simulation model of the Vancouver Lower Mainland. The total model has been split into various submodels, namely population and demographic, economic, transportation, land utilization, health systems, pollution, human ecology, resources and public services. Each of these submodels are capable of forecasting certain results given the appropriate data from previous years. The director of the IIPS project states its goal in these words:

> Its initial aim was to develop a simulation model of man/environment interaction in the Lower Mainland of British Columbia, and this was hopefully to be only one of a series of steps leading to more responsive decisions affecting the quality of life in an urban region [6].

The following description indicates the approach to the problem taken in this thesis. The model of urban growth was divided into interacting submodels. In order to make the simulators easier to monitor they are forced to use common routines for communication with a user, the data base management, and output generation. At the heart of the system there is a central controller - the supervisor - which decides when to execute the various components in the system. The system is represented schematically in figure 1.

The components of the system were designed so that they possessed the following attributes. The execution of the system can be understood by persons who may not be familiar with computing equipment, and it is possible to explain to a user in a short time period how to operate the basic parts of the system. The system has a set of very easily understood commands by which the user corresponds with the system. The data management routines permit the simulators to reference the data base with ease. The graphical output routine is device independent, and is capable of displaying various types of graphs. The user need not worry about running the simulators to create the appropriate data before he wants some graphical output - he merely indicates the desired graph. The supervisor, through the DISPLAY command, does all of the work in order to generate the data values to pass to the graphical output routines. This means that the simulators are effectively hidden from the user. The system allows certain variables, called policy variables, to act as parameters to the simulators, for example interest rate or population growth. The values of all of the policy variables are kept in the policy vector,

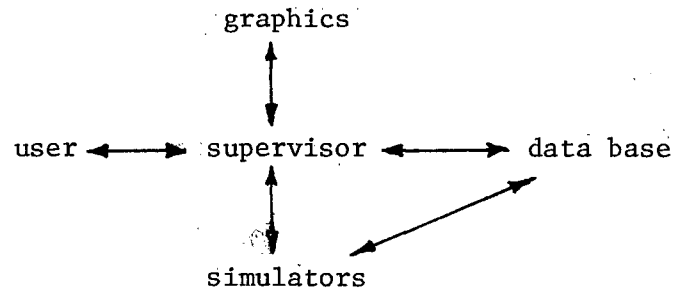


figure 1: representation of system

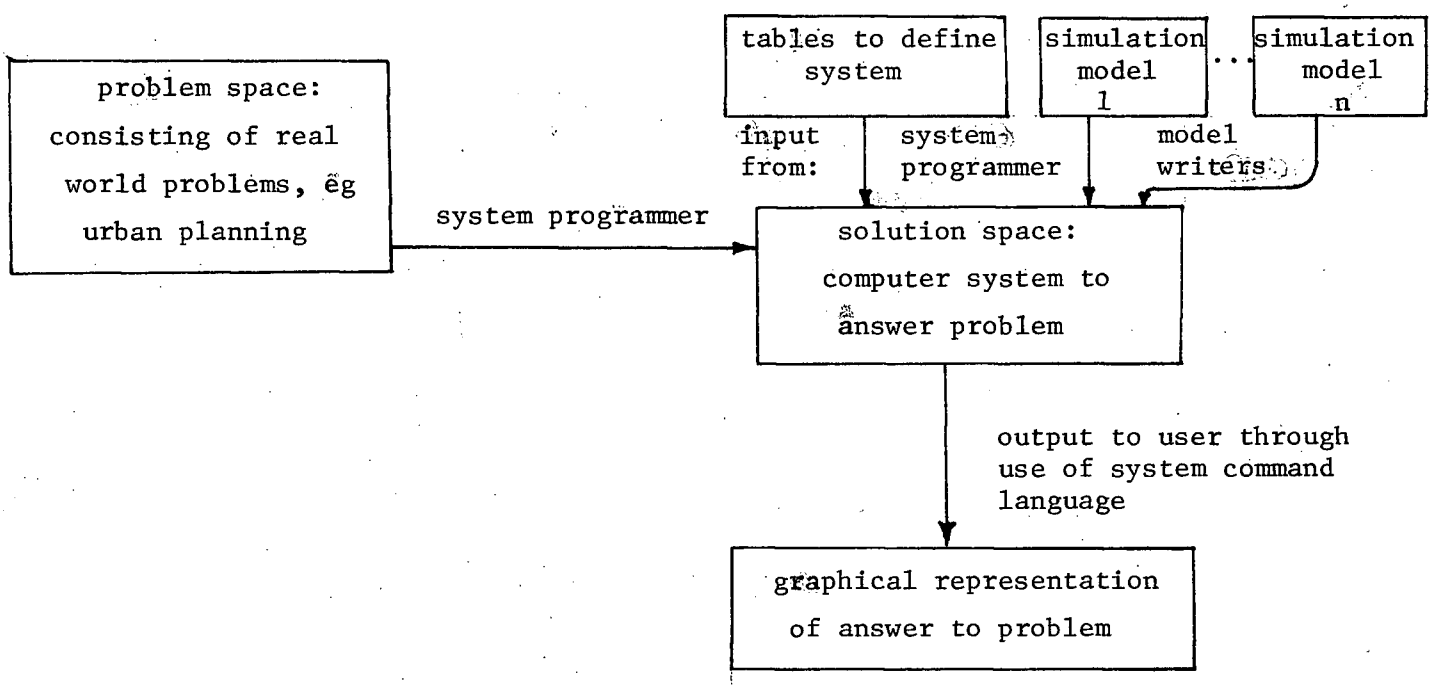


figure 2: problem domain

which is passed to the simulators when they are loaded. The system allows the policy variables to be changed at any time by the user. The system is table driven so that it is independent of which simulators it is executing. These tables describe the state of the entire system, and are very easy to input. Figure 2 sums up the domain of the problem.

My contribution to the system includes design and implementation of the supervisor. The most powerful command in the system is the DISPLAY command; hence the bulk of this thesis is devoted to its description. The most difficult algorithm in the supervisor is the generalized extraction process that is responsible for retrieving the correct output of the simulators to pass to the graphics supervisor (the routine to monitor the graphical output).

The rest of CHAPTER I has a description of the overall system into which the supervisor fits. CHAPTER II has an overview of the supervisor. CHAPTER III describes the supervisor in detail. It describes the algorithms developed in order to implement the supervisor - especially those used in the DISPLAY routine. CHAPTER IV has some suggestions for further extensions to the system.

§ B. <u>comparison with other systems</u>

In 1968 an urban model called BASS [5], of a similar nature to IIPS, was developed for the computer (IBM 7094). Its objective is stated as follows:

> The BASS Model is designed to serve as an analytical tool for exploring possible alternative impacts of major public and private investment decisions upon land utilization [5, p.24].

BASS used some of the same design criteria as our system. The BASS

report states that the system should be flexible: "the program should be easy to run with different values for the various coefficients and parameters... and it should be easy to make future changes in the various submodel programs...in particular, changing a submodel should not require a major redesign of the entire simulation program." [5, p.418]. Since the simulators could not all fit into core at once they were sectioned using the OVERLAY facility. Our system makes use of the advanced computer technology which allows dynamic loading of programs from disc files to memory. The BASS simulation stored in a COMMON area data that needed to be passed between simulators. There were no common I/O routines and no graphical output capability. The BASS simulation did not have a supervisor - there is a linear sequence of calls to simulators which are all arranged in a specific order. There was no attempt at creating a general framework into which any set of simulators may operate.

Ingram [7] describes a system similar in structure to the BASS system. It does not make use of a supervisor to control the simulators, but has forced all of the simulators to act as in-core subroutines to a main program. This Detroit system only allows for the output of tables.

The paper by Myers [11] on general information management systems describes design features that should be encorporated into any information management system. The data base definition and data base content should be flexible and easy to modify. The system must "logically define data so that it can be processed and used in a common way, and... store data based on this definition" [11, p.299]. The physical storage of data should have key values for retrieval. Our system encorporates all of these ideas.

Lipner advocates in [10] some desired properties of computer-based urban information systems based on experience with the Boston Cities Model developed at M.I.T. He states that "any really useful urban information system must produce graphical as well as tabular output" [10, p.525]. He advocates the use of common I/O routines, and the use of man-machine communication capable of being used by people who are not computer-oriented.

Some of the features of CODAS [4] are similar to those in our system. The CODAS system is interested in "retrieving data from files, performing some basic operations on them, and displaying them in any of several output formats" [4, p.67]. Our system has the added complication that it is simulators which produce the data files. The CODAS system has a command language that consists of setting up parameters and keywords into a 'request packet'. The data files have a restricted format: one key value followed by only two additional fields. There was no attempt to generalize their system.

LISTAR [1] is an "on-line interactive system which permits a user to define, search, modify, and cross associate data files " [1, p.313]. LISTAR has a logical description of its files. LISTAR permits the user to create an association between any two files, whereas in our system the retrieval routine is responsible for setting up the association. LISTAR makes use of a supervisor in very much the same way as our system. The LISTAR supervisor accepts commands in format-free structure and passes the parameters to subroutines that correspond to the command names. LISTAR stores the data base in an auxilary storage. The LISTAR files may be simple (linear) or complex (hierarchical), whereas in our system the structure of the files may only be simple. LISTAR shares

many features in common with our system. But our system need not worry about modifying the file structure dynamically since it is the simulators that ouput into the files.

Kidd [8] proposes a language for retrieving values from a complex data structure. This structure is similar to the one proposed in CHAPTER IV as an extension to our present system.

There has been some criticism by Levy [9] of the "political and educative function of the model" [9, p.7]. He believes that the model will be limited in use to groups already possessing extensive resources and influence. He points out that the models are based on the status quo; but a more detailed design of the models could allow for more radical policies. He states that "the introduction of the simulation into the political sphere will virtually force decisions to be channelled through the model" [9, p.15]. I believe the model will be used to indicate trends caused by policy decisions.

Our system is unique in the fact that it presents an information management system where the data generation is through simulation.

§ C. <u>the overall system</u>

IIPS needed quite a sophisticated computer system to monitor the simulators. The opportunity was ripe for the development of a simulator supervisor and graphical output package. Although the system was designed with the IIPS project in mind, it is by no means dependent upon the specific nature of their simulators; it was designed with the express desire for generality and flexibility, although this is only verified after extensive use. The system is divided into various natural well-defined components, namely the user commands, the

supervisor, the simulators, the file maintenance, and the graphics. The interfaces between these components were well-defined during the design phase. Dividing a system into separate components has many advantages: the problem becomes much clearer, and the coding becomes easier. This principle was used whenever possible during the design of the system.

§ C.1 the user commands

The user communicates with the rest of the system through a command language; this sort of procedure is standard in any large user oriented computer system. The command language was designed with various criteria in mind: simplicity, brevity, uniformity, and ease of understanding.[1] The commands are simple in their structure. The words chosen for the language are short in length, but clear in meaning, and distinguishable from each other. The syntax is such that parameters which are not essential are optional and they default to the most common value. And abbreviations may be used whenever desired to shorten the communication time; frequently used phrases or whole commands may be stored away for recall later through a user defined name. The ease of understanding a command has not been sacrificed for the sake of brevity or simplicity. As the user becomes more sophisticated room must be provided for him to express his individuality within the command structure. He must be able to modify various default options and use more complicated options of the commands as he becomes more experienced. All of the commands have the same overall structure: an alphabetic command name followed

---

[1] The MTS command language was used as a model.

by parameter fields. The prefix character @ is used to prompt for a command to be entered. The command language satisfies the design law of least astonishment, which states that a command should do what it logically appears to do [3]. By disengaging the command language from the simulator control we now have a uniform language common to all simulators.

§ C.2 the supervisor

The supervisor controls the action of the whole system. Its function will be explored in great detail in later chapters. At this stage its relation to the rest of the system will be briefly shown. The supervisor interprets the commands and executes the appropriate routines - hence upon its shoulders rest the communication between the user and the various parts of the system. Through the DISPLAY command the simulators are run to produce any data that is missing from a desired graph; the file maintenance and data extraction routines are called; and the graphics routines are activated. Passage through the supervisor is necessary before the modules of the system are executed.

§ C.3 the simulators

The simulators are run only at the request of the supervisor through a DISPLAY command. This removes from the user the responsibility of running the simulators and generating the correct data for a graph before asking for a display. This is a powerful feature. The user does not have to worry about which simulators are necessary to produce the data and whether or not the data is already present. The data which is used for input may be original data or output data from

a previous simulator run. Original data is not treated specially so it may be modified during a later simulation run. Simulators are forced to produce only the data that is required for the graph.

The simulators are run by year - in other words given a year as input the simulator generates results for that year. Hence all output from the simulators is linked to a year. A simulator is just considered as a subroutine to the system so as input it needs all parameters necessary to define the state of the simulator. All of the input and output is stored in disc files. The simulators' only access to the world is through the data base and the policy vector. It is permissible for a simulator to demand as input the output from another simulator as long as there is no closed cycle of requests. An example of a simple closed cycle of requests would be a simulator S_1 using input I_1 that is the output $\emptyset_2$ from simulator S_2. But S_2 uses as input I_2 which is the output $\emptyset_1$ of S_1. This is shown in figure 3. Simulators may be added or removed from the system extremely easily (but care must be taken to make sure that output from a removed simulator is not used as input to a remaining simulator). Of course, the simulators must conform to the standard read-write routines provided for file maintenance; and the simulators cannot provide any external output. Hence, the simulators and the user are completely separated: the user does not know the details of running the simulators, and the simulators do not know about the existence of users.

§ C.4 <u>the file maintenance</u>

The file maintenance routines were developed in order to present a standard package to the simulator coders. A uniform approach is required since the simulators interact and the supervisor must read their output.

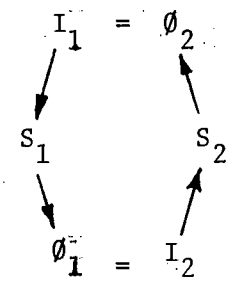


figure 3: example of closed cycle of requests

keylist for file i

K_{11}	. . .	K_{1n}	L_1
K_{21}	. . .	K_{2n}	L_2

contents of file i

K_{11}	. . .	K_{1n}	F_{11}	. . .	F_{1m}
K_{21}	. . .	K_{2n}	F_{21}	. . .	F_{2m}

where: K_{ij} is a key value

F_{ij} is a field value

L_i is an MTS line number

figure 4: format of a file, with its keylist, in the data base

All of the data handled in the system is stored in disc files. The entries, called records, in the file each must begin with a set of keys that identify the record for sorting. The format of a file with its keylist is shown in figure 4. These keys need not be unique, but their value and all the line numbers of their associated records are kept in a table called a key list. Whenever a record is added or deleted some adjustment must be made to the key lists. The reason for the key lists being kept is that they provide for quick random access to records which otherwise could not be obtained without a sequential search. Depending upon its needs, the simulator may use sequential access or use random access since both are available. The file maintenance routines in conjunction with the command language and the graphics routines remove much of the unnecessary drudgery in simulator writing.

§ C.5 <u>the graphics</u>

At present the graphics package is the only form of visual output for the simulators. The simulators always output their data into files from which the DISPLAY routine retrieves the data values that it passes to the graphics supervisor, which is responsible for graphics output. The graphics package can only be activated through the DISPLAY command, hence all output is controlled by the supervisor. If it is to satisfy the needs of all possible simulators it must be capable of quite a variety of graphs. The graphs may be shown on a variety of different types of devices, namely a remote printer, the source terminal, a teletype, a plotter, a refresh display screen (Adage AGT-10), or a storage tube (Tektronix). Each device may be divided up into separate

areas each of which may contain its own graph. This should enhance visual comparison of graphs. The following types of graphs may be shown on any of the above devices: bar, line, dot, contour, or map. Any graph may be superimposed upon another graph for ease of comparison. The order of displaying the graph may be altered by choosing any one of the variables in the DISPLAY command to sequence the graph. Since output devices have only a two dimensional graph surface some tricks must be employed to visualize any graphs in three-dimensions: two-dimensional slices are given of the graph in three-dimensions. The rate at which these graph slices are shown can be controlled through a parameter in the DISPLAY command. Before the actual graph is plotted, the scaling of the values has to be done, and the titles to the axes have to be added. Each graph that is displayed on an area of a device is stored in a push down stack with a default level of four. This feature enables a previous graph to be reshown on a device without having to recalculate the graph values.

This completes the brief description of all the parts in the system. It is hoped that with the availability of making policy changes, and displaying all related variables by means of the extensive graphics package that the user can test his ideas. It is also hoped that the user's imagination will be stimulated into thinking of new hypotheses to test, whose results can be compared with the predicted or desired results.

§ D. <u>implementation</u>

This system is written to operate under the University of British Columbia's IBM 360-67 duplex. The University uses the Michigan Terminal

System (MTS). Whenever possible we have used features provided under MTS and by the UBC Computing Centre; hence the system is installation dependent. MTS offers excellent remote terminal support and it is intended that most of the usage of this system will be from a remote terminal. The implementation of the system is in IBM-360 assembler language and FORTRAN-IV. At this point a hearty thanks should be given to UBC's Computing Centre which has provided an excellent computing and debugging environment, without which the system would have never proceeded so far so fast.

CHAPTER II THE SUPERVISOR - AN OVERVIEW

§ A. the role of the supervisor

Since the supervisor is the heart of this system, attention is focused on it in a general way before examining it in detail. Because the simulation task is split up into independent sub tasks, a supervisor is necessary which will oversee the execution of the system. The supervisor must interpret the desires of the user and take the correct sequence of steps to produce the desired results.

The role of the supervisor can be best exposed through examining the consequences of the user issuing a simple DISPLAY command. When a user issues a command the supervisor branches to the section of code written for that specific command. A DISPLAY command may necessitate the generation of data which is required for a graph but not yet present in the data base. If we need to generate data, the supervisor must run the simulators which produce the data. The data may be in a static file so it does not have to be generated. When all of the necessary data is present in the data base the DISPLAY routine extracts from the data base the data for the graph. These values are then sorted and passed to the graphics supervisor. After the code for the DISPLAY command is finished control is returned to the supervisor which then returns control to the user, who may enter another command. The supervisor's role as the heart of the system is presented in figure 1.

§ B. the design of the supervisor

The supervisor was designed so that it would perform the following functions:

1. Initialize the system by loading tables. As much specific information as possible is read in as initial data so that it is very easy to add, delete, or modify for example, fields in a file, files or simulators. A description of the following is read in: each file and the fields contained within it; each simulator and the input and output files associated with it; each device name for graphical output; each class name and the files associated with it; and each policy vector name and the files associated with it. The supervisor must interface with the SCANNER, which replaces each text string in the command with an integer equivalent.

2. Interpret all of the commands issued by the user (the commands are issued by a user either interactively from a remote terminal or from a batch station).

3. Know when to call a simulator, and which one to call in order to generate the correct data.

4. Load the simulator and pass control to it.

5. Extract from the data base the values needed for a graph, and pass the appropriate array to the graphics supervisor.

6. Make policy vector changes and invalidate the appropriate data due to the policy change.

7. Respond to errors and return intelligent error messages.

8. Establish logical area numbers on devices.

9. List the policy names, class names, file names, simulator names, field names within a file, file names within a class, data values for fields or files.

10. Return to MTS mode and then later to restart the program from where it was interrupted.

11. Save calculated data in a permanent file, and to restore this data at a later time in the same run, or in another run of the system.

12. Define a text string by a symbol, and then replace the text string whenever the symbol is mentioned.

13. Explain the syntax and meaning of the commands to an unfamiliar or forgetful user.

14. Restore a graph previously shown on an area, or erase the present contents of an area.

15. Set certain run time parameters. With a system which performs such a wide variety of tasks, it is obvious that a clear and concise command language is necessary.

At present the supervisor performs all of the above functions and also some less important functions. Each function is written as a separate unit so that adding more functions to the supervisor is very easy. Some of the extensions that are planned for the supervisor are discussed in CHAPTER IV.

CHAPTER III THE SUPERVISOR IN DETAIL

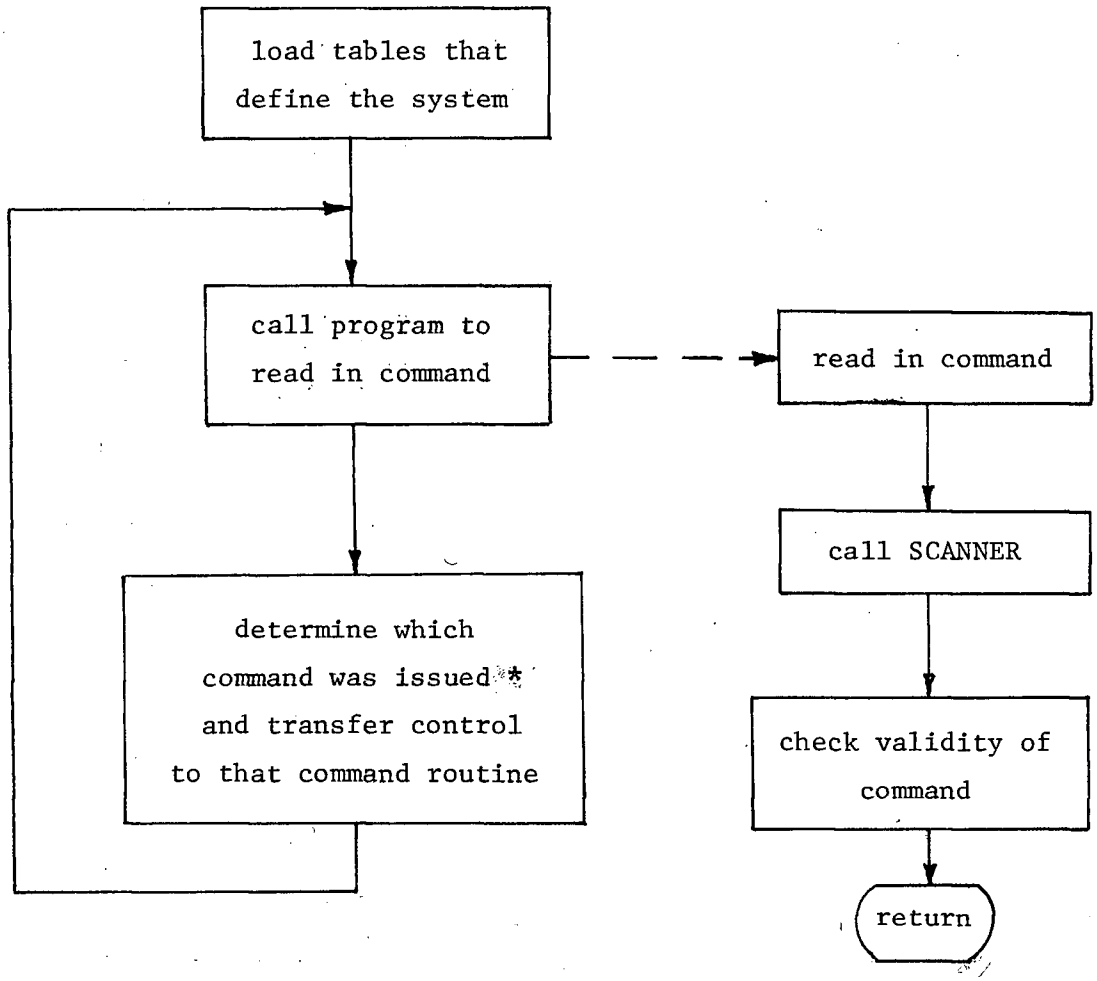
§ A. organization of the supervisor

The supervisor is organized in a hierarchical structure in which each component is self-contained except for parameter passing. This is visualized by the flow chart in figure 5. The SCANNER routine converts the alphanumeric strings into integers so that variable length strings may be treated as integers within the program. The main program is responsible for transferring to the correct routines - so we must look at the called routines in order to extract the details of operation.

§ B. loading tables for the supervisor

The idea of loading tables of information that define the state of the system is important. This means that there are few specific details programmed into the system - most of the details come from the input tables. In order to change the number of fields in a file, or change the number of input files to a simulator, or change the name of a simulator or any other change in the input data is a trivial task. Of course, most of the changes involve altering some code within a simulator. Now it should be clear why it was strongly stated that this system is not dependent on the IIPS project: any simulator may be used that has modified its input-output to comply with the conventions. The aim has been to satisfy the computer law that "one man's constant is another man's variable." [3].

The program for loading the tables is similar in structure to the main program just described. It initializes some variables. It then reads in a keyword table which is used by the SCANNER when it assigns



* command names are listed in the appendix.

figure 5: flow chart of supervisor

integers to alphanumeric strings. The keyword table also indicates whether the string is a special character, keyword, blank, integer, or an ordinary string. Then the program reads in an input line in character format, and has it translated into integers by calling the SCANNER. Some of the input lines contain command words which indicate the type of information that follows. When the program encounters a command word it branches to the appropriate routine to handle the setting up of the table. The last level has now been reached: the routines that create description tables for files, simulators, the policy vector, classes and devices.

Just before the structure and contents of the various tables are described, it would be instructive to explain how they have been implemented. Fortran arrays are not very adaptable for dealing with a varying number of fields. But the assembler DSECT provides the ideal tool for handling the table information. Blocks of information can repeat an indefinite number of times, each field within the bolck can have a name, and pointers from a main structure to a substructure for varying number of entries within a fixed structure can be used. Choosing to handle the tables with the assembler DSECT had implications throughout the rest of the supervisor: it forced most of the coding to be done in assembler language.

§ B.1 <u>the file description table</u>

The file description table consists of a logical description of each value file (file which contains the data values) in the data base. Remember that the data base consists of many value files each of which is associated with a specific simulator. Each file description has a pointer to a decription of all fields within the file. Each of the records

in the value file contains a list of data values. The reason for the logical description is to associate a name with each file and its fields, and to provide more information about each which will be needed when certain decisions are to be made. Figure 6 shows how the file description is stored.

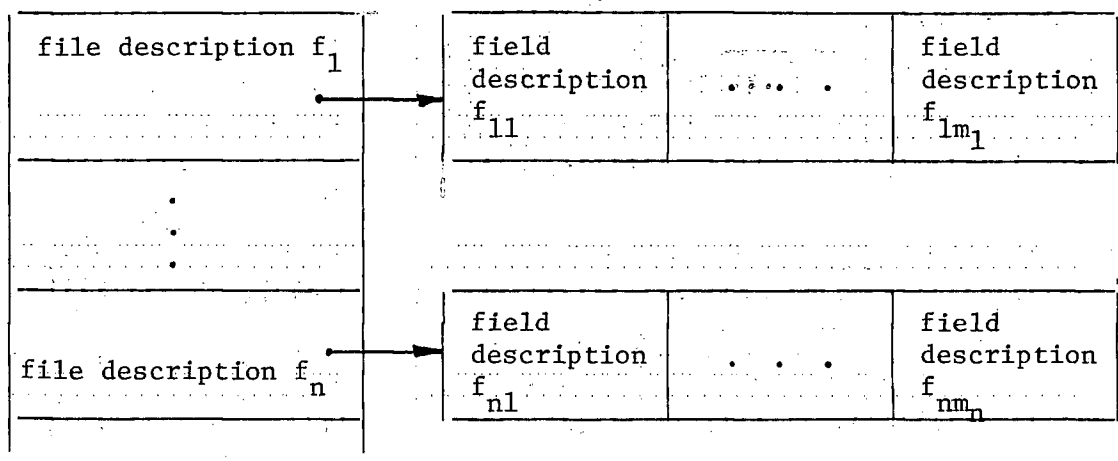
When the simulators wish to reference a file they need only use the logical file number. The system file may be a temporary file by default or a permanent file by request. If any initial data for a file is to be modified then the data is copied into a system file. It is instructive to understand the need for all of these different names: due to their independence, the simulator, the user, and the system may all use different names to reference the same file. The table contains the address of the key list if the file is currently being referenced. The entry point name is to be used when control is passed to the simulator. Each entry in the file table contains a pointer to the description of the fields within the file. The field description cannot appear within the file description table because there are a varying number of fields within a file.

§ B.2 <u>the simulator table</u>

The structure of the simulator table is very similar to the file table. Each main structure (a DSECT overlay) has the simulator name and facts about it; the substructures have a varying list of input and output files. Figure 7 shows how the simulator table is stored.

§ B.3 <u>the policy vector table</u>

The structure of the policy vector table is similar to the simulator table. The table is necessary for the inclusion of policy vector



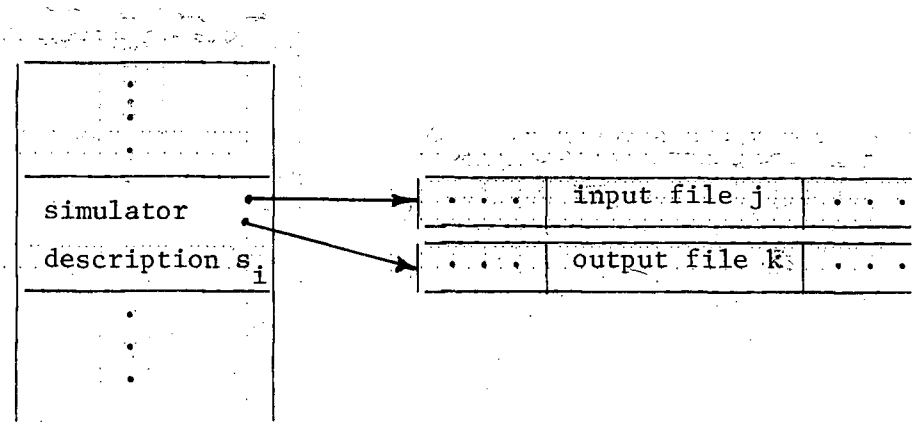
file description table contains:

1. logical file number
2. user name for file
3. name of permanent file for initial data
4. name of system file for data
5. address of keylist
6. number of keys
7. number of fields
8. format of all values in file
9. location of object module of simulator that produced data
10. entry point in simulator
11. pointer to field description

field description table contains:

1. user field name
2. units for name
3. indication of whether field name is a key or not

figure 6: file description table



simulator table contains:

1. simulator name
2. name of object file for simulator
3. flag to indicate if it is in core
4. loader address of object module
5. pointer to input table
6. number of input files
7. pointer to output table
8. number of output files.

input table contains:

1. logical file number
2. flag to indicate whether or not input values are for current year or previous year

output table contains:

1. logical file number

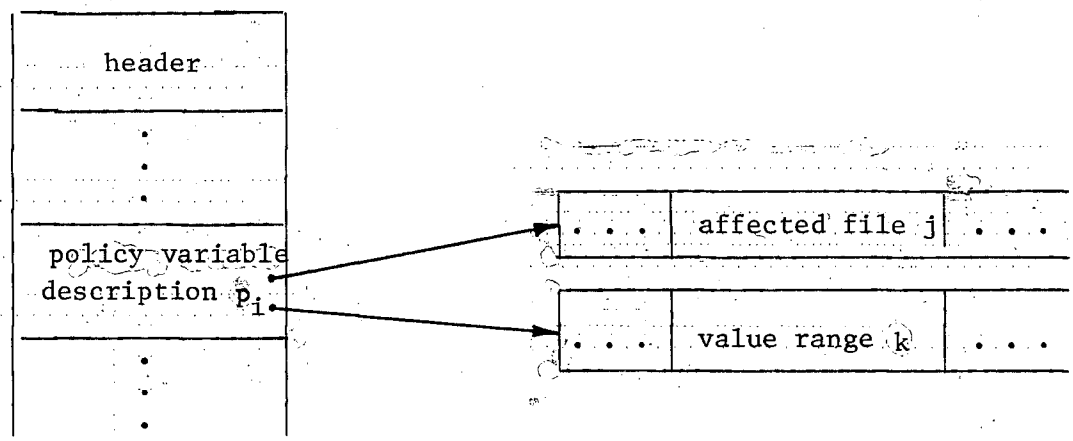
figure 7: simulator description table

intervention by the user. The range of the policy variable and its values may be changed dynamically through the CHANGE command. The policy table has a header, which does not repeat for each new description, that contains the number of policy descriptions, the default year for begin year (the first year for which simulators produce data) and end year (the last year for which simulators produce data). The structure of the policy vector table is shown in figure 8.

The order in which the policy variables are stored needs the establishment of a communication convention because the simulators which need access to values of the policy vector need not use the same names as the user. Therefore, the convention was established that the order in which the policy variables appear in the policy vector description would be the same as in the CØMMØN array that contains the values. Both the supervisor and the simulator use the same CØMMØN block name hence transfer of data values is trivial. The first few positions of the policy vector are reserved for the supervisor to pass information to the simulator. For instance, the convention was established that the first position would be the projection year (the year for which the simulator is to produce data), the second position would be the end year (the last year of simulation), the third position would be the current year (usually the begin year), and the fourth position would be for the year increment value, defaulting to one.

§ B.4 <u>device table</u>

Knowledge of the last two tables, the device table and class table, is not essential for an understanding of the supervisor, but it is included for the sake of completeness. The structure of the device table



policy vector table contains:

1. name of policy variable (user name for element in policy vector)
2. minimum increment size
3. units of policy variable
4. number of files affected by policy change
5. pointer to a list of files
6. number of value ranges
7. pointer to list of value ranges
8. base year (first year the user is allowed to make policy interventions)
9. minimum value of policy value
10. maximum value of policy value

files affected table contains:

1. logical file number

value range table contains:

1. value of policy variable
2. low range of interval
3. high range of interval

figure 8: policy vector table

is very simple because it has no varying length items. This table is necessary for the graphics supervisor which is responsible for plotting graphs onto the appropriate device. It contains a device name, a device code number, and a device type code. The user refers to the device by the device name. The supervisor refers to the device by the device code number. The device type code is necessary to group devices that are physically different but which may be treated similarly by the graphics supervisor; for example the line printer, IBM 2741 teletype, IBM 2260 display station, and IBM teletype may all be treated in a similar manner.

§ B.5 class table

The structure of the class table is similar to the simulator table. It is necessary in order to define a hierarchy of files. Since a simulator may reference many files it is useful to be able to group all of these files under a class heading for various commands. The table contains the name of the class (same as simulator), the number of files in the class, and a pointer to a list of the logical file numbers. It contains the number of policy variables within the class, and a pointer to a list of the policy names and the position they have in the common block. The list of policy variables is used only in an option of the LIST command.

§ C. external interfaces

It was stated previously that the interfaces between the supervisor and the various modules were well-defined. Now that most of the groundwork has been laid these interfaces can be examined in more detail. The main interfaces are shown in Figure 1. These involve the user commands, data base, simulators, and graphical output.

§ a. user commands interface

The user converses with the system through the command language. The command language was written for the maximum user comfort; hence it contains many varying length alphanumeric strings, some to be treated as special characters or keywords. The interface involves a section of code known as the SCANNER, which acts as a translator. The SCANNER intercepts the command before it is accepted by the supervisor. Its function is to substitute any replacement names, and to replace the alphanumeric strings with their integer equivalents obtained from the keytable described earlier. (Chapter III, § B). The purpose of such a front-end is to simplify coding throughout the rest of the system by allowing references to variable length strings to be done through integers. This string of integers is passed by the supervisor to the correct routine to analyze the command; therefore any change in command structure need only be handled by the lowest level in the supervisor structure.

§ b. data base interface

The data base may be referenced by the supervisor or by the simulators; the supervisor may only read the contents, but the simulators may read and write the contents. They both ultimately interface through the READ-WRITE routines, which may be either sequential or random. The sequential read is just like a FORTRAN logical read. When using a sequential read one may issue a find command which locates the beginning of the read by using a key value, or else one will read the next record (if the last record read was from this file) or it will read the first record. But the random read uses a key value to identify the line to

read. In order that the random read not involve a sequential search for the key value, a list of key values and their associated line numbers is kept with each file. This key list physically occupies the initial portion of each file in the data base. The sequential write and random write operate in the analogous manner.

It is necessary to consider how the supervisor and the simulators determine the contents of a record in a file. The supervisor uses the file description table, in order to associate a field name with a value in the data base. When the user requests information about a field name the supervisor must look up the name in the file description table and then determine the position of the field within the record. The simulator uses a program dependent logical description in order to identify the elements of the array of values for field names. The names assigned to the values need not be the same as those in the file description table.

The policy vector may be considered as part of the data base. The simulators may only read it, but the supervisor may read and write it. The supervisor uses the logical policy description table in order to discover the position of the policy name in the value array. The position is the same in the value array as in the policy vector description. The simulators may assign any variable names to the elements of the value array but the elements must correspond in meaning to the elements in the supervisor policy description table.

§ c. <u>simulator interface</u>

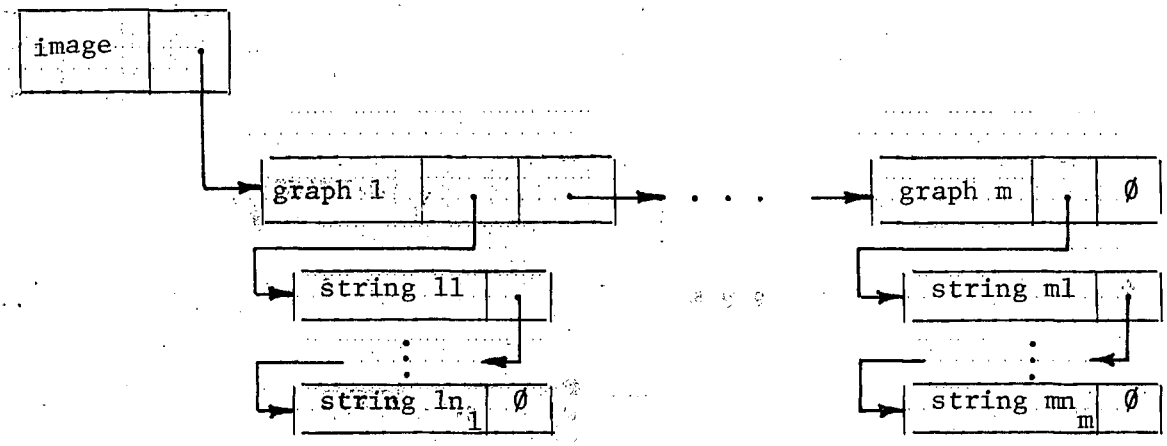
The simulators are interfaced to the data base and policy vector, and to the supervisor. The interface between the simulator and the data base and policy vector has already been described in the previous section. The supervisor is responsible for loading a simulator and transferring

control to the proper entry point when data must be generated by it. A simulator is dormant until it is called by the supervisor. The simulator will be asked to produce data for only one year at a time. Since this may be any year, the simulator must store away the state of the model after each run. The simulator will comply by generating all of its data for that year and writing into its output files. After generating data for a year, the simulator will pass control back to the supervisor. The supervisor is also responsible for unloading the simulator when it is through its current task of generating data. The simulator acts very much like a subroutine to the supervisor.

§ d. graphics interface

The graphics supervisor is called by the supervisor during the execution of the DISPLAY routine. The graphics supervisor expects quite a lot of information from the supervisor. It expects the information to be passed in tables common to both programs. Broadly speaking, the tables contain some parameter values extracted from the DISPLAY command, and some items that the supervisor must generate from its own tables, and a list of the data values to be graphed. The tables have many entries because the graphics supervisor is responsible for producing a wide variety of graphs. The information is stored at three levels: in a FORTRAN CØMMØN area, and in two areas which each have a DSECT overlay. A pictorial view of the information structure is given in figure 9.

The FORTRAN CØMMØN area contains the description of the whole image to appear on the graphic device, which may include several superimposed graphs. All of the information in the CØMMØN area can be



the image CØMMØN area contains:

1. the pointer to the first graph DSECT
2. area number on which the image is to appear
3. display rate
4. flag to indicate whether the image is to be superimposed or to be a new image
5. the length of the title
6. image title

the graph DSECT contains:

1. pointer to the first string of the graph
2. pointer to the next graph in the image
3. number of strings in the graph
4. number of points in each string
5. a code to indicate whether the type of graph is bar, line, dot, map or contour

the string DSECT contains:

1. pointer to the next string in the graph
2. flag to indicate whether or not the graph is to be sequenced by the string
3. the units for the axis
4. length of the axis label
5. axis label
6. all of the data values for the string

figure 9: information structure for graphics

obtained from the DISPLAY command. More than one graph DSECT may appear in an image if superposition has taken place. The string DSECT has descriptive information local to each field name. The order of the values within each string will be explained in the discussion of the DISPLAY command. From this generalized format, the graphics supervisor manipulates the data values and produces the various types of graphs necessary.

§ D. internal reaction of supervisor by command

In order to explain the lowest level of the supervisor hierarchy it is necessary to examine the reaction of the supervisor to a command. When a command is issued by the user the supervisor passes control to the proper section of code which analyzes that command. By far the most important commands are the DISPLAY command and the CHANGE command. The other commands execute very useful functions, but they are of secondary importance compared with the DISPLAY and CHANGE commands.

§ a. the DISPLAY command

The DISPLAY command is responsible for a wide variety of tasks. It must call the appropriate simulators when data is needed; it must handle policy variable names in the DISPLAY command; it must extract and sort the data values before they are passed to the graphics supervisor. The data extraction algorithm is divided into two logical parts: the variables in the DISPLAY command are arranged into a tree structure which has as its foundation the linking of the display variables by common keys; and the strings of data values are extracted from the data base using this tree structure. The information retrieval algorithms involved in extracting the data and sorting it for graphical output

were designed to be general enough so that they could operate on a more complex data base. These algorithms really form the foundation for the DISPLAY command!

The code for the DISPLAY command has been modularized like the rest of the system. The display task splits up into logical sections each of which is relatively independent. It has been divided up into sections that:

1. parse the command string.
2. build a relation tree for the display variables.
3. determine whether or not simulators must be loaded.
4. load the correct sequence of simulators so that the proper data values are generated
5. extract the data values
6. sort the data values for the graphics supervisor.

§ 1. <u>general philosophy</u>

Flexibility, simplicity, and generality were incorporated into the DISPLAY routine during the design phase. The DISPLAY routine must be capable of handling a wide variety of tasks. It must be flexible enough to meet the many desires of users at various levels of sophistication. Since a user may not be familiar with computing equipment or with the details of the simulators the DISPLAY routine performs many of the mundane functions necessary to produce the resulting graph. The idea is to make the production of graphs appear simple to the user. The user need not worry about calling a simulator, setting up initial data, scaling graphs, and the like. The algorithms used in the DISPLAY routine must be general enough to accomplish the variety of tasks required of it at present, and to meet some of the extensions planned for it in the future.

Certain assumptions had to be made on the structure of both the files and the DISPLAY command. If the same field name appears in more than one file then it must have the same key structure. The assumption is necessary to insure that unique graphs result from a DISPLAY command. Files are generally of two types: a file with time as a key, and a file without time as a key. This distinction seems artificial, but it is necessary. If a file does not have time as a key then simulators cannot produce data for the file, and hence the file is assumed to have static data already present.

Only one policy variable may appear as a variable in the DISPLAY command. Policy variables are much different than non-policy display variables. A policy variable is not contained within a file and simulators cannot produce policy values. Also each time a policy value is altered, through the CHANGE command, it causes some information within the files affected by that policy variable (this list is stored in the policy vector table) to be nullified. The nullified information consists of all records with the key value time greater than the 'from' modifier if it is specified. If the 'from' modifier is missing then the invalidation always begins at the same year, the base year, for a policy variable. It is independent of the current year used for the DISPLAY routine. This limitation to only one policy variable is partially for the protection of the user. If more than one policy variable is allowed then the number of policy changes and consequently the regeneration of data that would occur would demand a great deal of CPU time. For each policy change in one of the variables all of the other policy variables must run through all of their policy changes, hence as more policy variables are added the amount of extra work increases factorially with the number of policy changes per variable.

§ 2. parsing the command string

The initial task of the DISPLAY routine is to identify all of the fields and their modifiers within the DISPLAY command. This section of code was written so that in general the order of the fields within the DISPLAY command is not important. When the supervisor passes control to the DISPLAY routine the output of the SCANNER is passed in a CØMMØN area. Before the parsing procedure can be adequately explained, the syntax of the DISPLAY command must be shown:

$$\text{DISPLAY [SEQUENCED BY] } v_1 \text{ [FRØM } v_{11} \text{ TØ } v_{12} \text{ BY } v_{13}] \begin{bmatrix} \text{SEQUENCED} \\ \text{AT RATE } r_1 \end{bmatrix} \text{ BY } v_2 \ldots$$

$$v_n \text{ [FRØM } v_{n1} \text{ TØ } v_{n2} \text{ BY } v_{n3}]\text{[AT RATE } r_n]\text{[TYPE=t]} \begin{bmatrix} \text{NEW} \\ \text{SUPERIMPØSED} \end{bmatrix}$$

$$\text{[ØN AREA a]}$$

where:

v_i is the i'th display variable

v_{ij},a,r_i are the values associated with the preceding parameters (may be real or integer)

t can be 'bar', 'line', 'dot', 'contour', or 'map'

n is the number of display variables (from 2 to 4)

[] indicate optional portions

The method of attack is as follows. The output of the SCANNER is a string of integers, and a string of flags indicating the significance of the corresponding integer. These strings are passed in a COMMON area to the DISPLAY routine. Since a DISPLAY command may be issued to graph only the values of a policy variable or all of the policy vector (syntax not shown) a check for this situation is made immediately. It is inconvenient to search the command each time some information from the command is needed. Therefore as information

about each display variable is discovered it is placed in a section of storage which is overlaid by the display DSECT. Storing the information in this display DSECT has the advantage that any change to the syntax of the DISPLAY command only affects the initial part of the program. The display DSECT contains the display name, the values of any 'FRØM', 'TØ', or 'BY' modifiers, the value of the rate if specified, and a flag to indicate whether or not this field is to be used to sequence the display. It contains more information which is described later because it is not filled in until later stages of the DISPLAY routine. The display DSECT is in CØMMØN and is used to store information discovered in one section of code but not used until another section.

§ 3. building a tree of variable relationships

The second stage of the DISPLAY subroutine involves building a tree structure to show the relationship between the display variables (v_i's) used in the command. This stage is necessary for several reasons. The consistency of the DISPLAY command must be checked. Each of the variables mentioned in the DISPLAY command is compared with entries in a field description table associated with some file description table. If the display variable is not found in any field description table then each name within the policy vector is compared to the display variable. If the display variable is not found among the policy variables then the DISPLAY command cannot be processed. The field names mentioned in the DISPLAY command need not appear in the same file. There must be some method of relating the files which contain the appropriate display variables. The obvious method to make certain that the field names in two separate files are linked is to insist that there exist

at least one common key between the files. If there are more than two display variables then the linkage between the files need only be pairwise, in other words, it is not necessary to have a common key among all files that contain display variables. For example consider the logical file description of the keys in following three files.

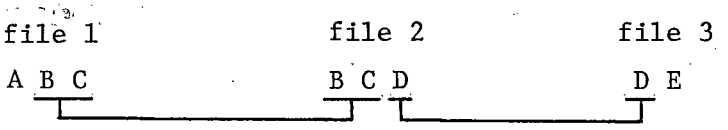


figure 10: example of pairwise links between files.

Then file 1 is related to file 2 via keys BC, and file 2 is related to file 3 via key D. With the simple addition in each file of a special sequence of fields, called keys, the capability of the DISPLAY command has been extended tremendously by allowing cross-file displays. It is at this stage that the DISPLAY command is examined for consistency in the use of relating fields through common keys. This information is necessary for the extraction process.

§ 3.1 method of attack

Why is it necessary to build the relationship into a tree structure? There are several other more straightforward methods of approach. One method could be to just look for a left-to-right linkage in the display variables ($v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4$). Another method would be to pick any one of the display variables as a starting position and then look for any sequence of pairwise links such that every variable is in the chain ($v_{i1} \rightarrow v_{i2} \rightarrow v_{i3} \rightarrow v_{i4}$). The original ordering has been scrambled so that a linkage between the variables may be established. Neither of these methods were used. A more general method was developed that

encompasses these two methods as special cases.

The general method involves building a tree structure of the relationships between the display variables. It is quite easy to see how this encompasses the first two methods. One of the extensions planned for the system is to allow a more complex structure for the data base. Whenever possible this extension has been kept in mind when the algorithms were developed. If the complex data base has a tree structure then this algorithm can be used to discover the linkages between the variables. Now to explain how this algorithm works.

The algorithm is divided into two parts: the first part checks every display variable to make certain that it is contained in a description table; the second part builds the tree. The operation can be visualized by examining the flow chart in figure 11. The first section searches through all of the field descriptions comparing each field name with a display variable. The flag that is set within the display DSECT to indicate a policy variable is checked during the second phase because a policy variable has no key and therefore cannot be linked to any of the non-policy display variables. When one of the non-policy display variables is found the following information is added to the display DSECT. The position of the field in the file is stored for use when the data value is to be extracted from the file. The total number of keys associated with the file is recorded. The name of the file which contains the field is stored. A flag is set to indicate the field has been found. Also data is initialized within the display DSECT for the second phase; this is done because all of the tree pointers are stored within the display DSECT.

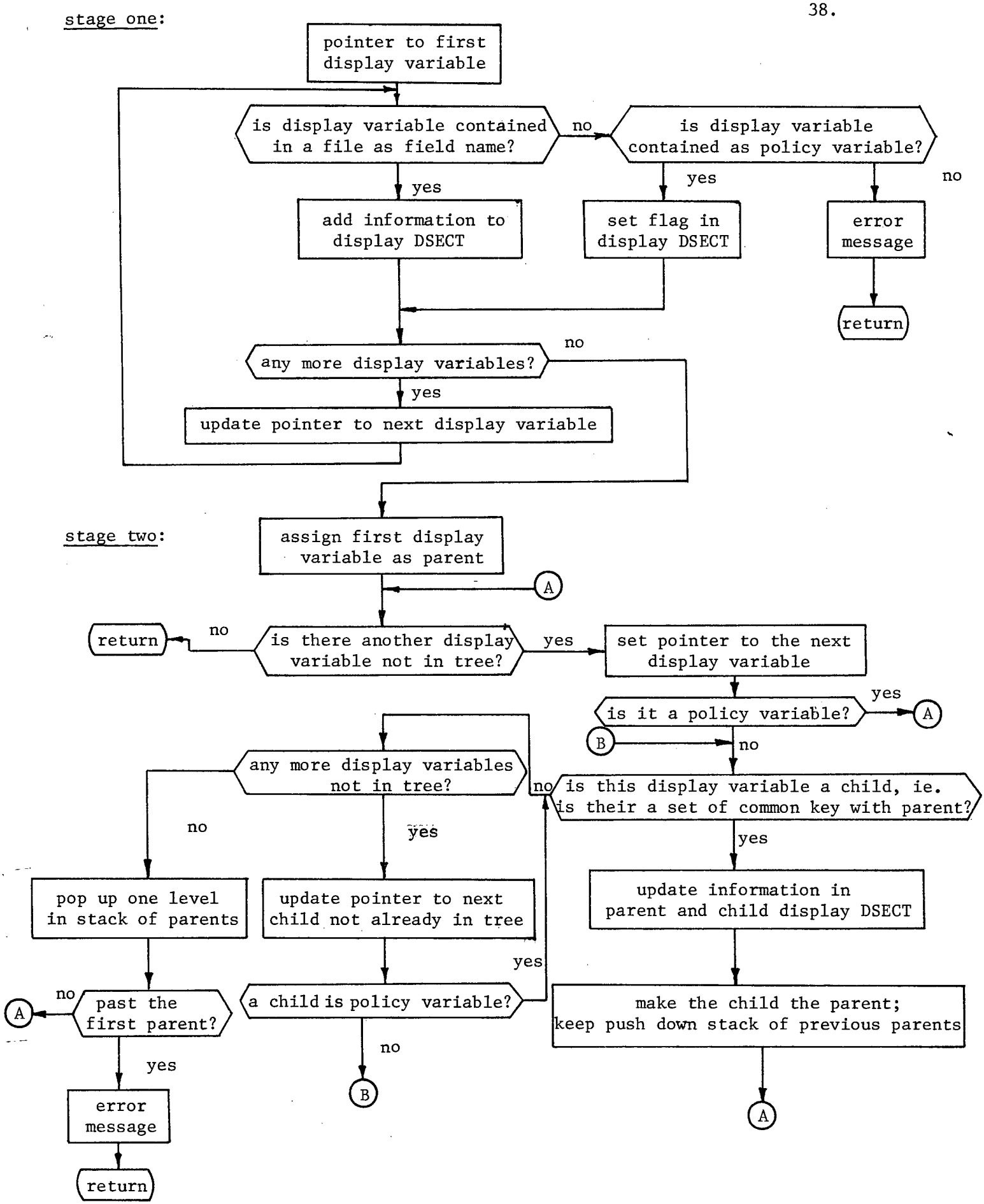


figure 11: flow chart for building of relationship between display variables

The second phase of the algorithm assigns the first variable in the DISPLAY command to be the parent (it is assumed that the parent-child terminology used in discussing nodes of a tree is familiar to the reader). The program then searches for a child of the parent. A child is any variable which is related to the parent by a set of common keys. The program first tries the second display variable as a prospective child. If no common keys exist with the parent then the program tries to match each of the remaining display variables in turn to the parent via common keys. If no match is found then the DISPLAY command must be rejected because there exists a variable which cannot be linked to any other variable. Whenever a match can be made a search is made for the maximal set of common keys. The maximal set of common keys is desired for generality. The keys are used extensively during the extraction process. When a child to the parent is found, information within the parent's and child's display DSECT must be updated. The following are added to the child's display DSECT: the address of the first key in the file, the starting position of the matching keys with the parent, the number of consecutive matching keys, a pointer to the display DSECT of the parent, a flag to indicate the variable has been placed in the tree. The following are added to the parent's display DSECT: starting position of the first matching key, an update of the number of children, and a pointer to the display DSECT of the child. A push down stack is kept of all previous parents. The newly found child is now given the role of parent and the routine proceeds to search through all of the unclaimed display variables (those not already in the tree) to find a suitable child. If a child cannot be found then the last parent is obtained

from the push down stack and a search is made for another child of the parent. Of course, it is assumed that a policy variable cannot be linked into the tree. It should be noted that because the first element in the DISPLAY command is chosen as the first parent that it is the root of the tree. Hence the left-to-right ordering in the display variables is found if it is present, otherwise any consistent ordering is detected.

§ 3.2 examples

Suppose the following type of DISPLAY command is issued:

DISPLAY v_1 BY v_2 BY v_3 BY v_4

where v_i $(i=1,\ldots,4)$ are all non-policy display variables.

The examples in figure 12 give all possible non-isomorphic trees that result from the above DISPLAY command. If a display variable v_i exists which is not related to any other field then disjoint trees result, and hence the DISPLAY is invalid. If any of the display variables is a policy variable then it does not appear in the tree. The algorithm is independent of the number of display variables.

§ 4. generation of data

After the tree of variable relations is built, the data associated with the display variables must be generated. If a policy variable is present in the DISPLAY command then a special sequence of code must be executed. Each time the value of the policy variable changes the CHANGE routine must be called, and quite possibly all of the values associated with each display variable must be re-generated. The data for a file is generated by a recursive routine, called FILL, that calls the simulators. Once all of the data has been generated then the

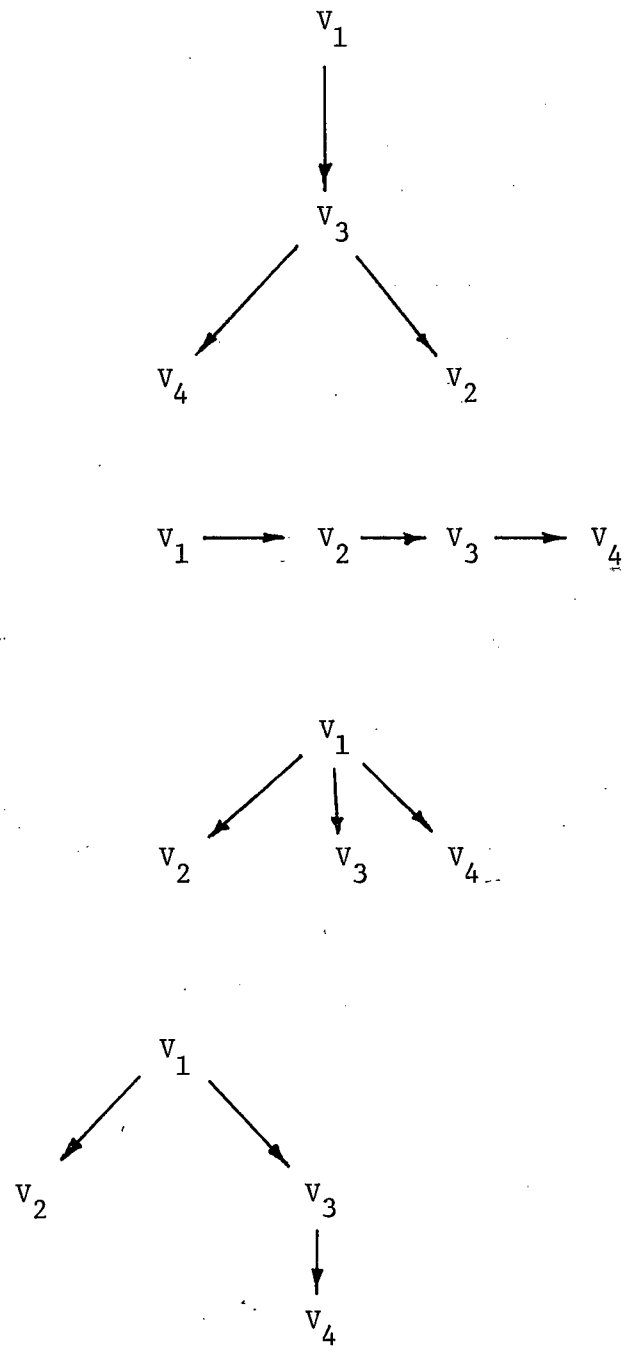


figure 12: examples of relationships between display variables.

pertinent data is extracted. One may ask why the data is not extracted as soon as it is generated, but it turns out that splitting the data generation and data extraction into two separate algorithms makes each one easier to code and easier to conceptualize since they are distinct processes. This is the general framework into which the following details of operation fit. The algorithm is visualized through the flow chart in figure 13.

§ 4.1 method of operation if no policy variable

The first step the program takes is to establish limits for the time interval when it calls the simulators. Remember that simulators are called by time. That is the key they use to know what data to generate. First the program checks the display DSECT to see if time is mentioned in the DISPLAY command. If so then a check is made to see whether a 'FRØM' or 'TØ' modifier is mentioned. If so then the 'FROM' time is copied into a special variable holding the begin year (first year to produce data), and the 'TØ' time is copied into a special variable holding the end year (last year to produce data). If there are not 'FRØM' or 'TØ' fields associated with time, or if there is no time field mentioned in the DISPLAY command then the program extracts the default values for begin year and end year from the policy vector.

Throughout this section it is assumed that no policy variable is present in the DISPLAY command. Once the file that contains the display variable has been located, each of the key fields is compared with the keyword 'time'. If the key 'time' is not present in the file then the assumption is made that the file is static and that all of the information is already in the file. In other words the contents of the file are time independent and must be supplied as initial data for the file because it cannot be generated by a simulator.

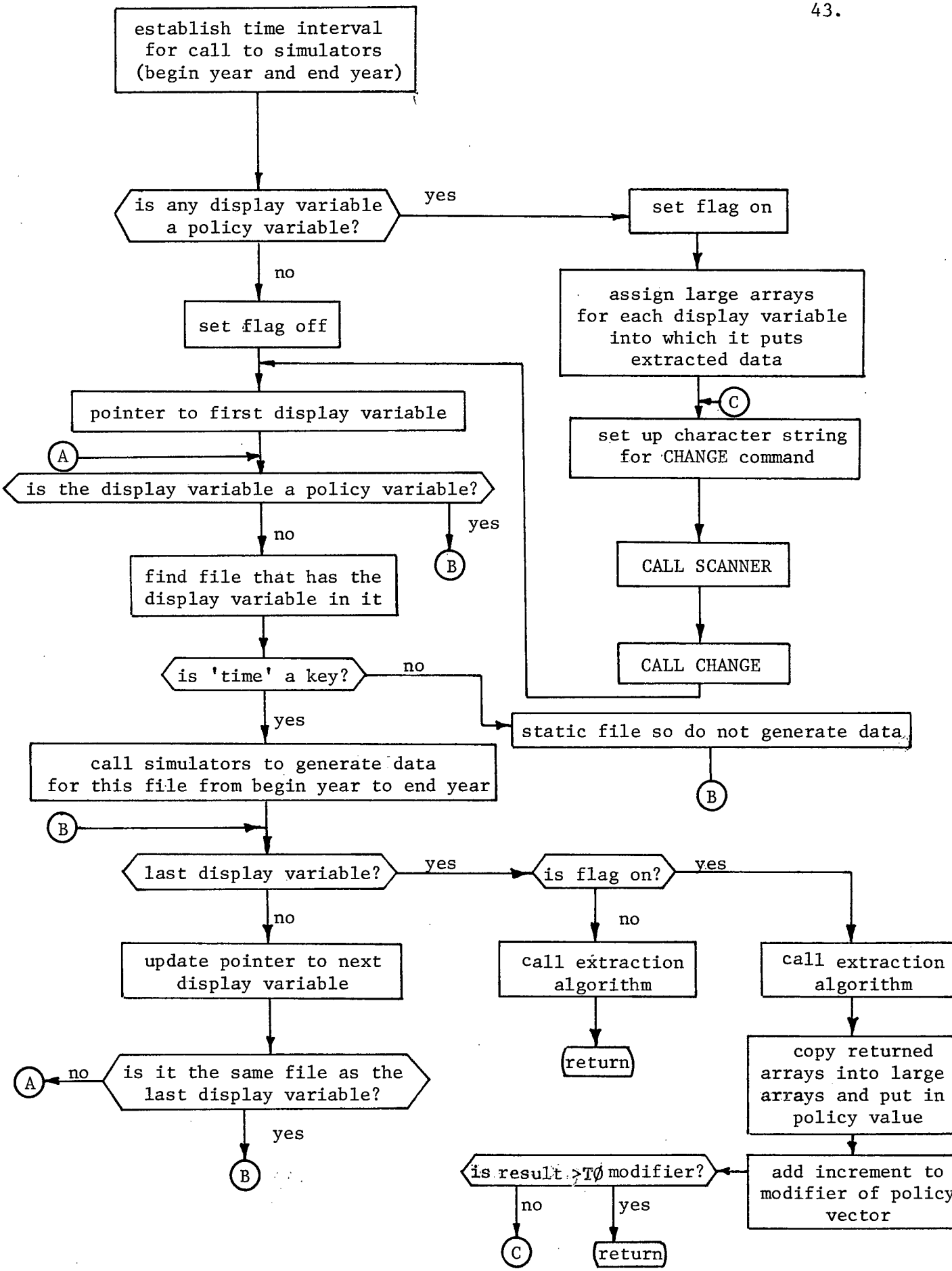


figure 13: flow chart for generation of data

The routine FILL is called when the simulators are to generate the data. The routine FILL is described in section 4.3 It is worth mentioning that when a simulator generates data for a file it generates data for every field within the file, not just the field mentioned in the DISPLAY command. Hence a check is made before the procedure is restarted to see if the next display variable and the last display variable are in the same file. If they are then the data has already been generated. After all of the data is generated for a display variable then the simulators are dynamically unloaded (they were dynamically loaded in the first call to the routine FILL), and the key lists of the input and output files associated with the simulators are released from core storage. After all of the data has been generated for all of the display variables then a call is made to a routine that extracts the pertinent data.

§ 4.2 <u>method of operation if a policy variable is present</u>

Before the program logic is described in detail some general information about the policy vector and changes in policy variables should be known. Suppose that all of the data has been generated with the policy value fixed. When the policy value is altered for the next iteration some data is going to be nullified - possibly some of the data that was previously generated. So before the policy value is changed the previously generated data must be saved in a storage location. But data can only be stored after it is extracted from the data base, hence the extraction algorithm must be invoked before the policy value is changed. It is useful to know that the data extraction routine returns n arrays(where n is the number of non-policy display

variables) each containing a list of the data values for a display variable.

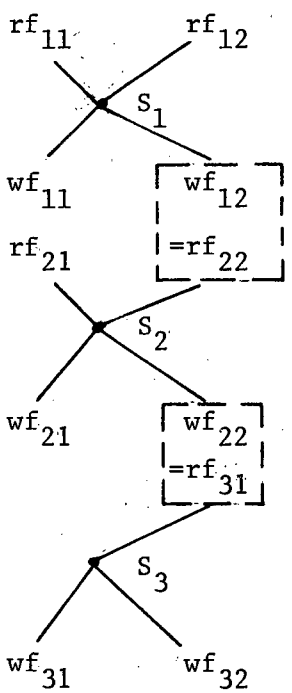
The following program logic treats the policy variables. Suppose that a policy variable is found. Large storage areas are assigned dynamically to each display variable. A pointer to the storage is placed in the display DSECT. A check is made to make certain that the program does not try to generate data for a policy variable. All of the other display variables have data generated exactly as described in the last section.

In the last section after all of the data is generated the simulators are unloaded and the extraction process is called. But now if the flag is on then the simulators are not unloaded because it is possible they will have to generate more data after the policy value is changed. The extraction process is called. The arrays of data values into which the extraction process places its results are copied into the larger arrays that were created as soon as it was discovered that a policy variable was present. These large arrays can be extended dynamically if they are not large enough to hold the copied values. The extraction process does not return an array for the policy variable. But it is necessary to have the present policy value associated with each of the extracted values when the values are passed to the graphics supervisor. Therefore in the large array set aside for the policy variable the policy value is repeated the same number of times as there are values in the arrays returned from the extraction process. After the small arrays are copied into the large arrays the space reserved for the small arrays is released. Then the increment modifier is added

to the current policy value. When all of the policy value changes have been made then the simulators are unloaded, the key lists released, and the data values are sorted and then passed to the graphics supervisor.

§ 4.3 description of the subroutine that calls the simulators

When data for a file has to be generated for a specific time then the routine FILL is called. The routine FILL is recursive because the input data files of the simulator that has to produce the desired output file may not have enough data. Hence the simulator which produces data for these files must be loaded, but once again the input files for this simulator may not have enough data. It is assumed that a file can be written by only one simulator, but a simulator can write many files, and read many files. The structure of the simulator inter-relationships may be shown in the following example:



where: rf_{ij} is a file read in the simulator S_i

wf_{ij} is a file written out by the simulator S_i

S_i is a simulator

figure 14: example of recursive simulator calls.

In the above example if data is required from file wf_{32} then rf_{31} must have enough data if S_3 is to be run; but suppose it does not have enough data. Since $rf_{31} = wf_{22}$ then the program must check all of the input data for S_2. Suppose that the input file rf_{22} does not have enough data but the input file rf_{21} does have enough data, then the knowledge that $rf_{22} = wf_{12}$ is used and FILL looks at the input files for S_1. Now suppose that rf_{11} and rf_{12} both have enough data so that FILL can start poping up from the recursive calls, and finally generate data for wf_{32}.

Now the details of operation will be explained. The first operation is to test the file to see if it is a static file or not. This is done by comparing the name of the simulator that generated the data to the special name 'nosim'. If the file is static then it is assumed that all of the data is present so a return is made from FILL. If the file is not static than a test is made to see if there is any data for the current year (the year for which the simulator is to produce data). This is done quite conveniently by calling the routine SFIND. If the data is present for the current year then a return is made from FILL. If there is not enough data then the object module of the simulator that produces the data must be loaded. This is done by using the name of the simulator, which is in the file description table, to find the correct entry in the simulator description table, which has all of the information about each simulator. If the simulator is not in core then the object module is loaded. After it is loaded the loader address is placed in the simulator description table. Now each of the input files of the simulator must be examined to make certain that they have enough

data. In each file a check is made to see whether the simulator wishes to have data for the current year or from the previous year. It is assumed for simplicity that a simulator may use input data from only one previous year. This is an assumption that should be relaxed in a more general system. If the previous year's data is required then the increment is subtracted from the current year. The recursive loop is entered by calling FILL to discover if there is data for the current year. This procedure must be carried out for all of the input files used by the simulator.

After a state is reached such that all of the input files have enough data then before control is passed to the simulator the policy vector must be filled. It must be filled with the correct policy values for the current year; and this must be done for each call to a simulator because the policy values can vary over time. This is done by picking the correct values from the policy vector description table and placing them in a common array which the simulators reference. The current time is placed as the first value in the policy vector so that the simulators know for which year to produce data. Control is then transferred to the simulator so that it can generate data, and one recursive level is then diminished. The simulators remain loaded so that they may be used in the generation of other data. They are unloaded when all of the data is generated.

§ 5. <u>extraction of data</u>

The problem of data extraction falls under the category of information retrieval. The problem is essentially to retrieve some information from a large data base. The algorithm to extract the data is

quite general even though the data base used in the system is quite simple. Each file in the data base consists of records with only one format. In order to generalize this structure each file should be allowed a tree structure of its elements. With this type of data base the relationship of the display variables is a tree structure. A more detailed discussion of this extension is in CHAPTER IV, but it is mentioned now to illustrate the sense of using the general algorithm to build a tree of variable relations in section 3. But even with the simple data structure the data extraction is quite complicated. In fact most of the problems of data extraction that would arise with a more complex data base also arise with the simple data base. The reason is that the building of the tree structure of relationships that is used when the data extraction is trying to mark keys so that it knows which data value to extract becomes more difficult with the complex data base not the extraction process. Before the data is extracted it is assumed that there exists a tree structure for the relationship between the display variables. Once the data is generated and placed in the data base the only means of identifying it at a later stage is by using the keys which are associated with the data value.

It has been stated that extracting data from even a simple data base is difficult. Here are some of the reasons. Remember that more than one display variable exists so a whole set of data values must be extracted such that each display variable has a data value assigned to it. These data values must be consistent, in other words they must all be linked through key values using the tree structure to show which

keys are to be used. But the necessity of having to extract a consistent set of data values causes some problems. Suppose a file has three keys and the routine is trying to match key values on the second and third key. There may exist more than one set of common key values that are the same but that have different data values. This is true because the first key may vary while the second and third keys remain the same. Therefore the match process may find n ($n \geq 1$) key values matching with 1 key value (or 1 key value matching with n key values) which is reasonable, and it is taken care of by duplicating the data value associated with the one key value n-1 times. If the program matches n ($n \geq 1$) key values to n other key values then each of the corresponding data values are matched. If the program matches n ($n \geq 1$) key values to m ($m \geq 1$, $m \neq n$) key values then nothing reasonable can be done and the data extraction process must terminate with an error message that the DISPLAY command is invalid. All of these problems are also encountered if the data base is more complex. But most of the additional problems are encountered in the setting up of the tree of relationships. Now it should be clear why the data generation and data extraction are broken up into independent modules.

§ 5.1 <u>method of operation</u>

The extraction operation involves two subroutines: EXTRACT and REXTRACT. EXTRACT is the driver for the recursive extraction routine REXTRACT. The general idea is that the driver does some initialization, and then sets a pointer to the key values in a record associated with the dominant parent (the root of the tree). It then calls REXTRACT which tries to match key values beginning from the leafs of the tree.

By beginning at the leafs of the tree data values may be duplicated at the lowest level first if it is discovered at the next higher level that an n to 1 match has occured. The algorithm is recursive since the procedure is the same for every parent in the tree. When all matches have occured the routine returns to EXTRACT which then updates the pointer in the root of the tree and calls REXTRACT again. This gives a very crude idea of the main operations of the extraction process.

To get a better idea of how the extraction process works more details are given in the flow chart in figure 15. If any of the display variables is a policy variable then no extraction is done. A copy is made of the keylist for the file associated with each display variable, and its location is placed in display DSECT. This may seem inefficient if more than one of the display variables occur in one file, but it is easier to do this than to handle special cases within the recursive code. Now for each unique set of keys in the dominant parent the routine REXTRACT is called. By set of keys it is meant the complete set of keys, not just the common keys with one of its children. The routine REXTRACT is given a pointer to the set of keys upon which a match is sought, and a pointer to the display DSECT of the parent. The reason REXTRACT is called only with the unique keys is that if two keys are the same then the program avoids duplicating the set of values. When all of the key values have been examined then the keylists are freed and the code returns from EXTRACT. Therefore EXTRACT is nothing more than a driver for REXTRACT.

All of the work of extracting the sets of data values falls on the shoulders of REXTRACT. REXTRACT initializes some parameters then makes

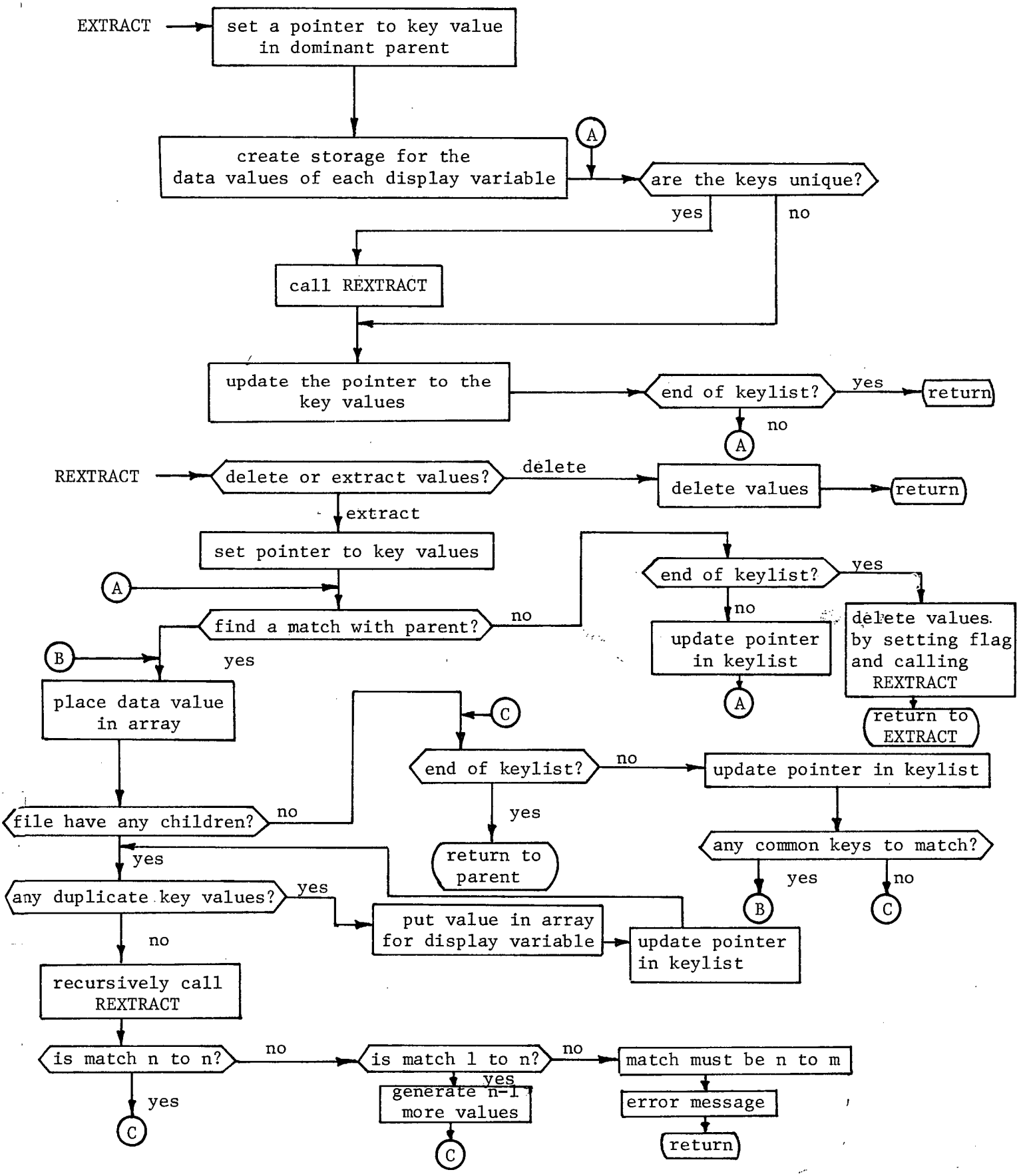


figure 15: flow chart for extraction of data

a check to see whether or not the call is to delete data values or to extract data values. The reason data values may be deleted is that in a child it may be discovered that there is no common key value that corresponds to the common key value in the parent. Hence all data values recorded for that match in the parent and in every relation back to the dominant parent have to be deleted. Otherwise the routine begins examining the first set of key values in its keylist. A pointer is set to the current set of keys, and a match is sought between these keys and the keys passed to REXTRACT. The match is tested only on the common keys. The position of the first common key and the number of common keys is already in the display DSECT. If the line number associated with the key list is zero then it is assumed that the key values have already been matched. If REXTRACT finds a match with key values that have a non-zero line number then the number of points returned is increased, and the data value is placed in the data array for the variable. This is done by calling a routine which returns a data record given a line number and a logical file number. Then the data value for the appropriate field in the data record is extracted and placed in the data array for the display variable. If there is no more room in the data array then more space is obtained. Then the line number associated with the matched key value is zeroed out. The rest of the key list is searched for a duplicate match of all keys unless the file has no children. This is done to catch all duplications at the higher level so that the lower level is not called with the same key values. Whenever a duplicate match is found the data value is placed in the array for the display variable. If the file has no children then REXTRACT looks for more

common keys because it is at the lowest level and it wants to search for all of the common key values before returning to the parent. If a match is not made after comparing the key values then REXTRACT must move the pointer in the key list down one record to examine the next set of key values. The key lists are sorted in such a manner that the last key is sorted within the last -1 key,...; the second key is sorted within the first key. Therefore if a match is sought with a set of keys beginning with the first then the search for further matches after one is found may terminate with the first failure. If a match is sought with a set of keys such that the first key is not included then a sequential search has to be made of the whole key list when looking for matches.

When the keylist is exhausted after looking for all duplications the search moves on to the child; that is if there is another child. If not then a return is made to the parent. If there are more children then the parameters must be set up for the recursive call to REXTRACT A pointer must be set to the display DSECT of the parent, and a pointer to the key values in the parent that must be matched in the child. After the program returns from the recursive call the type of match must be determined. In other words is the match an n to 1, n to n, or n to m match?

When the code returns from the recursive call the number of key values matched in the current file is compared to the number of matches returned. If the match is 1 to 1 then REXTRACT looks for the next match of the common keys with the parent. If only one match is returned but the current file has n matches then the program is made to

repeat the recursive loop with the same key values n-1 times; and hence generate the same value n-1 more times. Then REXTRACT looks for the next match of common keys with the parent. If the recursion failed to produce any match then all of the data values extracted from display variables higher in the tree (those already called in the tree structure) must be deleted. This is done by moving the pointer in the data array back over the correct number of data entries. REXTRACT then returns to EXTRACT to get a new set of keys to search for in the first file. If n values are matched and the current file has only one value then this value is duplicated n-1 times. If an n to n match occurs then another match of common keys is sought with the parent. If an n to m match occurs then an error message is given.

§ 5.2 <u>example</u>

With an algorithm this complex an example should prove helpful. Suppose the following data base exists:

file 1

a	c	v_1
1	1	100
1	2	101
2	1	102
2	2	103
2	2	103

file 2

a	c	b	v_2
1	1	3	200
1	1	4	201
1	2	4	202
1	2	4	202
2	2	3	203
2	2	5	204

file 3

d	b	v_3
1	3	300
1	4	301
2	4	302
2	5	303

<u>figure 16</u>: example of a data base

Where a c are the keys for file 1; a c b are the keys for file 2; and d b are the keys for file 3; and v_i is the value for the i'th display variable which is associated with the record. Of course, each file may have more keys other than those specified; and each file may have

more value fields (maybe even a key) other than the one mentioned. But only one field value is extracted from a file. Remember that even if all the values wanted are in the same file the extraction process treats each value as being in a separate file. Three files have been chosen for simplicity but the algorithm is independent of the number of files. The algorithm to generate a tree structure of relationships generates the following chain: file 1 is related to file 2 through keys a c; and file 2 is related to file 3 through key b. Now everything is set up for the extraction process.

The following set of data values is extracted from the data base in the example. A pointer to the data values for each display variable is stored in the display DSECT associated with the display variable.

v_1	v_2	v_3
100	200	300
100	201	301
100	201	302
101	202	301
101	202	302
102	none	
103	203	300
103	204	303

figure 17: values extracted from the data base.

Let's examine how the data values are extracted. The routine EXTRACT sets a pointer to the key values 11 in the file 1 and to the display DSECT for file 1. The routine REXTRACT is then called. REXTRACT starts at the beginning of file 1 looking for a match to 11. When it finds the match the data value 100 is placed in the data array for display variable v_1. The rest of file 1 is searched for another

occurance of the key values 11. The search is terminated when the key values 12 are found because the key values are sorted. REXTRACT then calls itself recursively looking for a match in file 2 to the common key 11. In file 2 the key values 113 match the parent. Hence the data value 200 is added to the data array for display variable v_2. File 2 is then searched for any further occurance of the complete key values 113. None is found so REXTRACT is again called recursively looking for a match in file 3 with the common key value 3. The key values 13 are found in file 3 which match the common key with the parent. Then the data value 300 is placed in the data array for the display variable v_3. File 3 is searched for any further occurence of the key value 13. Since the recursion cannot be continued by calling children of the parent file REXTRACT then looks for any further occurence of just the common key 3. None is found. Note that the whole keylist must be searched because the key list is not sorted on the key b, therefore the value 3 may appear anywhere. After the end of the key list is reached REXTRACT pops up one level of recursion to file 2. Now the type of match between the parent and the child is examined. It is a 1 to 1 match hence nothing special is done. Now another occurence of the common key value 11 with the parent is searched for in file 2. The key values 114 are found hence the data value 201 is added to the data array for v_2. Another occurence of 114 is searched for in file 2. None is found. REXTRACT calls itself looking in file 3 for the common key 4. The key values 14 are found, and hence the data value 301 is added to the data array for v_3. There are no further occurences of 14. But the current file has no children so REXTRACT searches for any further occurence of the common key value 4. Another record 24 is found, hence

the data value 302 is added to the data array for v_3. There are no further occurences of the common key hence we pop up one level in the recursion. REXTRACT then examines the type of match. It is a 1 to n (n=2) match so that the data value 201 must be copied n-1 times in the data array for v_2. There are no further occurences of the common key 11 so REXTRACT pops up one level to file 1. It notices that there was a 1 to n (n=3) match between file 1 and file 2 so that the data value 100 must be copied n-1 times in the data array for v_1. There are no more occurences of the common key 11 so REXTRACT returns to EXTRACT.

EXTRACT then updates the keylist pointer so that it points to the key values 12, and it calls REXTRACT again. The description of REXTRACT will be shorter this time. REXTRACT finds the key value 12 in file 1 so the value 101 is added to the data array for v_1. REXTRACT calls itself to look for the common key 12 with file 2. The key value 124 is found in file 2 so the data value 202 is added to the data array for v_2. Now a duplicate key value 124 is found, so the value 202 is added to the data array for v_2. But REXTRACT is only called for the first key value 124. REXTRACT calls itself looking for the common key 4 with file 3. The key values 14 are found in file 3, hence the data value 301 is added to the data array for v_3. Since file 3 has no children REXTRACT looks for another occurence of the common key 4 and finds 24, hence the data value 302 is added to the data array for v_3. REXTRACT pops up one level and finds n to n (n=2) match so it does nothing special. Since no other occurence of the common key 12 is found in file 2 REXTRACT pops up one more level to file 1 and finds a 1 to n (n=2) match. REXTRACT then copies n-1 times the data value 101.

Since there are no further occurences of the key value 12 REXTRACT returns to EXTRACT.

EXTRACT updates the pointer in the key list to point to the key value 21. When REXTRACT is called it finds the key value 21 in file 1 so it adds the data value 102 to the data array for v_1. REXTRACT then calls itself to look for the common key 21 with file 2. But no match is found, hence REXTRACT pops up to file 1 and removes the data value 102 from the data array for v_1. REXTRACT then returns to EXTRACT. It should now be quite clear how to extract the data values for the key value 22 in file 1.

§ 6. <u>preparing data for graphical output</u>

The graphics supervisor expects its input to be in a standard format which is used to handle the requests for all of the various types of graphs. All of the data has now been extracted into n data arrays. But this data must be massaged before the graphics supervisor is called. The n arrays of data must be copied into one array for sorting purposes. An array is dynamically created which is large enough to hold all data points. But now some of the data values must be rejected because they are not in the range specified in the DISPLAY command. The 'FRØM' and 'TØ' modifiers must be checked when the data is being copied. If the modifiers are present then they are stored in the display DSECT. The program examines the corresponding values in each of the data arrays to make certain they are in the correct range. If all of the data values are in the correct range then the set of data values is added to the new single array. If any one of the data values in the set is out of range then the set is rejected and the next set is examined.

Then the storage for all of the previous arrays is released. Then a routine called PREPLØT is called, and then the DISPLAY routine returns to the supervisor to accept a new command.

The routine PREPLØT essentially sorts the array into the order demanded by the graphics supervisor, and fills up the information DSECTS used by the graphics supervisor. PREPLØT first sorts the array of data values. Suppose that the following skeleton of the DISPLAY command exists: DISPLAY v_1 BY v_2 BY v_3 BY v_4. Then the data values are sorted in v_4-v_3-v_2 order, in other words the array is first sorted by v_2, then by v_3, then by v_4. The sorting is done by calling a system sort routine. Remember that the graphics supervisor wants the data for each display variable contained within a string DSECT. Hence now that the data array is sorted it must be split up into separate arrays each of which contains only points for one display variable. The location of these arrays is stored within the display DSECT, and storage for the old array is freed. Now the program must fill in the picture, graph, and string DSECTS which were explained in section B of this chapter. Most of the information desired by the picture, graph and string DSECTS is contained within the display DSECT, the file description table, the policy description table, the DISPLAY command, or easily calculated. The only trickly calculation is the graph title. It is created by concatenating the field names together interspaced with the word 'by'; for example the title 'population by time' is created from the field 'population' and 'time'. Since these tables are easy to fill a detailed description is not given.

§ b. the CHANGE command

The CHANGE command is necessary in order to alter the policy vector, and hence allow policy intervention by the user. The syntax of the CHANGE command is:

CHANGE policy-variable-name [TØ] value[[FØR] interval]

where policy-variable-name is any variable in the policy vector, value is the new value for the policy variable, and the interval is the range over which the policy change is to take affect. The range must be over time, which is a current limitation of the system. The interval is expressed as: low-value[[TØ] high-value] where high-value may be ommitted in which case the default value is the end year. The CHANGE command executes two major functions: it invalidates portions of the key lists for all files affected by the policy variable; and it changes the value ranges in the policy vector table. In order to change the entry in the policy vector table the program must indicate there is another value range, and insert the new value range in the appropriate spot in the value range table. If the interval is missing in the command this implies the value range starts at the base year + 1 and ends at the end year. If the interval range falls outside these default limits then it is ignored and the default limits used. The list of files affected by a policy variable is contained within the policy vector table. The key list for each of these files is altered so that the invalidated records cannot be referenced. All the records from the base year + 1 or from the low range in the interval specified in the command, which ever is greatest, to the end year are invalidated.

§ c. the other commands

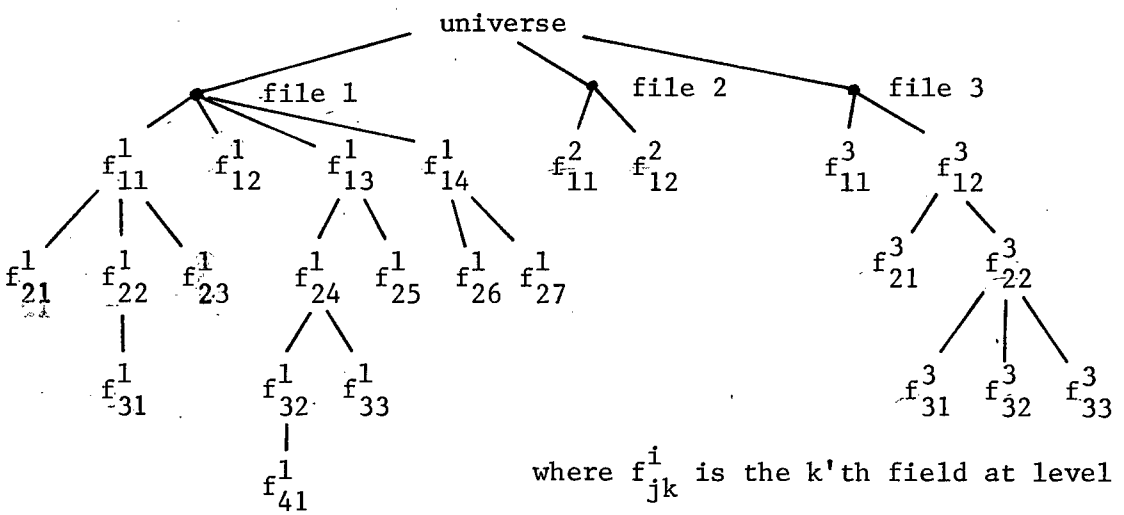
The other commands in the supervisor definitely play second fiddle to the DISPLAY and CHANGE command. These other commands provide the user with functions necessary to complete the system. The implementation of these commands is quite straightforward and is therefore not discussed. The syntax of these other commands is contained in the appendix. Also some of these commands are used in the example contained in the appendix.

CHAPTER IV FUTURE EXTENSIONS TO THE SYSTEM

The present system provides a very powerful tool to aid the development of multiple interacting simulators. But like any other large computer system it should never remain static: there are many changes that could make the system even more powerful and useful. The ideas for some of these changes have been lurking in the background ever since the design of the system; the rest of them evolved through the experience of running the IIPS models. Here are a few of the major extensions that would be useful.

§ A. introducting a complex data base

The largest major change to the system would be the introduction of a complex data base. This change was kept in mind during the design of the system. What form will the complex data base take? The fields in the data base will be related in a tree structure fashion. The data base will still be divided into files; but now the relationship between the fields within each file would be reflected by a tree structure. At present they are related sequentially. For example, the structure of the data base may look as follows:



where f^i_{jk} is the k'th field at level j in file i

figure 18: example of complex data structure

What advantages are there in adopting this more complex data structure? Often one would like to indicate that certain fields are related to each other. For this purpose a tree structure is general enough (a graph is more general but it is not necessary, and it is hard to implement). Within the data structures used at present there is no way to indicate that fields are related because each record is sequential in nature. This complex data structure is not intended to be of use to the simulators - they do not need to know the structure of the data because they read and write sequential records. But the complex data base will allow the user to ask a great variety of questions. He will have an information retrieval system at his fingertips to examine the output of the simulators.

§ a. consequences for the present system

Naturally the introduction of a complex data structure will force changes throughout various parts of the system. These changes will be felt most by the data management interface (hereafter refered to as I/O interface). The changes to other sections of the system will be minimal due to the modularization of the code during the design.

§ a.1 the changes to the major sections of the system and their interfaces with the supervisor

The changes to each section of the system and the changes to each of the interfaces with the supervisor will be examined. The present user commands will not be altered at all; nor will its interface with the supervisor. The only change will be the addition of new information retrieval commands to utilize the additional knowledge that fields are related. The supervisor has been designed so that the addition

of new commands is trivial. These new commands will involve implementing tree touring algorithms to extract wanted information. Hence there will be a great reliance on the new I/0 interface.

The simulators need not change at all. This implies that the I/0 interface will accept a great deal of responsibility. The simulator will expect, as it does now, its I/0 to be sequential in nature. Hence if the simulators do not have to bother adjusting to the new complex data structure then the I/0 interface must transform the complex data record into a sequential record during a simulator read and transform a sequential record into a complex data record during a simulator write. There are many ways to implement such a transformation. This is no trivial task, but at least all of the additional work will be isolated in one section of code. If the I/0 interface were not a separate module then most of the system would have to change.

The graphics supervisor need not change, and neither should the graphics interface. If the results of the information retrieval commands are to be graphed then they should be massaged to fit the current graphics package - it is general enough. Probably most of the output from the new commands will be in tabular form.

Obviously the data base and the I/0 interface will be completely changed. The changes to the data base will be twofold: the logical description will change and the data value structure will change. The logical description is contained in the file description table. Hence it must be changed so that each element in the table will have a pointer to its sister field, its first child, and a back pointer to its parent

(it is assumed that the binary tree representation for a general tree will be used for storage efficiency reasons). The value tree will also have to incorporate pointers within its structure. In the most general case it will have to allow for varying number of repetitions at any level within the value tree structure. But the fact that the simulators produce the entries in the data base makes some aspects of data management easy: all the records are of fixed length and no dynamic deletion or insertion of fields or records will be allowed because the simulator output cannot alter during run time. The simplest method of approach is to stipulate that the simulators produce data for a sequential record that contains all fields in the tree. This change can be made with the minimum of effort. A more general change would be to allow the simulator produce data for all of the children of a field (in other words a subtree within the data base). But this implies changes to the simulators so that they will accept any key as data generation parameter. The simple change mentioned above assumes that the simulators are still called by time, hence time is the universe element in figure 18.

The I/O routine is responsible for reading and writing from the data base. When reading from the data base the I/O must be able to transform the value tree into a sequential record. The simulators will want to read the whole tree; but the supervisor may just want to read a subtree so the I/O routine should be sophisticated enough to handle this case. When writing into the data base from a simulator the I/O routine must place the fields in the sequential record into the proper places in the value tree. Hence the I/O interface will bear the brunt of changing the data structure.

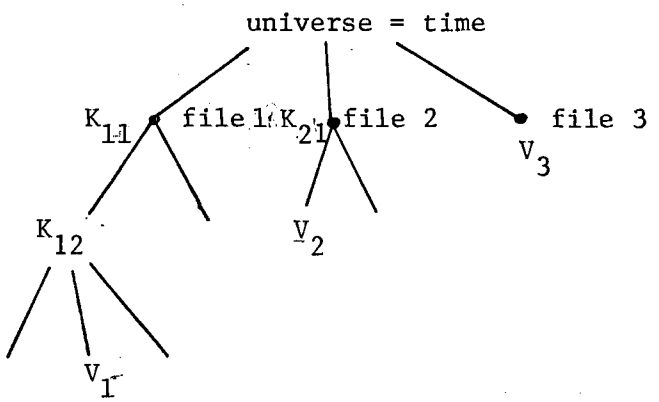
§ a.2 the changes to the DISPLAY command

Discussion of the DISPLAY command occupied much of this thesis therefore the effects on it of changing the data base will be examined. First it should be noted that the only changes to the supervisor hierarchy involve adding more commands, and modifying the routine to load the file description table. The first is trivial, and the second is quite easy to change. The other command subroutines (CHANGE, LIST, SAVE and RESTØRE) that refer to the data base need not be altered because they only refer to the data base through the I/O interface. Within the DISPLAY command the parsing of the command string will not change.

The building of a tree of variable relations between the display variables will be slightly different. The DISPLAY command will still allow cross-file references, but the check for consistency will be different. The program will still look for a common field in both files, but this common field must be the parent of the display variable. For generality the maximal number of previous common parent relations will be sought. This implies that the code which searches through the file description table must be changed, but otherwise the algorithm remains the same. The algorithm to generate data does not have to change at all. This is because the data is generated by calling the simulator, so there is no direct reference to the data base.

The extraction process will have to change a little with the introduction of a complex data base. It will be assumed that the chain of parents from the root of the tree down to the parent of the wanted field are all keys. In this way a sort of sequential

record can be formed from the cahin of parents and the wanted field. Once again there are no major changes since the extraction routine refers to the data base through the key lists and the I/O interface. For example, suppose the following skeleton of the DISPLAY command were given: DISPLAY v_1 BY v_2 BY v_3. Suppose that the following logic tree exists:



where:

K_{ij} are keys

V_i are display variables

figure 19: example of logical tree

This is equivalent to the following sequential file structure:

file 1	file 2	file 3
time k_{11} k_{12} v_1	time k_{21} v_2	time v_3

figure 20: equivalent sequential file

Now in order that the DISPLAY command be consistent the following must be true: $k_{21} = k_{12}$. Then there will be a link between file 1 and file 2, and file 2 and file 3. Hence the additional difficulties created in the extraction process by the introduction of a complex data base are absorbed by the I/O interface and the building of the tree of variable relations. The sorting of the extracted data, and the preparing of data for graphical output will not change.

§ B. removing the restriction that simulators run by time only

At present the simulators must be passed the key value for time in order that they know for which year to produce data. But the concept of calling a simulator by time should be generalized so that it is not dependent on time. One form of generalization would involve calling a simulator with a variety of keys. But this involves major changes to the simulators, in fact it may involve adding more simulation code if the simulator does not produce data over that key. For example, if the population model were asked to produce data for region instead of for time and no data currently exists for region then the simulator must be modified so that it produces these results. Whichever keys were chosen as candidates for the simulator parameter they would have to be common to all the simulators currently under the control of the supervisor because all data generated for a DISPLAY command must be with respect to the same key for consistency.

If simulators could be called with a variety of parameters, then the one chosen by the supervisor would be any one that appeared in the DISPLAY command, or else one that was chosen by default. If one of the keys were in the DISPLAY command then the DISPLAY routine could generate the data points more efficiently. The syntax of the DISPLAY command could be changed so that the user could specify which key will be used as the simulator parameter. This is of some importance to him because even though the key may not appear in the DISPLAY command all of the data will be indirectly generated using values of that key. But the overhead in changing the system is quite high, and probably not worth implementing. A simple method of generalization would be

to only allow one key with which to call the simulators, but read in this key (in the present case 'time') as part of the initialization tables. Since every set of simulators must have at least one such key, but not necessarily time, the system would then no longer be dependent on the IIPS models.

§ C. <u>introducing an arithmetic expression in the DISPLAY command</u>

There will be many times when a user is not interested in the data values of the variables which he is allowed to display. He would rather examine some arithmetic combination of the variables. This is not possible at the present time. It is desirable that each display variable be allowed to contain an arithmetic expression. For example: DISPLAY $((v_1 + v_2 + 3)/v_4)$ BY v_5 BY $v_6 * v_7$.

This will imply changes in the DISPLAY routine, but not very many. The routine that parses the DISPLAY command must be changed significantly so that it can recognize any arithmetic expression. It would probably be a clever idea to store the definition of the arithmetic expressions as a tree structure for later reference. The general idea will be to consider the DISPLAY command to consist of all simple v_i's (display variables) with i=1,...,n where n is the total number of display variables mentioned in the command (including all display variables in the arithmetic expressions). Hence the program is made to have n elements in the display DSECT (previously n had been assumed to be a maximum of four, but this limit was never used explicitly).

The DISPLAY routine then proceeds through the building of a variable tree of relations, the generating of data, and the extracting

of data without any changes necessary. Immediately after the data extraction a section of code must be inserted to calculate the value of the arithmetic expression for each occurence of the data set in the extraction process. This will require the use of the structure for the arithmetic expression stored previously during the parsing of the command. Hence there will now once again be a maximum of four data values in each extracted set of values that have to be examined by the graphics supervisor. The preparation for graphical output may proceed exactly as it did before. Once again an extension can be made without disturbing the present system very much.

§ D. interrupting the DISPLAY command

At present when a DISPLAY command is issued all of the points in the graph are calculated and then the graph is displayed. But it could happen that when a user examines the graph as it is being outputted he will want to interrupt the process for a variety of reasons. It may be that he has typed in the wrong variable, or he decides that he does not want to see the graph for the whole interval originally specified. It may also occur that the user becomes aware that the DISPLAY command he has issued takes a lot of CPU time, and he does not want to incure the expense. This will be true especially if a policy variable is in the DISPLAY command. But at present there is no way of interrupting the process and printing out the graph to date.

A facility is needed so that a user may examine the graph as it is being calculated. But this implies looping through the last sequence of routines within the DISPLAY subroutine. The data generation

of all variables for one time frame would have to be followed by the data extraction of all values, followed by the graphics preparation of the data, and then followed by the graphical output. This implies a great deal of overhead in the execution of the DISPLAY routine, hence this option if available should not be used very often. The process should probably be executed under another command name, with the option of letting the user indicate how many cycles the program is to execute; for example at the end of each cycle the user could type in 'CØNTINUE'.

REFERENCES

[1] Armenti, A., Galley, S., Goldberg, R., Nolan, J., Sholl, A. "LISTAR-Lincoln Information Storage and Associative Retrieval System", AFIPS, Vol 36, 1970 (Spring Joint Computer Conference)

[2] Ash, D., Rashid, Z. "Command Language for IIPS", unpublished document, Department of Computer Science, University of British Columbia, 1971

[3] Boettner, D.W. "Command (Job Control) languages for general purpose computing systems", unpublished paper, University of Michigan Computing Centre, 1969

[4] Day, R.H., Mansfield, M.K., Ellis, M.E. "CØDAS: A Data Display System", CACM, Volume 12, Number 2 (Feb. 1969)

[5] Goldberg, M.A., Wendt, P.F., Walter, G.R. "Jobs, People and Land - Bay Area Simulation Study (BASS)", University of California Printing Department, 1968

[6] Holling, C.S., Goldberg, M.A., Kelly, R. "Vancouver regional simulation study 1970-1971", unpublished paper, Resource Science Centre, University of British Columbia, 1971

[7] Ingram, G.K., Kain, J.F., Ginn, J.R. "The Detroit Prototype of the NBER Urban Simulation Model", preliminary report, National Bureau of Economic Research (NBER), Madison, New York, 1972

[8] Kidd, S.W. "Incorporating Complex data structures into a language for social science research", AFIPS, Vol. 35, 1969 (Fall Joint Computer Conference)

[9] Levy, E. "Simulated Progress? The Vancouver Regional Simulation Project", unpublished paper, Department of Philosophy, University of British Columbia, 1971

[10] Lipner, S.B. "Requirements for the development of computer-based urban information", AFIPS, Vol 34, 1969 (Spring Joint Computer Conference)

[11] Myers, J.E., Chooljian, S.K. "An approach to the development of an advanced information management system", AFIPS, Vol. 36, 1970 (Spring Joint Computer Conference)

[12] Naylor, T.H., Balintfy, J.L., Burdick, D.S., Chu, K. "Computer Simulation Techniques", John Wiley & Sons, Inc., 1966

[13] Parker, J.L. "A graphics and information retrieval supervisor for simulators", AFIPS, Vol. 40, 1972 (Spring Joint Computer Conference)

APPENDIX

§ A. syntax of the command language [2]

The following description gives a brief outline of the command syntax. The text in upper case represents key words which must appear without change unless an abreviation is allowed. The text in lower case within quotes is to be replaced by the appropriate symbol by the user. The text within brackets is optional; hence if the user does not specify it, the enclosed parameters are assigned default values by the system.

§ A.1 CØPY

CØPY FRØM 'filename' TØ 'filename'

This command copies one file to another, like the MTS copy command.

§ A.2 EMPTY

EMPTY 'filename'

This command empties the contents of a file, like the MTS empty command.

§ A.3 CHANGE

CHANGE 'policy variable'[TØ]'value'[[FØR]'interval']

This command allows the user to change the value of a policy variable. The syntax of 'interval' is : $value_1$[[TØ] $value_2$][[BY] $value_3$], where $value_i$ may be real or integer.

§ A.4 DEFINE

DEFINE 'name' AS 'string'

This command allows the user to refer to 'string' through the use of the abbreviation 'name' in a command. The 'name' and its replacement 'string' are placed in a definition table.

§ A.5 DRØP

DRØP 'name'

This command removes a previously defined 'name' from the definition table.

§ A.6 ESTABLISH

ESTABLISH AREAS 'n_1'[TØ] 'n_2'] ON 'device name'

This command divides up the 'device name' into $n_2 - n_1 + 1$ areas and numbers them n_1 through n_2 for later reference. A 'device name' may only be divided into 1, 2, or 4 areas.

§ A.7 DISPLAY

DISPLAY [AS 'name'][SEQUENCED BY] 'v_1' [SEQUENCED] BY 'v_2' [SEQUENCED] BY 'v_3' [SEQUENCED] BY 'v_4' TYPE = 't' [NEW|SUPERIMPØSE] [ØN AREA 'n'][WITH NØ AXES].

This command allows the user to display graphs that are of interest to him. The AS 'name' specification allows the use of abbreviation (without using the DEFINE command) and the ability to name a graph. Each of the v_i's may have range modifiers, for example v_i[FRØM]$value_1$ [TØ] $value_2$ [BY $value_3$]; and may have a rate specified, for example v_i [AT] RATE $value_1$. Only one display variable may have SEQUENCED or AT RATE associated with it. The display variable v_1 may be replaced by a policy variable in which case a bar graph of the policy values is displayed, or by the key word PØLICY VECTØR in which case bar graphs are displayed for every policy variable. The 't' can be one of the following: BAR,DØT,LINE,CØNTØUR, or MAP

§ A.8 HØLD

HOLD [AS 'name'][SEQUENCED BY] 'v_1'[SEQUENCED] BY 'v_2' [SEQUENCED] BY 'v_3' [SEQUENCED]BY 'v_4' TYPE = 't' [NEW| SUPERIMPØSE][ØN AREA 'n'][WITH NØ AXES]

This command allows the user to produce an image but not to display it immediately. It allows a user to create separate images, and then concatenate them together for only one output graph (using the SHØW command). This is convenient for a line printer since it cannot be rewound. The HØLD command has the same syntax as the DISPLAY command.

§ A.9 SHØW

SHØW (AREAS n_1 TØ n_2 | 'device name')

This command allows the user to output on one graph the specified areas where graphs are being held (by using HØLD command). If 'device name' is specified then all of the areas associated with the device are outputted.

§ A.10 ERASE

ERASE (AREA 'n' | 'device name')

This command allows the user to erase the area 'n', or all of the areas on 'device name'.

§ A.11 LIST

LIST (FIELD_NAMES | FILE_ NAMES)[FØR 'class']

or LIST (CLASS_NAMES | USER_NAMES)

or LIST PØLICY_VARIABLES [FØR 'year'][FØR 'class']

or LIST DATA_VALUES [FOR 'data variable'][FOR 'interval'] IN 'file'

This command allows the user to examine the contents of some system tables.

§ A.12 GØ FØRWARD, and GØ BACK

GØ FØRWARD

or 'n'[GRAPHS]ØN AREA 'm' [SHØW ØN AREA 'k'][AT RATE 's']

GØ BACK

The GØ BACK command allows the user to display a previously calculated graph that has been overwritten by another graph. The GØ FØRWARD command allows the user to step forward through old graphs once the GØ BACK command is issued. The show option allows the user to copy the old graph to a new area.

§ A.13 EXPLAIN

EXPLAIN (SYSTEM | 'command' | 'error message number' | 'name')

This command allows the user to ask for a short explaination of various parts of the system. Name is any entry in the definition table or the policy vector.

§ A.14 HELP

This command is intended to help a disoriented user.

§ A.15 MTS

MTS

This command returns the user to MTS mode, during which a $RESTART returns him to the system.

§ A.16 SAVE

SAVE (USER_NAMES | SYSTEM | PØLICY_VECTØR | DATA | 'class name' | 'file name') IN 'myfile'

This command allows the user to save the specified sections of the current run in the predefined MTS file 'myfile'.

§ A.17 RESTØRE

RESTORE (NAMES | 'class name') FRØM 'myfile'

This command restores the information stored by the SAVE command.

§ A.18 SET

SET (RATE = 'n' | 'device name' NEW or SUPERIMPØSE | STØRE = 'number of graphs saved for each area' | YEAR = 'current year' | $$ = 'total dollars' | T = 'total time')

This command allows the user to change the default values set by the system.

§ A.19 CØM

CØM

This command allows the user to place a comment in input data

§ A.20 CØST

CØST

This command allows the user to examine the cost of the run data.

§ A.21 STØP

STØP

This commands ends the execution of the system.

§ A.22 SIGNØFF

SIGNØFF

This command signs a user off the computer.

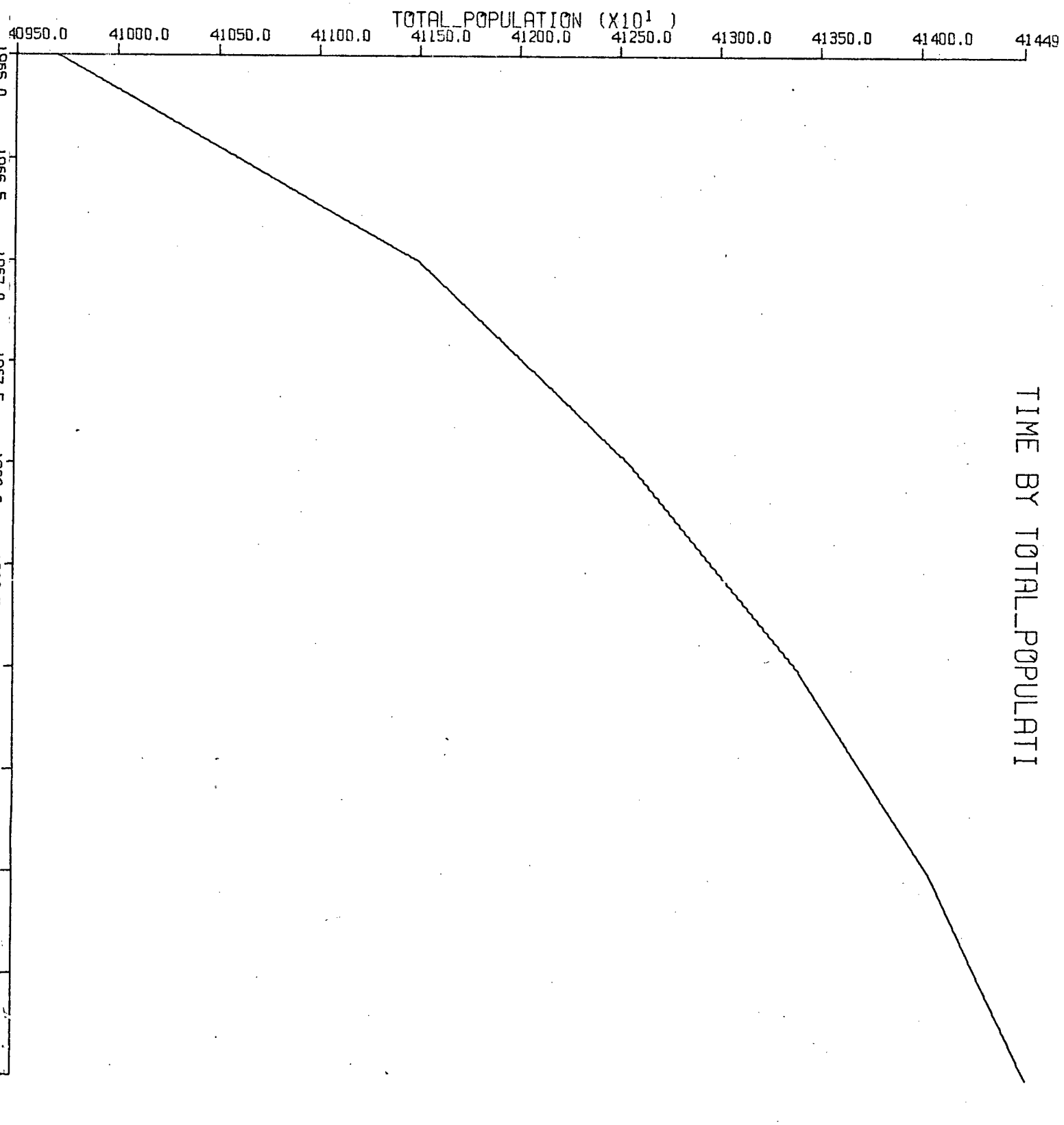
§ B. sample run with graphical output

The following computer print-out shows an example of an actual computer run. The graphs associated with the run are after the end of the computer text.

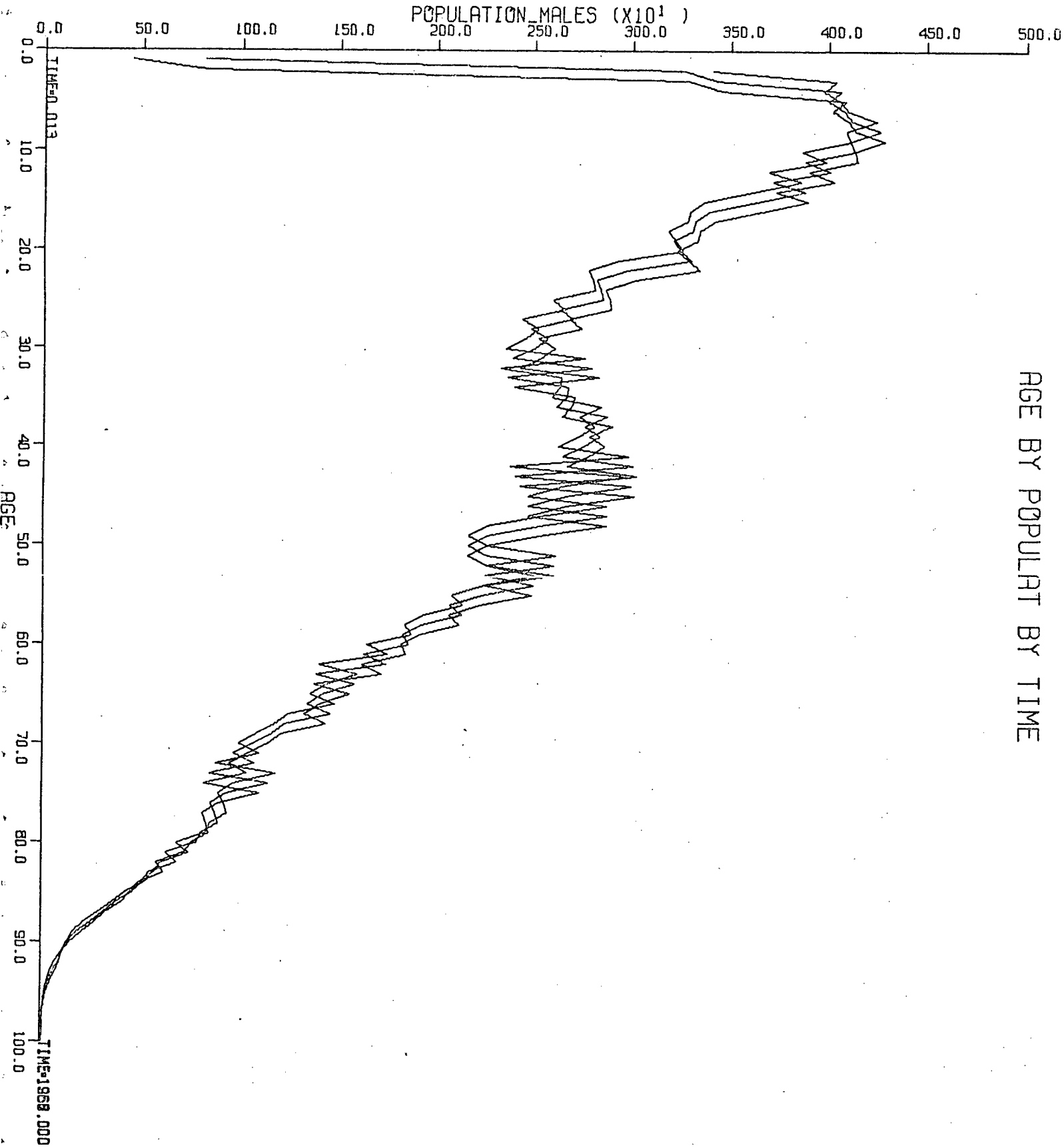
```
                UNIVERSITY OF BRITISH COLUMBIA: MTS(DS11-0054)
#**LAST SIGNON WAS: 13:27:35   WED MAR 29/72
# USER "HORS" SIGNED ON AT  13:29:49 ON WED MAR 29/72
#COM  **  THESIS EXAMPLE  -  ROBERT MCRAE  **
#COM : THE FOLLOWING COMMAND LOADS THE OBJECT MODULE NEEDED TO RUN THE SYSTEM
#RUN SYSTEM
#EXECUTION BEGINS
@LOADING COMPLETED; ENTER COMMANDS
@COM : WHENEVER AREA 1 IS REFERENCED THE IMAGE IS DRAWN ON THE CALCOMP
@ESTABLISH AREA 1 ON CALCOMP
@DONE
@COM : THE MAXIMUM DOLLAR ALLOTMENT FOR THE RUN IS SET TO 10
@SET DOLLARS TO 10
@DONE
@COM : EACH AREA HAS THE PREVIOS FOUR IMAGES SAVED IN A BUFFER
@SET STORE TO 4
@DONE
@COM : THE RESULT OF THE FOLLOWING DISPLAY IS SHOWN IN GRAPH 1
@DISPLAY TOTAL_POPULATION BY TIME FROM 1966 TO 1971 ON AREA 1
@COM : THE PREVIOS DOLLAR ALLOTMENT HAS BEEN USED UP
@****TOTAL DOLLARS EXCEEDED****
@IF YOU WISH TO CONTINUE ENTER:  YES
@YES
@NOW INCREASE THE EXCEEDED QUANTITY USING A SET COMMAND
@SET DOLLARS TO 100
@DONE
@COM : THE RESULT OF THE FOLLOWING DISPLAY IS SHOWN IN GRAPH 2
@DISPLAY POPULATION_MALES BY AGE BY TIME FROM 1966 TO 1968 ON AREA 1
@COM : THE RESULT OF THE FOLLOWING DISPLAY IS SHOWN IN GRAPH 3
@DISPLAY POPULATION_MALES BY AGE BY TIME FROM 1966 TO 1966 TYPE=BAR ON AREA 1
@COM : THE PREVIOS IMAGE , RETRIEVED FROM THE BUFFER WITHOUT NEED FOR
@COM : RECALCULATION , IS SHOWN IN GRAPH 4
@GO BACK 1 ON AREA 1
@COM : THE LAST IMAGE IS SHOWN IN GRAPH 5
@GO FORWARD 1 ON AREA 1
@COM : THE RESULT OF THE FOLLOWING DISPLAY IS SHOWN IN GRAPH 6
@DIS ALL_MUNICIPALITIES TYPE=MAP
@COM : THE SIZE OF THE STAR , WHICH IS USED TO SUPERIMPOSE ONTO MAPS ,
@COM : IS SET TO 35
@SET STAR_SIZE TO 35
@DONE
@COM : THE STARS WHICH INDICATE THE ACCESSIBILITY OF EACH REGION ARE
@COM : SUPERIMPOSED ONTO GRAPH 6
@DIS Y_CENTROID BY X_CENTROID BY ACCESSIBILITY SUPERIMPOSE TYPE=DOT ON AREA 1
@COM : THE CALCOMP IS NOW DIVIDED INTO FOUR AREAS
@ESTABLISH AREA 1 TO 4 ON CALCOMP
@DONE
@COM : THE NEXT FOUR COMMANDS RETRIEVE THE LAST FOUR IMAGES AND DISPLAY THEM
@COM : IN EACH OF THE FOUR AREAS AS SHOWN IN GRAPH 7
@GO BACK 0 ON AREA 1 SHOW ON AREA 1
@GO BACK 1 ON AREA 1 SHOW ON AREA 2
@GO BACK 2 ON AREA 1 SHOW ON AREA 3
@GO BACK 3 ON AREA 1 SHOW ON AREA 4
@COM : THE POLICY VARIABLES FOR THE POPULATION MODEL ARE LISTED FOR 1968
@LIST POLICY_VARIABLES FOR POPULATION FOR 1968
@
@MIN_AGE_FIRST                                    13.000000
@MIN_AGE_SECOND                                   14.000000
@MIN_AGE_THIRD                                    15.000000
```

```
@MIN_AGE_FOURTH                                    16.000000
@MIN_AGE_FIFTH                                     17.000000
@MIN_AGE_SIXTH                                     18.000000
@LOW_AGE_SIXTH                                      1.000000
@HIGH_AGESIXTH                                    100.000000
@MAX_CHILDREN                                       6.000000
@MALE_SURVIVORSHIP_PV                               0.0
@FEMALE_SURVIVORSHIP_PV                             0.0
@RATIO_MALES:FEMALES_AT_BIRTH                       0.500000
@SURVIVING_BIRTH_RATEAT_BIRTH                       0.990000
@MALE_MIGRATION_INCREMENT_PV                        0.0
@FEMALE_MIGRATION_INCREMENT_PV                      0.0
@MALE_MIGRATION_WAVE_PVMENT_PV                      0.0
@FEMALE_MIGRATION_WAVE_PVNT_PV                      0.0
@MALE_MIGRATION_PVWAVE_PVNT_PV                      0.0
@FEMALE_MIGRATION_PVVE_PVNT_PV                      0.0
@
@
@EXPLAIN CHANGE
@    CHANGE 'POLICY VARIABLE' TO 'VALUE' " FROM N1 TO N2 "
@
@         PROVIDES A METHOD OF CHANGING A POLICY VALUE WITH
@         COMPARATIVE EASE.  THIS CHANGE WILL NOT PRODUCE ANY
@         DISPLAYS. THAT CAN ONLY BE DONE WITH A LIST , SHOW OR
@         DISPLAY COMMAND.  POLICY VARIABLE IS ONE OF THE ENTRIES IN
@         THE POLICY VECTOR.
@
@              EXAMPLE: CHANGE POPULATION_GROWTH TO 4
@              THE POPULATION GROWTH IS CHANGED TO 4% FROM
@              CURRENT YEAR TO END YEAR.
@
@              EX.  CHANGE MORGAGE RATE TO 8 BETWEEN 1950 AND
@              1953
@              BETWEEN THE YEARS 1950 AND 1953 THE MORGAGE RATE
@              IS CHANGED TO 8% .
@COM : THE VALUE OF ONE OF THE POLICY VARIABLES IS CHANGED
@CHANGE MIN_AGE_FIRST TO 25 FROM 1967 TO 1969
@DONE
@COM : THE POLICY VARIABLES ARE LISTED AGAIN TO DEMONSTATE THE CHANGE
@LIST POLICY_VARIABLES FOR POPULATION FOR 1968
@
@MIN_AGE_FIRST                                     25.000000
@MIN_AGE_SECOND                                    14.000000
@MIN_AGE_THIRD                                     15.000000
@MIN_AGE_FOURTH                                    16.000000
@MIN_AGE_FIFTH                                     17.000000
@MIN_AGE_SIXTH                                     18.000000
@LOW_AGE_SIXTH                                      1.000000
@HIGH_AGESIXTH                                    100.000000
@MAX_CHILDREN                                       6.000000
@MALE_SURVIVORSHIP_PV                               0.0
@FEMALE_SURVIVORSHIP_PV                             0.0
@RATIO_MALES:FEMALES_AT_BIRTH                       0.500000
@SURVIVING_BIRTH_RATEAT_BIRTH                       0.990000
@MALE_MIGRATION_INCREMENT_PV                        0.0
@FEMALE_MIGRATION_INCREMENT_PV                      0.0
@MALE_MIGRATION_WAVE_PVMENT_PV                      0.0
@FEMALE_MIGRATION_WAVE_PVNT_PV                      0.0
@MALE_MIGRATION_PVWAVE_PVNT_PV                      0.0
@FEMALE_MIGRATION_PVVE_PVNT_PV                      0.0
```

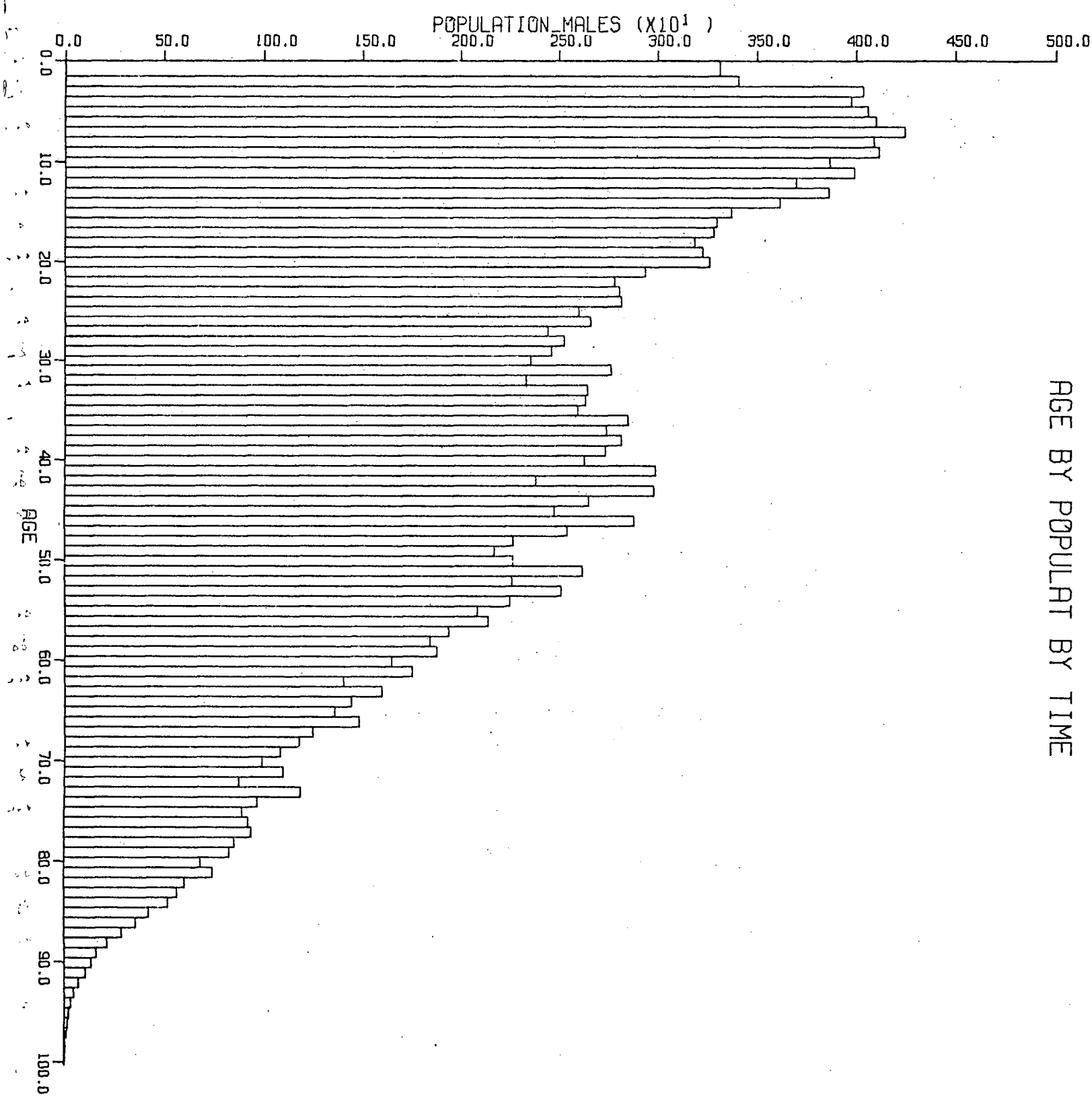
```
@
@
@COM : THE FOLLOWING COMMAND INDICATES THE COST TO DATE FOR THE RUN
@COST
@WAIT MEMORY=30.39 PAGE-HOUR, COST=$12.76
@ACTIVE MEMORY=294.56 PAGE-MIN, COST=$6.87
@ELAPSED TIME =19 MIN 15 SEC, COST=$1.34
@CPU TIME USED=171.76 SEC, COST=$16.69
@TOTAL COST SINCE SIGNON=$37.68 @ 1.4
@COM : THE CALCOMP IS AGAIN ASSIGNED AS AREA 1 ONLY
@EST AREA 1 ON CALCOMP
@COM : THE RESULT OF THE FOLLOWING DISPLAY IS SHOWN IN GRAPH 8
@COM : THE LOCAL GOVERNMENT MODEL , WHICH IS RUN TO PRODUCE THE FOLLOWING
@COM : DISPLAY , CALLS THE POPULATION MODEL FOR SOME OF ITS INPUT
@DIS TRANS_TO_FED_GOVT BY TIME FROM 1966 TO 1976 BY DATA_ITEM FROM 25 TO 25
@COM : THE FOLLOWING COMMAND ALLOWS THE USER TO RETURN TO MTS MODE
@MTS
#COM : A PERMANENT FILE IS CREATED
#CREATE SAVE
# FILE "SAVE" HAS BEEN CREATED.
#COM : ONE OF THE DATA FILES (WITH ITS KEYLIST) IS SAVED
#COPY -POP_INFO(-99999) TO SAVE
#COM : THE FOLLOWING ALLOWS THE USER TO RETURN TO THE SYSTEM
#RESTART
@COM : THE USER TERMINATES THE EXECUTION OF THE SYSTEM
@STOP
@PLOTTING WILL TAKE APPROX.    36 MIN    3 SEC
@SUCCESSFUL PLOT.
@PLOT IN SYSTEM, PLOT TIME    36 MINUTES   3 SECONDS
@ STOP     0
#EXECUTION TERMINATED
```


TIME BY TOTAL_POPULATI
TOTAL_POPULATION (X10¹)
40950.0
41000.0
41050.0
41100.0
41150.0
41200.0
41250.0
41300.0
41350.0
41400.0
41449
TIME
1966.0
1966.5
1967.0
1967.5
1968.0
1968.5
1969.0
1969.5
1970.0
1970.5
1971.0

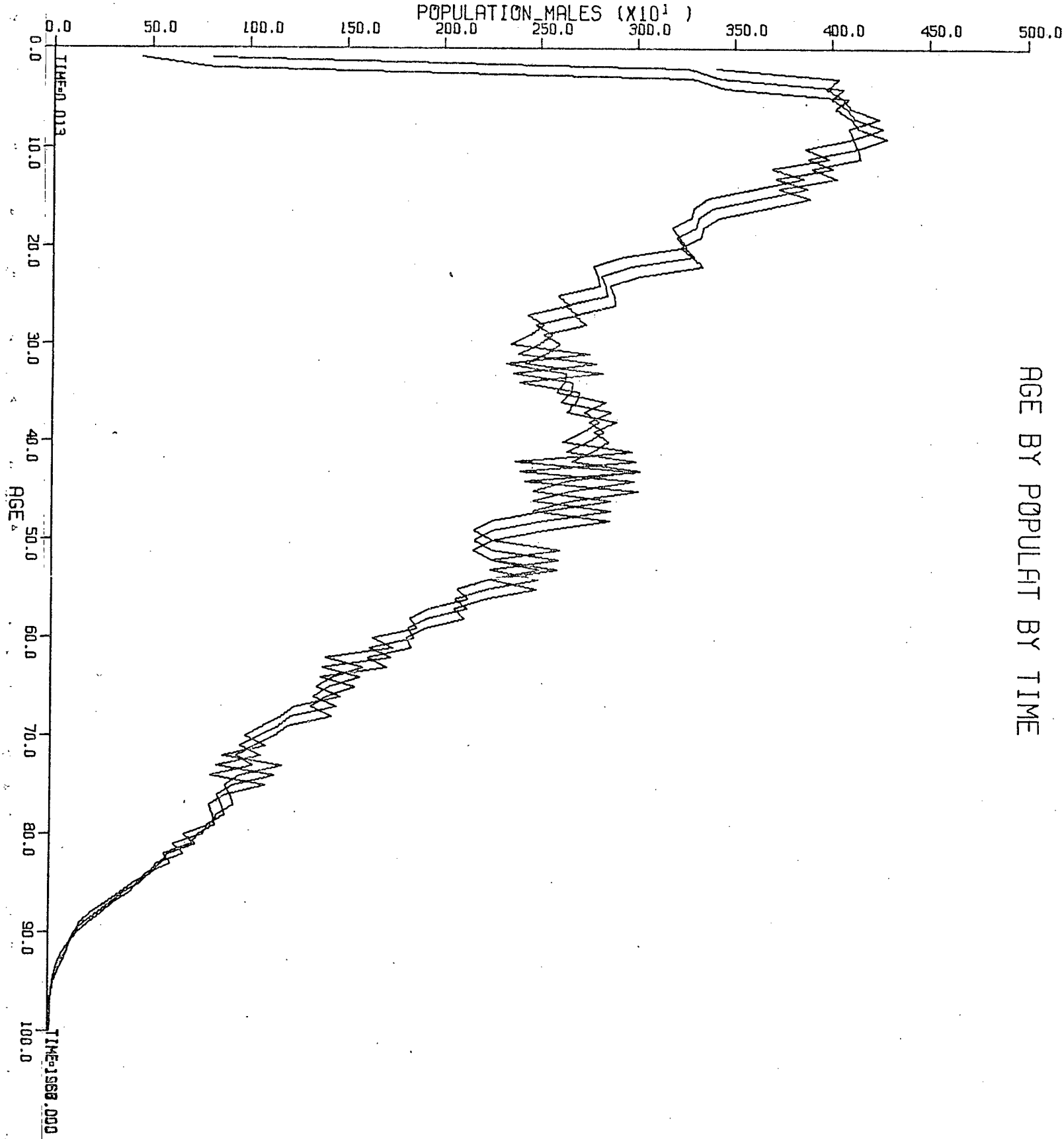
graph 1


AGE BY POPULAT BY TIME
POPULATION_MALES (X10¹)
0.0
50.0
100.0
150.0
200.0
250.0
300.0
350.0
400.0
450.0
500.0
AGE
0.0
10.0
20.0
30.0
40.0
50.0
60.0
70.0
80.0
90.0
100.0
TIME=0.013
TIME=1968.000

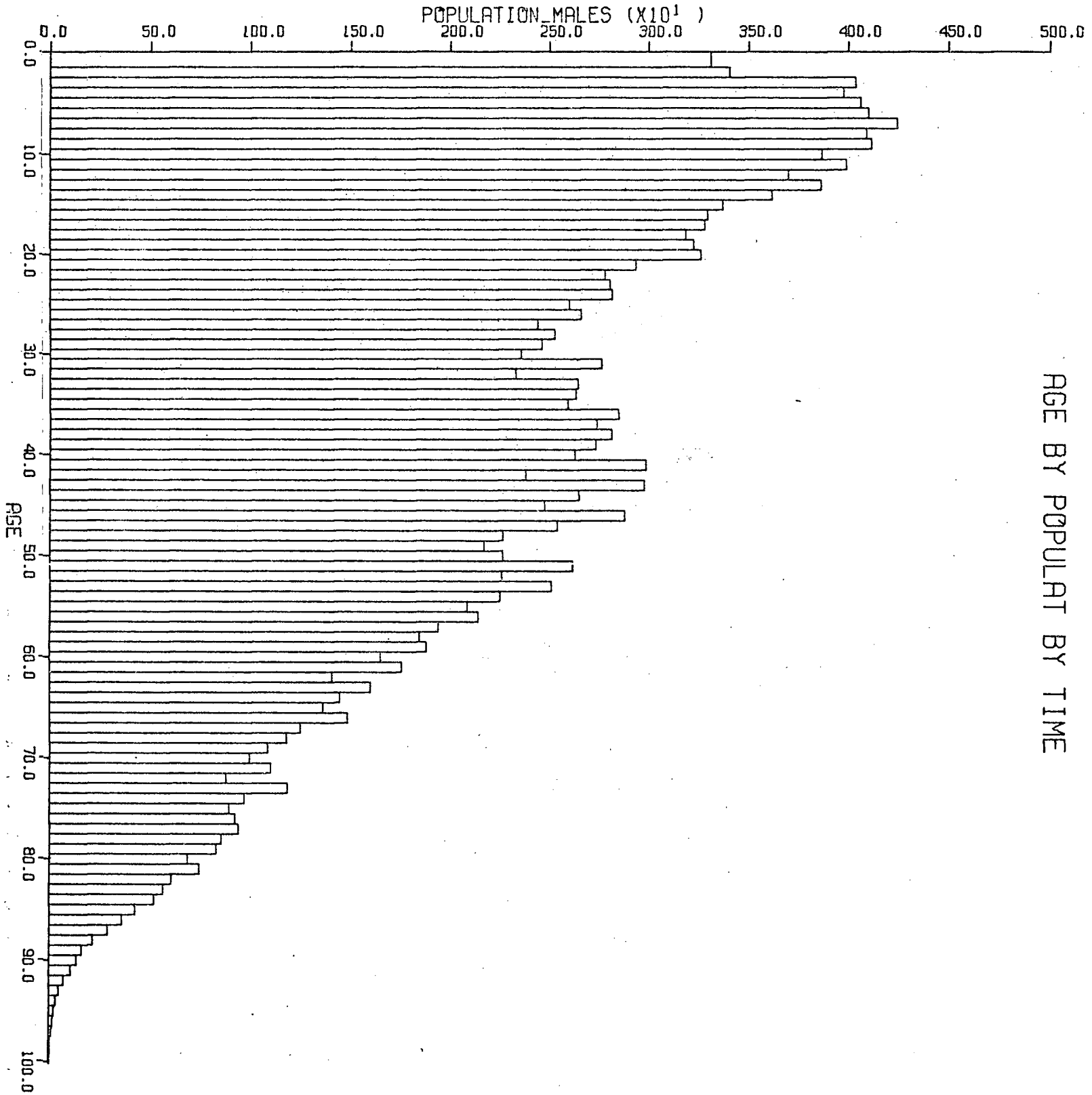
graph 2


AGE BY POPULAT BY TIME
POPULATION_MALES (X10¹)
0.0
50.0
100.0
150.0
200.0
250.0
300.0
350.0
400.0
450.0
500.0
AGE
0.0
10.0
20.0
30.0
40.0
50.0
60.0
70.0
80.0
90.0
100.0

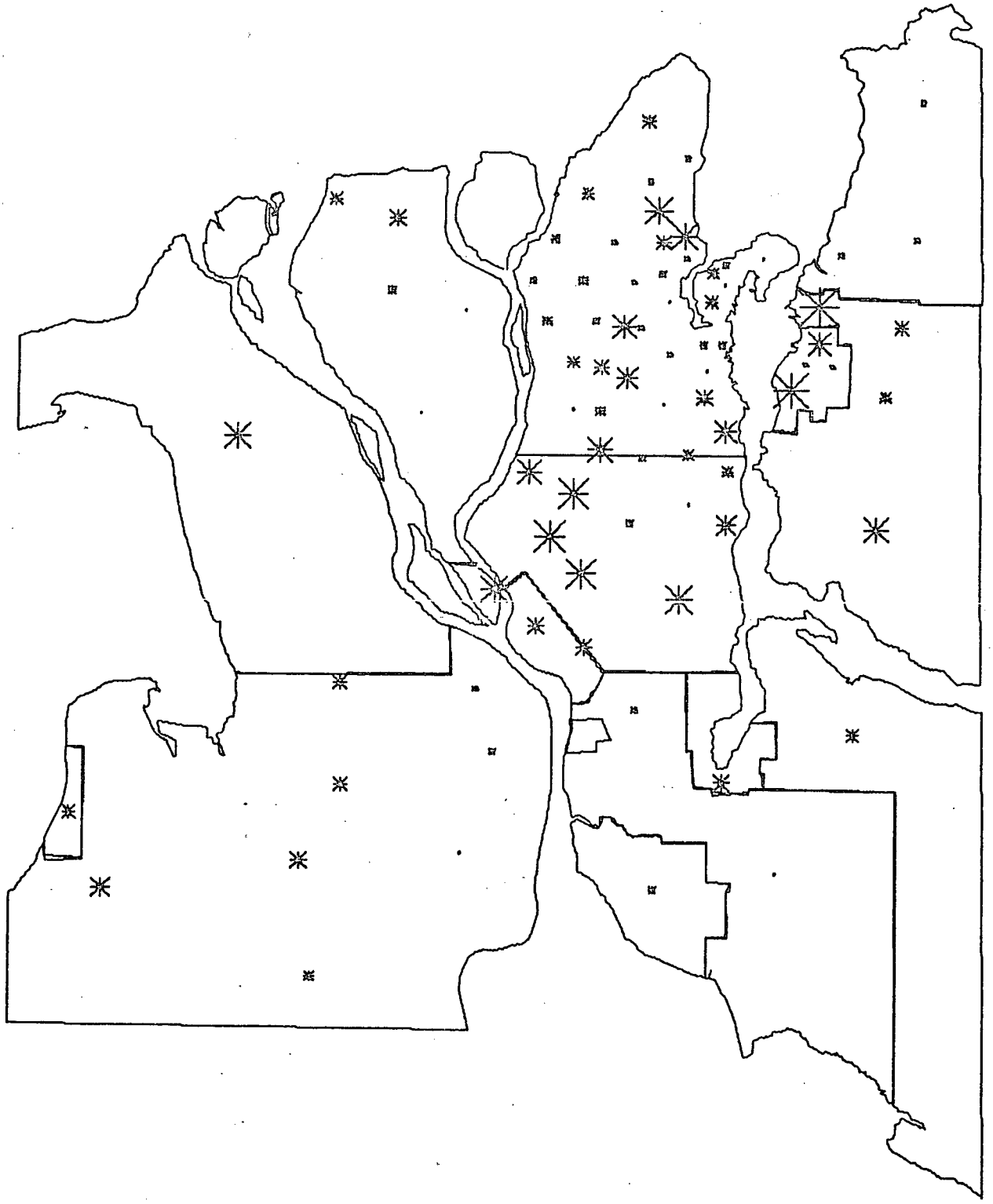
graph 3


AGE BY POPULAT BY TIME
POPULATION_MALES (X10¹)
0.0
50.0
100.0
150.0
200.0
250.0
300.0
350.0
400.0
450.0
500.0
AGE
0.0
10.0
20.0
30.0
40.0
50.0
60.0
70.0
80.0
90.0
100.0
TIME=0.013
TIME=1968.000

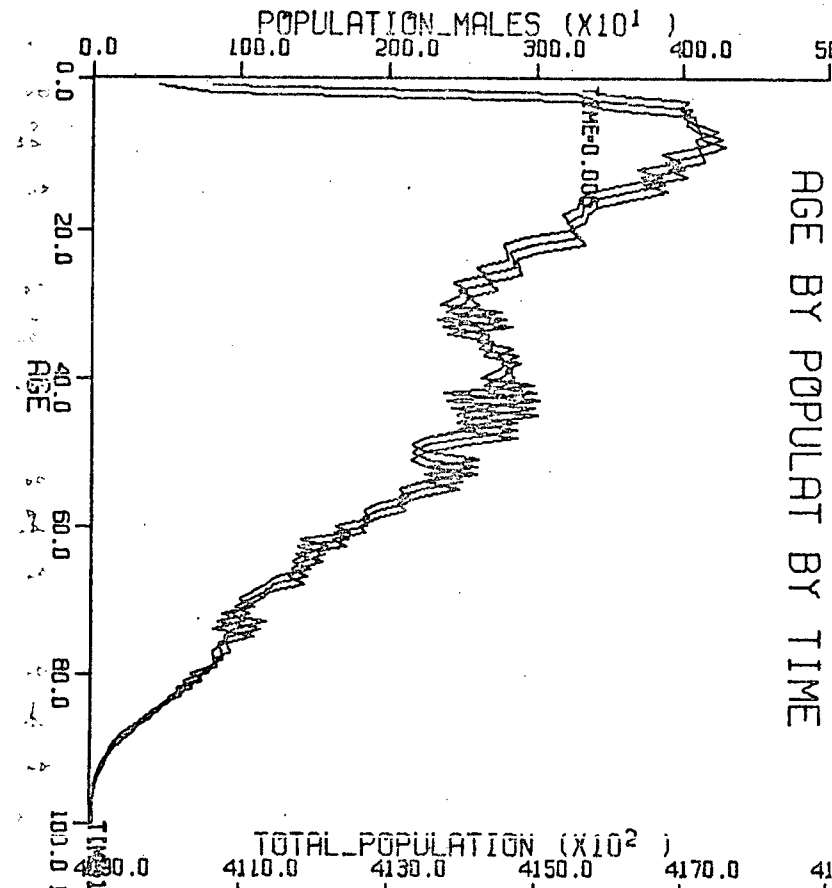
graph 4


AGE BY POPULAT BY TIME
POPULATION_MALES (X10¹)
0.0
50.0
100.0
150.0
200.0
250.0
300.0
350.0
400.0
450.0
500.0
AGE
0.0
10.0
20.0
30.0
40.0
50.0
60.0
70.0
80.0
90.0
100.0

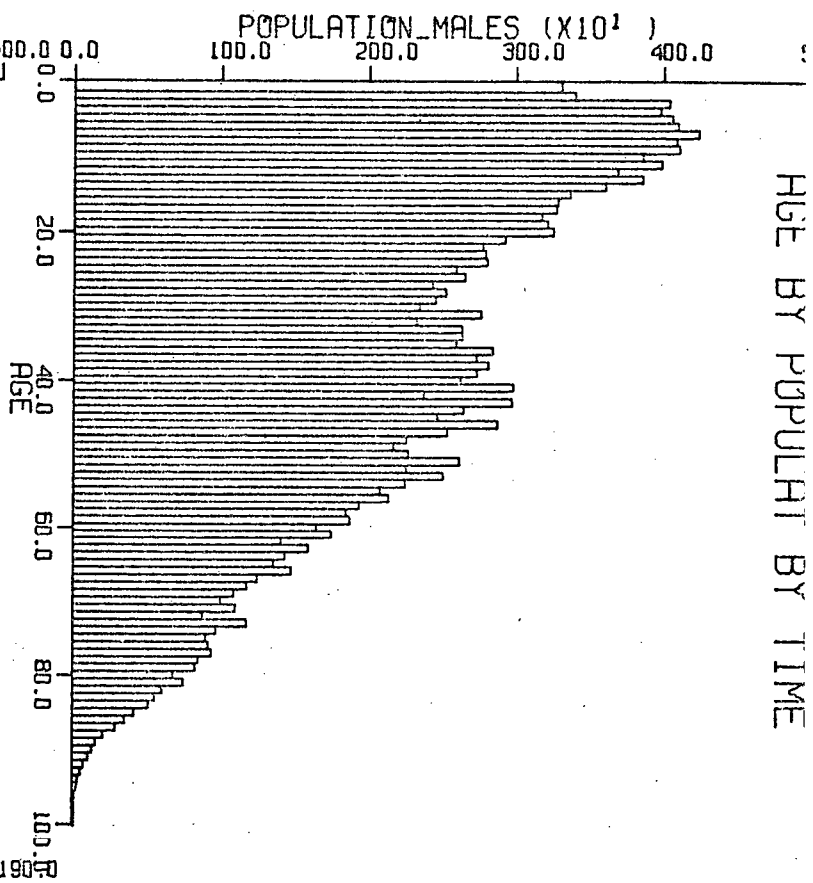
graph 5



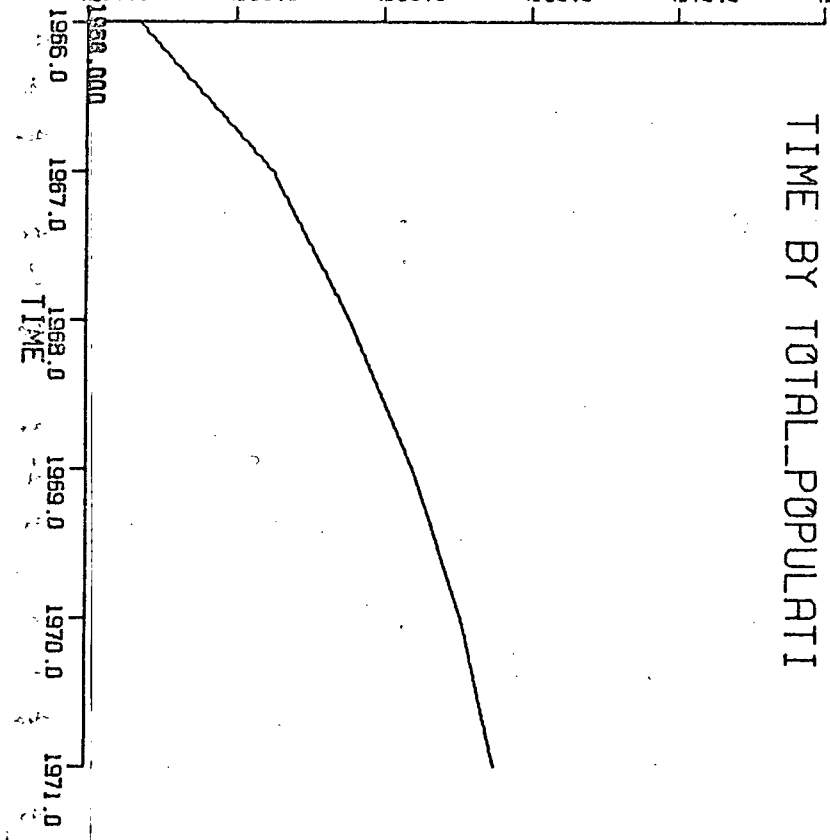
graph 6



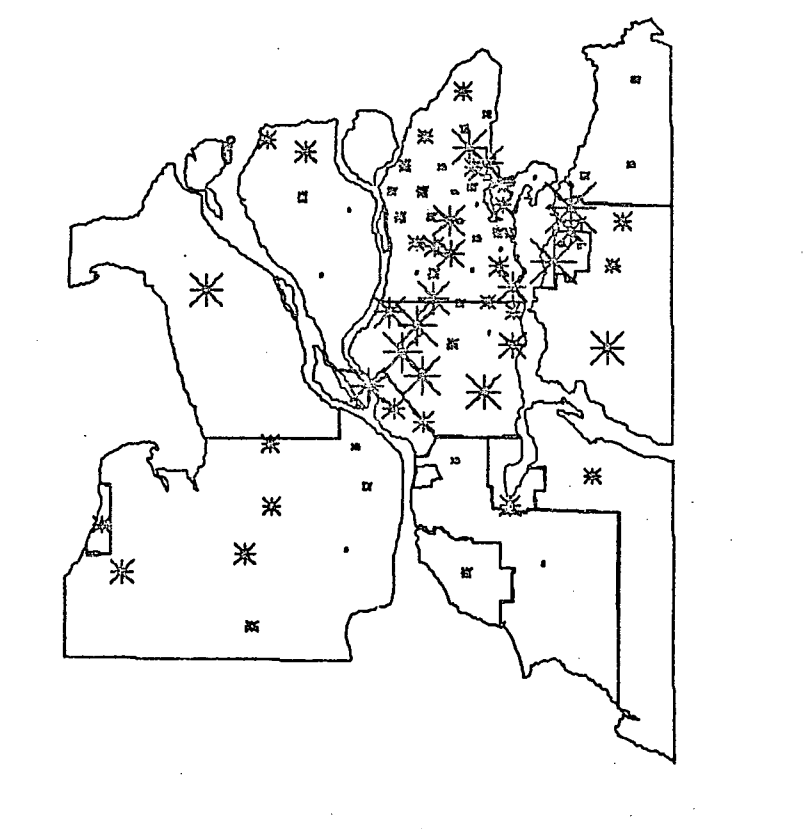
AGE BY POPULAT BY TIME



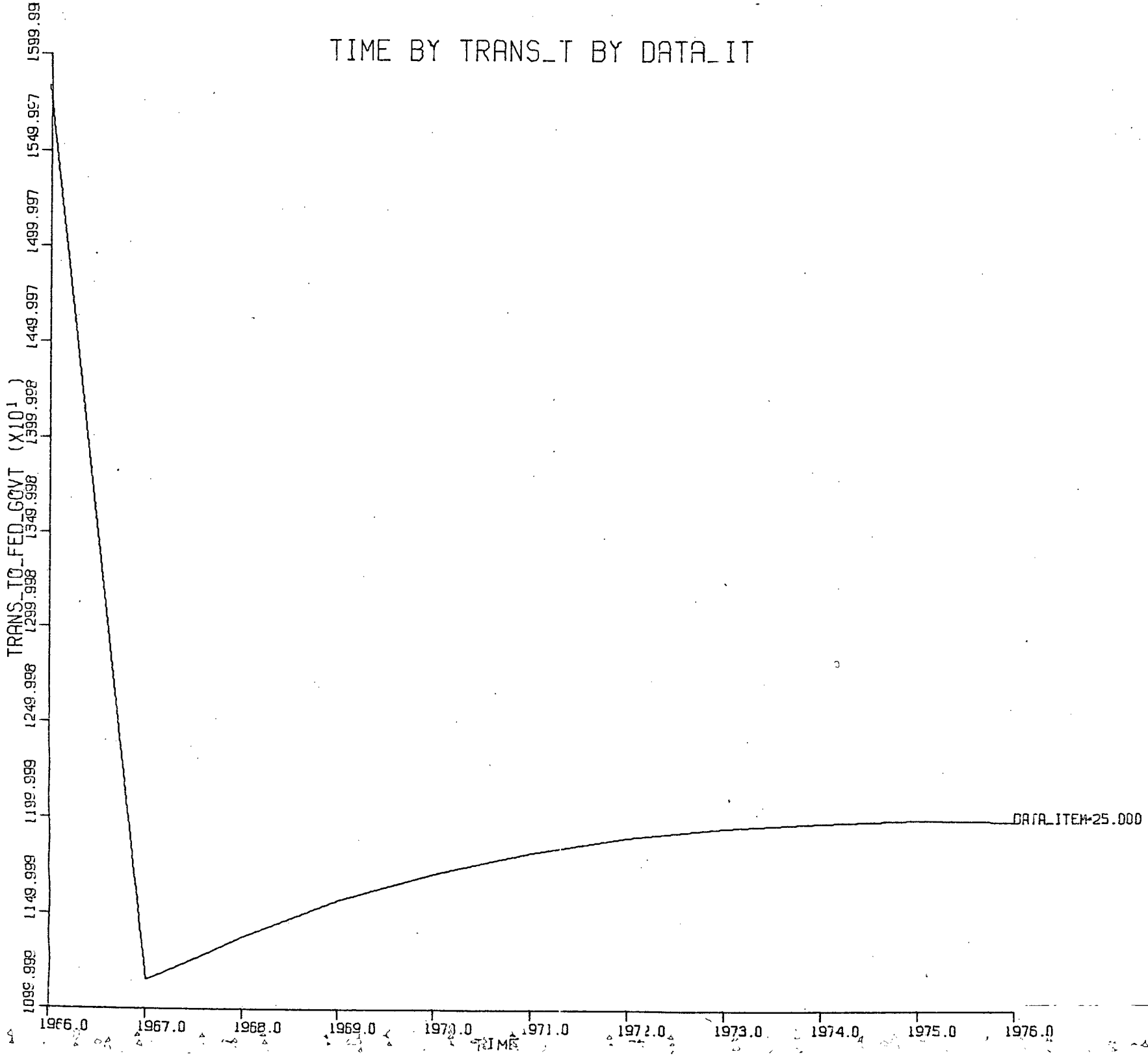
AGE BY POPULAT BY TIME



TIME BY TOTAL_POPULATI



graph 7


TIME BY TRANS_T BY DATA_IT
TRANS_TO_FED_GOVT (X10¹)
1599.99
1549.997
1499.997
1449.997
1399.998
1349.998
1299.998
1249.998
1199.999
1149.999
1099.999
1966.0
1967.0
1968.0
1969.0
1970.0
1971.0
1972.0
1973.0
1974.0
1975.0
1976.0
TIME
DATA_ITEM=25.000

graph 8