TRANSFORMING DATA FLOW DIAGRAMS TO SOFTWARE STRUCTURE USING THE YOURDON-CONSTANTINE METHODOLOGY

by

FRANCO ANTONUCCI

B.S.E.E., Fairleigh Dickenson University, 1975

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

Approved by:

David A. Gustafson

Major Professor

TRANSFORMING DATA FLOW DIAGRAMS TO SOFTWARE STRUCTURE USING THE YOURDON-CONSTANTINE METHODOLOGY

THIS BOOK CONTAINS NUMEROUS PAGES WITH THE ORIGINAL PRINTING BEING SKEWED DIFFERENTLY FROM THE TOP OF THE PAGE TO THE BOTTOM.

THIS IS AS RECEIVED FROM THE CUSTOMER.

THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE.

THIS IS AS RECEIVED FROM CUSTOMER.

LIST OF FIGURES

Chapter 1 - Overview

1.1 Introduction

The "Information Age' is a phrase coined in the last decade to describe the revolution that has taken place in the computer field. Larger, faster and less expensive computers have been built. However, as the hardware technology has taken giant leaps, the software technology has been stumbling and is in a state of confusion.

Guidelines were not available or not used by people developing software systems. Each corporation was using their in-house methodologies and sometimes even these were not being followed by all. Management's thinking was that if guidelines were enforced, it would take longer to develop systems. Therefore, guidelines were rarely enforced. Employees did not follow guidelines because guidelines were not easy to follow and they were often not given time to learn them. Systems developed without guidelines were poorly designed and often, once implemented, required extensive maintenance.

Out of all this confusion, ideas like 'structured programming' and 'Structured Design' (1) emerged. There was actually going to be an attempt to bring some order to this 'science' that up to that point seemed to be uncontrollable. Well at times it seems to still be under disarray but a conscious effort has

been made by both the academic and private sector in improving and organizing this 'science'.

The term 'structured programming' has been used with many different meanings. It is a way of developing programs to make them more readable and easier to understand. The goals of structured programming are to devise orderly, efficient methods for developing readable correct programs, and to identify and explain tools and techniques for solving programming problems (12). One of the design methodologies that emerged was called Structured Design. This methodology is better known today as the Yourdon-Constantine design methodology. The methodology was introduced in 1974 in a paper by W.P. Stevens, G.J. Meyers and L.L. Constantine (1) as a response to the continuing increase in cost for programmers' time. The authors felt that simplicity was the key to designing systems that would require less debugging and modification time. They also stipulated that systems can be divided into separate pieces in such a way that these pieces can be considered, implemented, fixed and changed with minimal considerations of effect on the other pieces of the system. The methodology is based on the concept that systems should be divided into simple, independent modules.

This project is one of a group of projects at Kansas State University that deal with automated development tools for the software life cycle. Using the concepts of the Yourdon-

Constantine methodology, this project will generate a 'first' cut software structure using data obtained from the data flow diagram. In order to arrive at a final design, the generated structure will have go through extensive refinements. It is planned that the input for this project will be obtained automatically from another project which is creating a Data Flow Diagram from an Entity-Relationship-Attribute (ERA) specification.

1.2 Fifth Generation

A subject of discussion among the computer scientists is the Fifth Generation of computing. A big stumbling block in the development of a new generation of computers and of information systems for these computers is software engineering (2). There will be a need for automated tools for every phase of the software development life cycle, from requirements specification generators to code generators to test data generators.

System design is a major part of the system development cycle. It is in the design phase that software is structured. Decisions are made about the hierarchy of modules and what each modules should do. These decisions are important since good modular-designed systems are more reliable, easier to maintain and to modify. Errors detected in this part of the development cycle are usually more easily corrected than

errors detected during or after implementation.

It is estimated that between 40 to 60% of all software development dollars is spent each year in conducting corrective, adaptive, perfective and preventative maintenance (3). Most of these problems can be eliminated if a system is properly designed and possibly designed with automated tools.

1.3 Design Approaches

The Yourdon-Constantine design methodology was introduced in 1974 (1) as Structured Design. The methodology was introduced at about the same time that other major methodologies were introduced such as the Jackson (3) and the Warnier-Orr (3) methodologies. Both the Jackson's and the Warnier-Orr methodologies are based on the concept that the structure of the system or software should be determined by the structure of the data. The Jackson methodology states that problems should be decomposed into hierarchical structure of parts that may represented by the three structural forms: sequence, repetition and condition. The Warnier-Orr design methodology is also a data structure oriented design methodology. However, the Warnier-Orr methodology stresses that the output should be the place to start in defining a system.

1.4 Yourdon-Constantine system design methodology

The Yourdon-Constantine methodology is based on the concept that the structure of the system should be determined by the flow of data through the system. In order to describe the flow of data through a system, a data flow diagram is used. In general, a data flow diagram is a network representation of a system (4). The system may be automated, manual or mixed.

Data flow diagrams (fig. 1.1) portray the system in terms of its component pieces, with all interfaces among the components indicated. Data flow diagrams are sometimes called Data Flow Graphs or "bubble" charts. Data flow diagrams are generated during the requirement analysis phase of the system development life cycle. Data flow diagrams are made up of four basic elements (4): 1) data flows, represented by named vectors; 2) processes represented by circles or bubbles; 3) files, represented by straight lines and 4) data sources and sinks, represented by boxes.

A data flow is a pipeline through which packets of information of known composition flow. A process accepts an incoming data flow and transforms it into an output data flow.

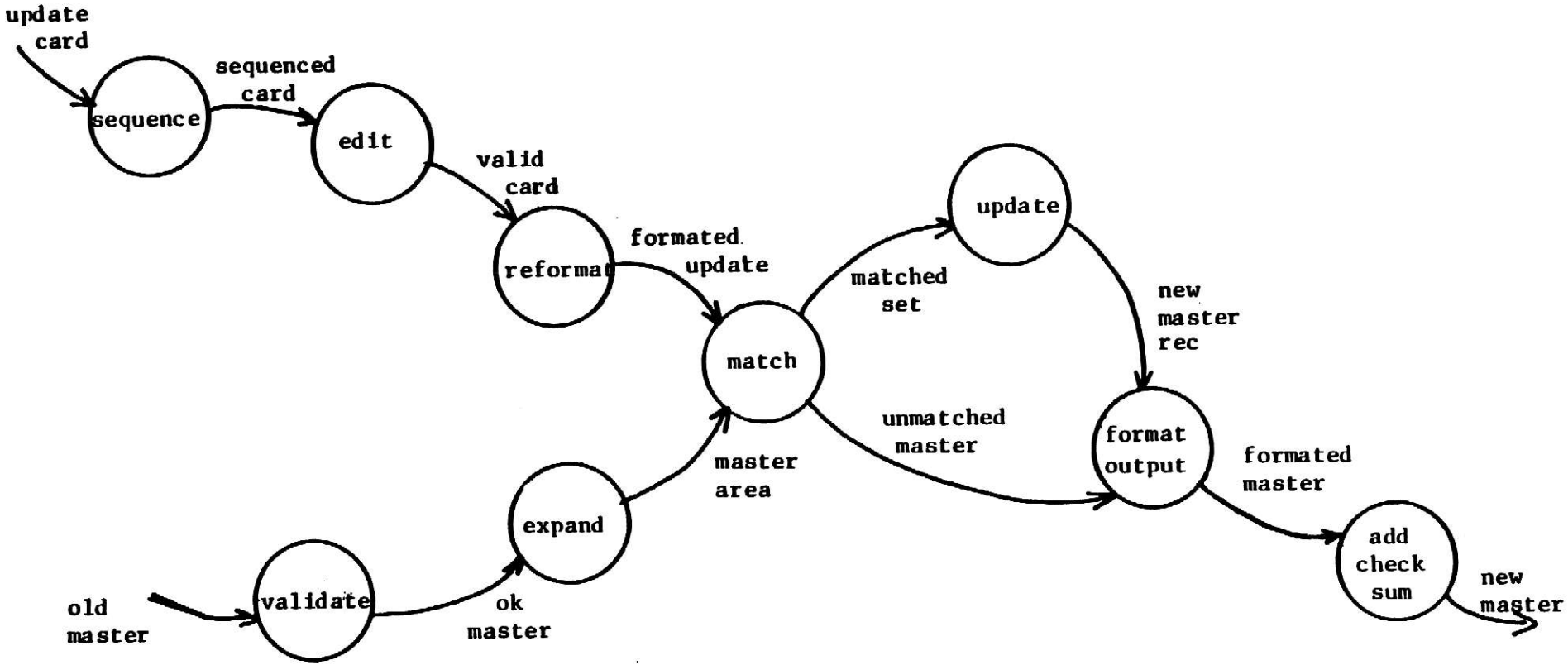

Figure 1.1 - DATA FLOW DIAGRAM

A file is temporary or permanent storage of data. The data could be stored on magnetic tapes, disk or any other data storage device. A source or a sink can be an organization or an individual that generates or receives data from the system. A user of a system is usually considered a source or a sink.

The Yourdon-Constantine methodology divides systems into two groups 1) transform centered and 2) transaction centered. This division of the system is required since the methodology treats each group differently. Transform centered systems are systems that have clearly identified input streams, central processing and output stream (4). Transform centered systems usually are very adaptable to this methodology. Transaction centered systems are systems that 'represent a data item that on evaluation triggers additional flow along one of a number of selected paths' (3). In these types of systems some information is evaluated and based on the result of the evaluation a path is selected for the data to follow.

1.5 Transform Analysis

Yourdon and Constantine in their book (5) define transform analysis as 'a strategy for deriving initial structural designs that usually are quite good (with respect to modularity) and generally require only a modest restructuring

to arrive at a final design'. The important word in the above definition is 'strategy' meaning that by following some set of procedures you will obtain good results but not necessarily perfect results.

Transform analysis divides a system data flow diagram into three types of data elements: the afferent, efferent and the transform (9) (fig. 1.2). The afferent data elements are those high-level elements of data that are furthest removed from physical input, yet still constitute inputs (5). Efferent data elements are those furthest removed from the physical outputs which may still be regarded as outputs. Transform data elements are those elements involved in the transformation of the inputs into outputs or afferents into efferents.

There are several steps required to derive a software structure from a data flow diagram (6). The steps are: 1) draw a data flow diagram; 2) identify all the major input and output streams; 3) identify the point were each input stream can no longer be considered an input; 4) identify the point were each output stream can no longer be considered output; 5) identify the central transforms; 6) draw the top two levels of the structure chart; 7) factor the second-level input, output and central transform modules; 8) continue to factor until the entire problem has been depicted in the structure chart.

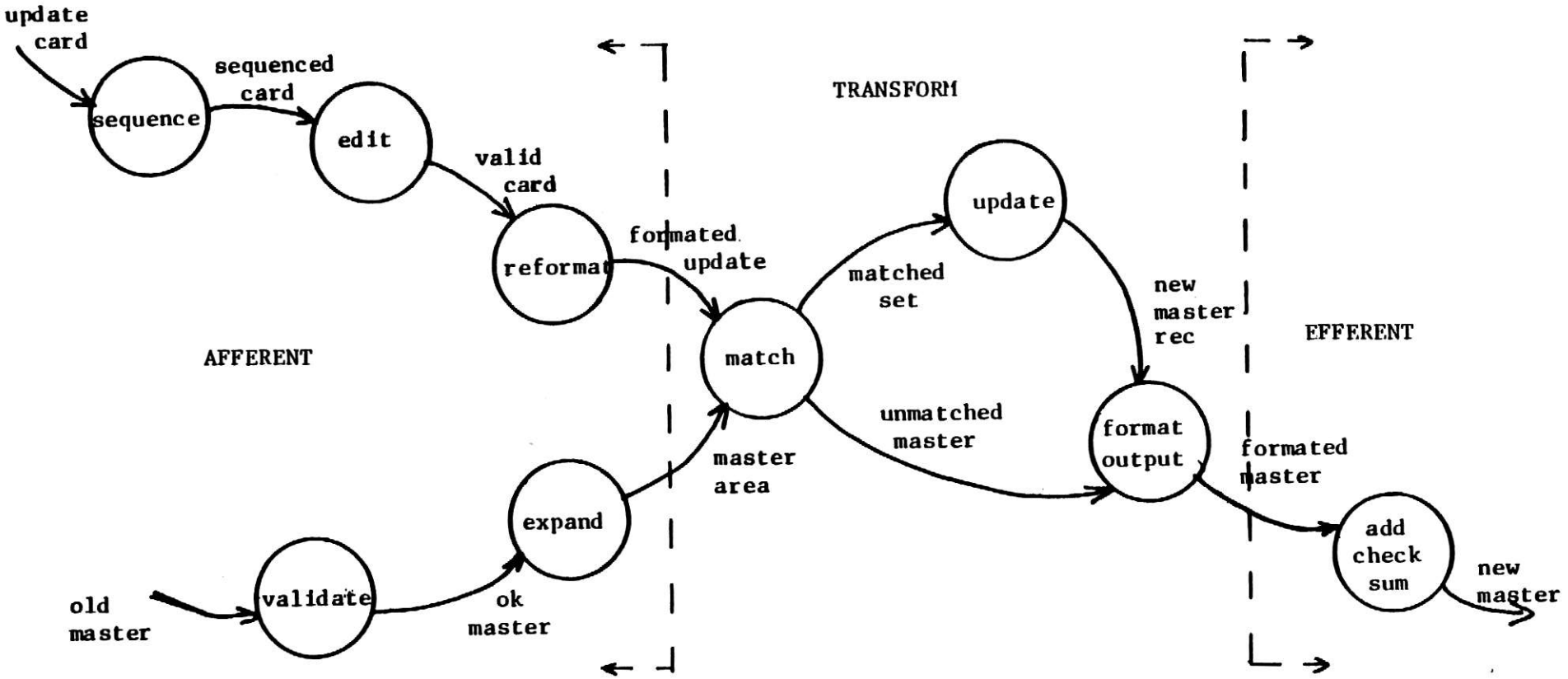

Figure 1.2 - PARTITIONED DATA FLOW DIAGRAM

The data flow diagram is derived from the requirement specification. The data flow diagram describes the system by using the four basic elements dataflows, processes, files, data sources and sinks. Drawing the data flow diagram is at times difficult and time should be spent in refining the original diagram until a diagram has been drawn that best describes the desired system. Once the data flow diagram is drawn one can proceed to the next step of transform analysis.

One of the most difficult parts of the Yourdon-Constantine methodology is to find where the input and output data streams can no longer be considered inputs and outputs. It is at this point that the data flow diagram is partitioned into the afferent, efferent and transform.

After the data flow diagram has been partitioned the top two levels of the structure chart can be drawn. The first level is the main module for the system. The second level is the top module for each of the branches (fig. 1.3).

On the second level module for the afferent and efferent branches, a third level module is attached for each major input and output path. On the transform's second level module, a third module is attached for each 'bubble' in the transform branch (fig. 1.4).

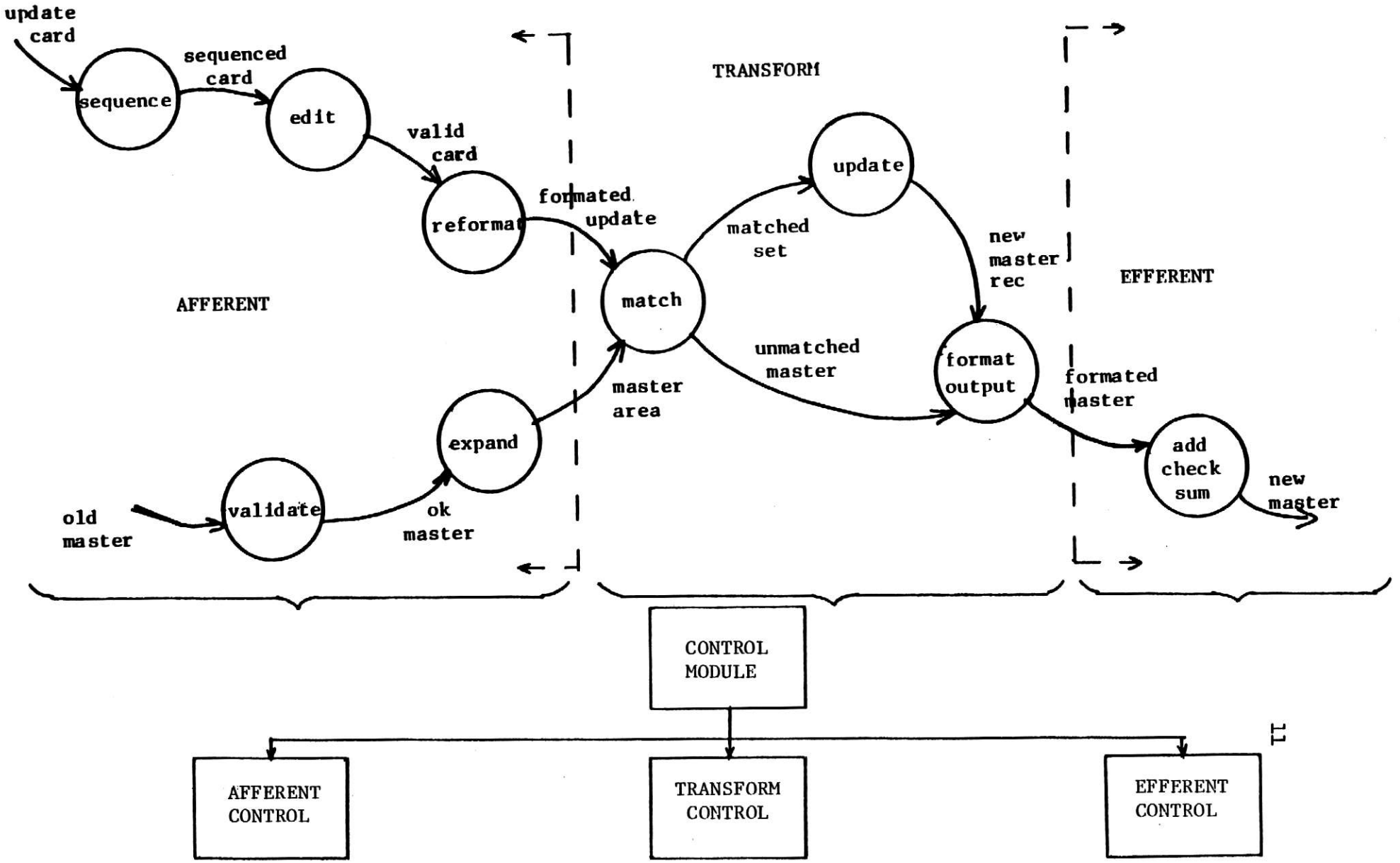

Figure 1.3 - FIRST LEVEL FACTORING

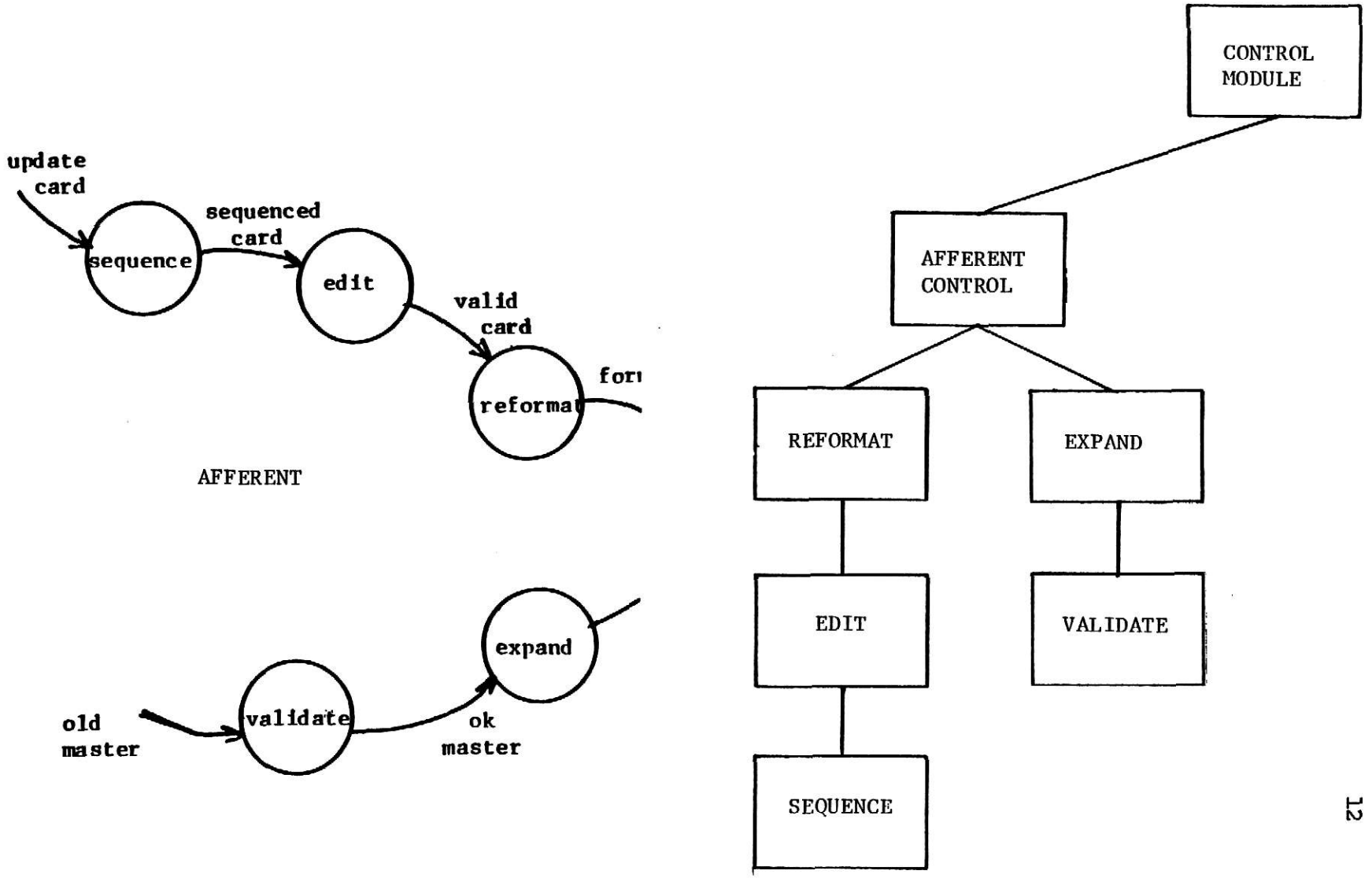

Figure 1.4 - SECOND LEVEL FACTORING

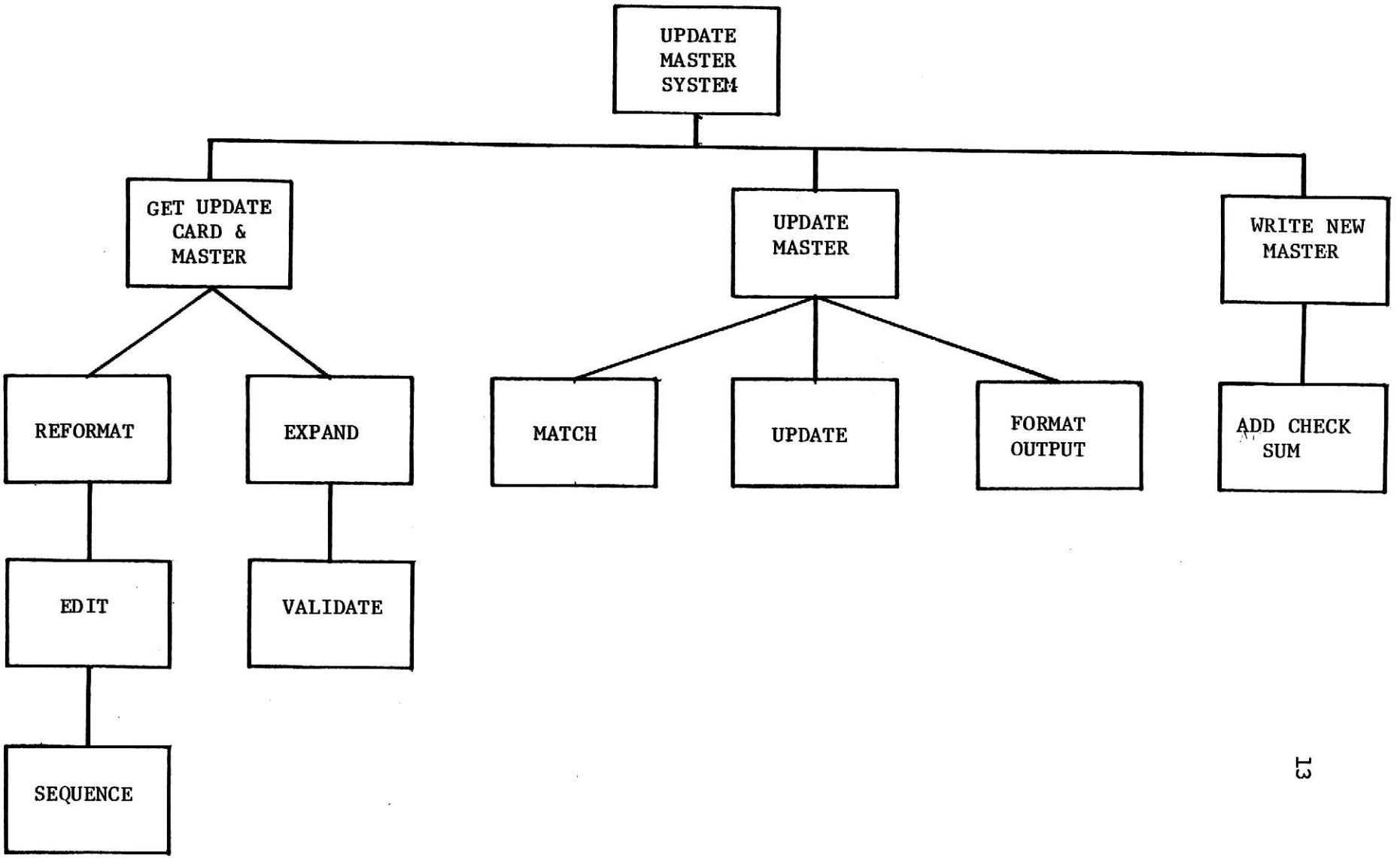

Figure 1.5 - SOFTWARE STRUCTURE

The factoring of modules continues until all modules have been broken down to their lowest submodules to a point where the design can be called a final design (fig. 1.5).

This project will automate the first part of the factoring process, the one that determines the general 'shape' of the software structure.

1.6 Summary

In comparing this methodology with other design methodologies in use today, mainly the Jackson and the Warnier-Orr methodologies, each seems suited for different types of systems. Both the Jackson and Warnier-Orr methodologies are best suited for systems where all the data structures in the program are compatible.

The Yourdon-Constantine seems well suited for systems where a data flow diagram can be easily drawn. One of the reference states (10) 'We find that this method and particularly its graphics do reveal previously unknown properties of some systems ------ This method turns out to be well suited to design problems where a well defined data flow can be derived from the problem specification'. The Jackson and Warnier-Orr are well suited for system that have well defined data structures (7).

One of the advantages of the Yourdon-Constantine methodology

is that it usually produces system designs that are easy to develop and to maintain (6). This methodology also ensures the correctness of a design at an early stage reducing the possibility of having to discard premature coding as the system is factored into lower level modules (7). By deriving a high modular design which the methodology encourages, maintenance of the system is usually easier (7).

Some of the drawbacks discussed below, provide an incentive for mechanizing the methodology. Some feel (11) that partitioning of a system into afferent, transform and efferent branches is artificial and that deriving the correct data flow diagram is still an art. In mechanizing the methodology, stricter rules may be enforced in partitioning a system.

The methodology has been described as hard to teach and that a highly-skilled work force is required for the methodology to be successful. By mechanizing the methodology, even by one or two steps as this project will do, some of the difficulty in teaching and learning the discipline will be relieved.

In researching the literature for this project, the author has noticed a lack of literature on attempts or even discussions on mechanizing the methodology. In one of the references (8) it is even mentioned that the methodology cannot be mechanized due to the lack of sophistication in available languages and operating systems.

This project is the mechanization of one of the steps of transform analysis. The author believes that this project should be continued. Future projects should be able to further mechanize this methodology for easier use and better system design.

Chapter 2 - Requirements

2.1 Overview

In order to be successful and useful this project must produce a textual and tabular representation of a software structure derived from a data flow diagram using the transform analysis concepts specified by the Yourdon-Constantine design methodology. The tabular output will show the hierarchical format of the modules in the system. In addition, there will be an output that will describe each module individually.

The Yourdon-Constantine methodology is based on the premise that the flow of data should determine the structure of the system. The methodology uses the data flow diagram which has been partitioned into afferent, efferent and transform branches as input.

2.2 Input

This project will use, as its input file, information derived from the data flow diagram. This file may have been manually created or may have been automatically generated by a mechanized system that generates a data flow diagram from a requirements specification. The input file will contain the module name, its input(s) and output(s) (Appx. A). It will also contain whether the module is part of the afferent,

efferent or transform branch. The input file will have certain restrictions; the keyword 'module' must be immediately followed by a colon. The keywords 'inputs' and 'outputs' shall immediately be followed by a colon and a 'newline' character. The name of the module, the inputs and particular the information must represent a complete data flow diagram. If the data flows are not correct, branches may be generated which will not be tied to the transforms.

2.3 User interaction

The initial input file does not have to contain the transform analysis information (a, t, e). Therefore, the system shall provide the capability of allowing the user to interactively add the transform information to each individual module. In order to obtain the transform information the user shall partition the data flow diagram into the afferent, efferent and transform as outlined in Chapter 1. The module will be displayed and the user will be prompted to enter the proper transform information (a, e or t) for the displayed module (see Appx. B). If any module in the input file does not contain the transform information, the system will not generate any output.

After this information has been entered and on subsequent runs, the user shall have the capability of interactively changing the transform information for any module. This will

be done by sequentially accessing each module and prompting the user for the new transform information. On subsequent runs for the same file, the user will be prompted to add transform information only if a module does not contain it. A description of how transform analysis is applied to a system is included in Chapter 1.

2.4 Output

The project will produce two types of outputs showing the hierarchical software structure. The types will be textual and tabular.

The first type of output will be textual. It will describe the characteristics and interconnections of each individual module. This output will contain the module name, the module's input(s), output(s), superordinate(s) and subordinate(s). The individual modules will be grouped by afferent, transform and efferent (see Appx. C).

The second type of output will be a tabular text. This output will list the names of the modules in an hierarchical format. There will be a main module which will have the name of the system (input file), a main module for each of the afferent, efferent and transform branches. A list of all the dependent modules for each branch will be listed below each main module (see Appx. D).

With the above two outputs, a user will have both an hierarchical representation of the system and a description of each individual module that makes up the system. This information will be useful in breaking down the modules into submodules as the design is refined.

2.5 General characteristics

The software for this project will handle systems with up to 150 modules. There is a maximum of 50 modules for each of the branches the afferent, efferent and transform. However, the number of modules per branch can be altered by changing parameters in the source code.

The system is modular in design. Each branch is processed by different modules. This eliminates having to change all branches if an error or a change occurs in one of the branches.

Chapter 3 - Design

3.1 Introduction

The system was developed using the "C" programming language and the Shell programming language under a UNIX operating system at Kansas State University.

Figure 3.1 shows the hierarchical software structure of the system. The system is modular in design in that the processing of the data is done by functions. Some of the functions have subfunctions that perform redundant tasks. The system can be divided into seven main sections; Process input, Load input, Process transform, Process afferent, Process efferent, Print each module, Print tabular report. The system is executed by invoking the shell 'TRANSFORM'.

3.2 Process input

The process input part of the system, Fig. 3.2, is responsible for the processing of the input into a data structure that can be easily manipulated for later processing.

The input file will be obtained either automatically from a system that generates a data flow diagram or it will be build through an editor. The information contained in the input file will be derived from a data flow diagram.

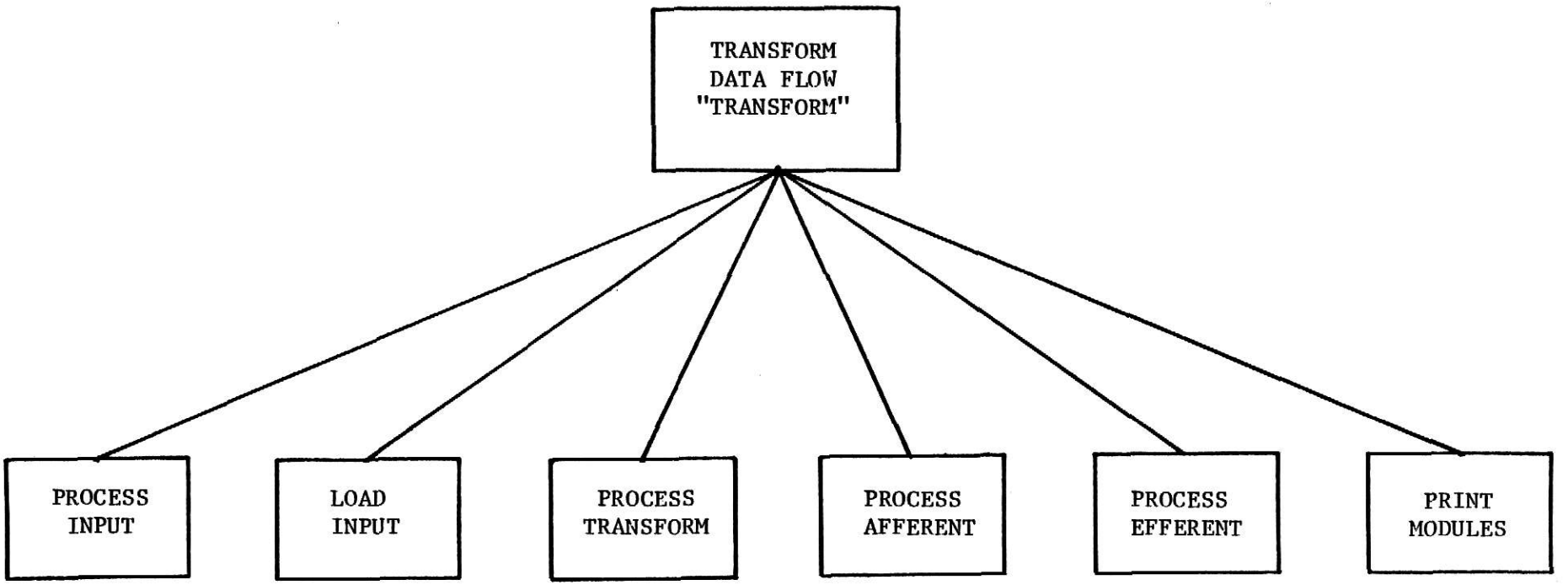

Figure 3.1 - HIERARCHICAL DIAGRAM OF TRANSFORM SYSTEM

The format of the input file has been designed for easy use and understanding by the user (see Appx. A). The file contains a module name followed by its input(s) and output(s).

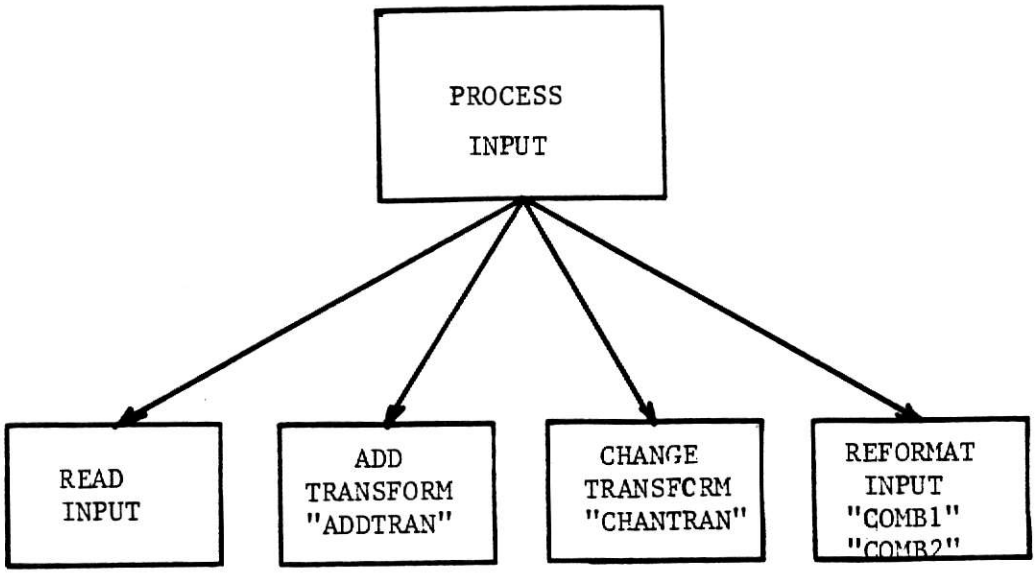

Figure 3.2 -- PROCESS INPUT SUBSYSTEM

The input data is derived from the data flow diagram and it may not contain the transform information: afferent, transform or efferent (a, e or t). Module ADDTRAN is responsible for prompting the user for the transform information. The user will enter either an 'a', 'e' or 't'. If this information has been added by another means (i.e. an editor), this module will not prompt the user for the information. The user will be prompted only to enter information for the modules that do not contain it. The transform information will be inserted in the

record next to the module name with a space separating the two (see Appx. B). A flag will be set if any of the modules are without their transform information stopping the system from executing further.

When all the transform information has been entered, the user has the option to change this information. CHANTRAN is a module that allows the user to change this information. The user is prompted if changes are required. This module reads the input file and displays each record containing a module name to the user. The user can either enter a, e or t if he wants to change the information or carriage return if no changes are required. The process ends when all modules have been displayed.

3.3 Reformating the input file

The input file is formated to be easy for the user to initiate, to read and to change. However, this format is not the best to use for processing. Two shell programs are used to convert the original input file to a format that is easier to process.

The COMB1 shell combines the module name with each of its input and outputs. These records are then sorted by module name and record number. Shell COMB2 formats these records in a final form by combining an input and an output with a module

name. If a module has more than one input and/or output, multiple records are generated for that module.

3.4 Load input

The process LOAD_INPUT fig. 3.3 is responsible for reading the reformated input file and loading it into its proper array according to the transform information specified on each record. The main function reads a string until a 'newline' character is encountered. It then determines if the record is part of the afferent, transform or efferent branch. Functions LOAD_A, LOAD_E and LOAD_T perform the actual loading. There is an array for each transform type, and it is assumed that all the records have the transform information.

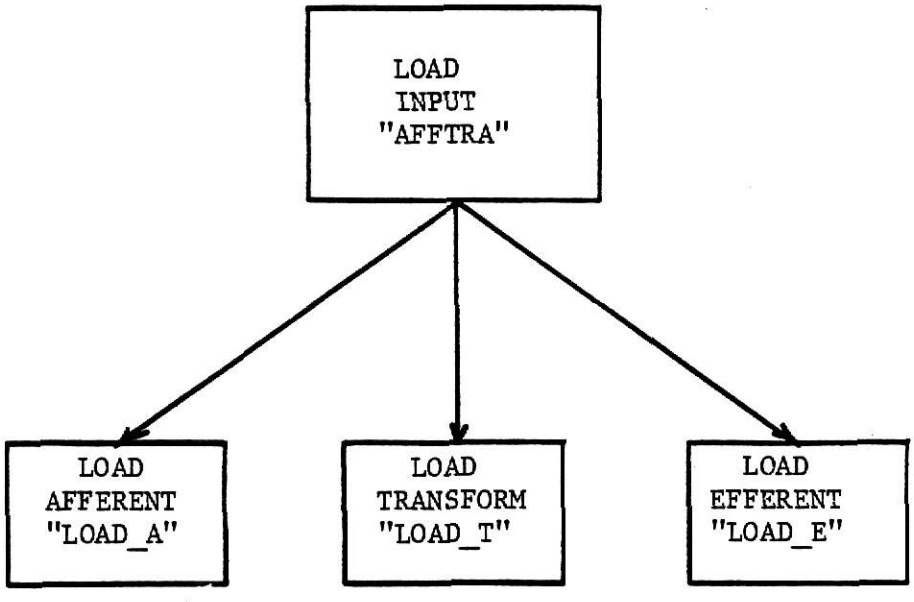

Figure 3.3 -- LOAD INPUT SUBSYSTEM

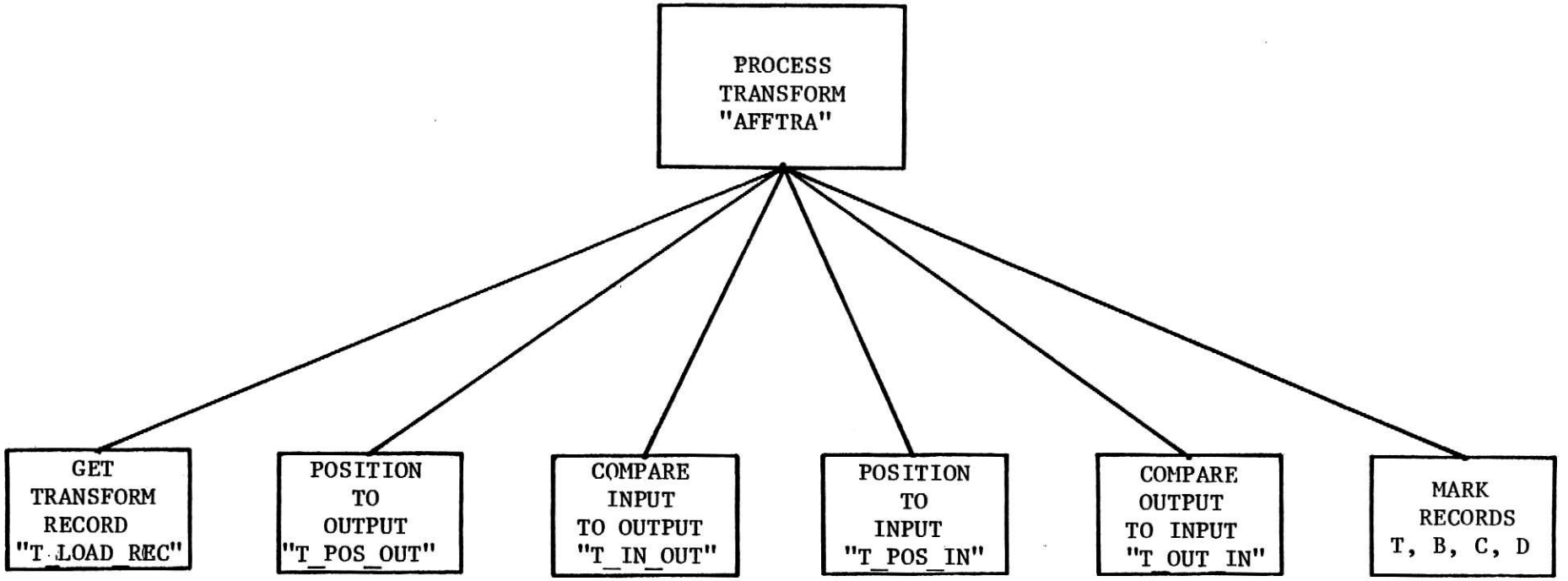

Figure 3.4 - PROCESS TRANSFORM SUBSYSTEM

3.5 Process transform

Figure 3.4 shows the hierarchical structure of the branch that processes the transform type records. The transform records determine the start of the afferent and efferent branches. Therefore, these records have to be processed first. T_LOAD_REC obtains an element of the transform array and it loads the module name, input and output into hold areas. The T_POS_OUT function accesses the next record and positions a pointer to the beginning of the output field. Function T_IN_OUT compares this output to the input in the hold area. If there is a match, it means that this output and the input do not connect to the afferent or efferent side of the structure. If the two do not match the next record is obtained , the same comparison is done until there is a match between two records or the list of records is exhausted. Functions T_POS_IN and T_OUT_IN perform the same task in comparing the output in the hold area to the input of the remaining transform records.

As the records are matched they are also identified by placing a character in the zero occurrence of the array. The identifiers are as follows: a) a 't' signifies that the record has no ties outside the transform branch; b) a 'd' if the input ties to the transform branch and the output to the the input of the efferent branch; c) a 'c' if the input ties to

the afferent branch and the output to the transform branch and d) a 'b' if the input ties to the afferent and the output to the efferent branch. These identifications are important for processing the afferents and the efferents.

3.6 Process afferent and efferent

The processing of the afferent fig. 3.5 and efferent fig. 3.6 is similar; therefore, they are discussed together.

In processing the afferent branch, a module from the transform branch identified with a 'b' or 'c' must be obtained. These are the module that get their input from the afferent branch. Once a module has been found through the function GET_TRANS, its input is loaded into a holding area. The function FIND_1_MOD finds the module in the afferent that is connected to the transform module via its output. When found the location of the input array where the transform module is stored is stored in a different array. This array will also store the location of the superordinate and the subordinate of each module. The module's superordinate and subordinate are found by matching the output of the module with the input of other modules for the superordinate and by matching the input of the module with the output of other modules to find the subordinate. The functions to do this are F_A_SUP and F_A_SUB with other functions that position and match the two fields.

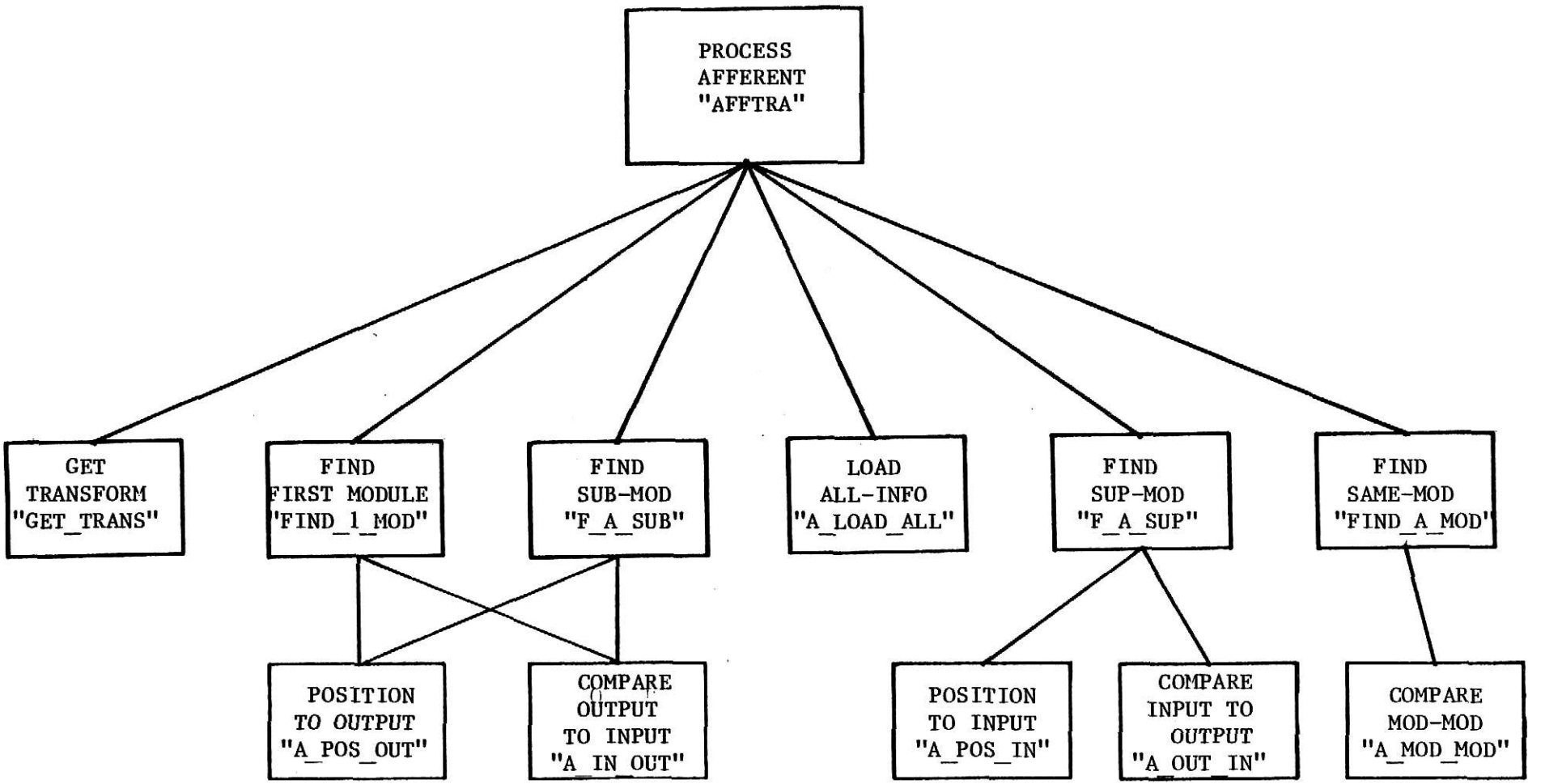

Figure 3.5 - PROCESS AFFERENT SUBSYSTEM

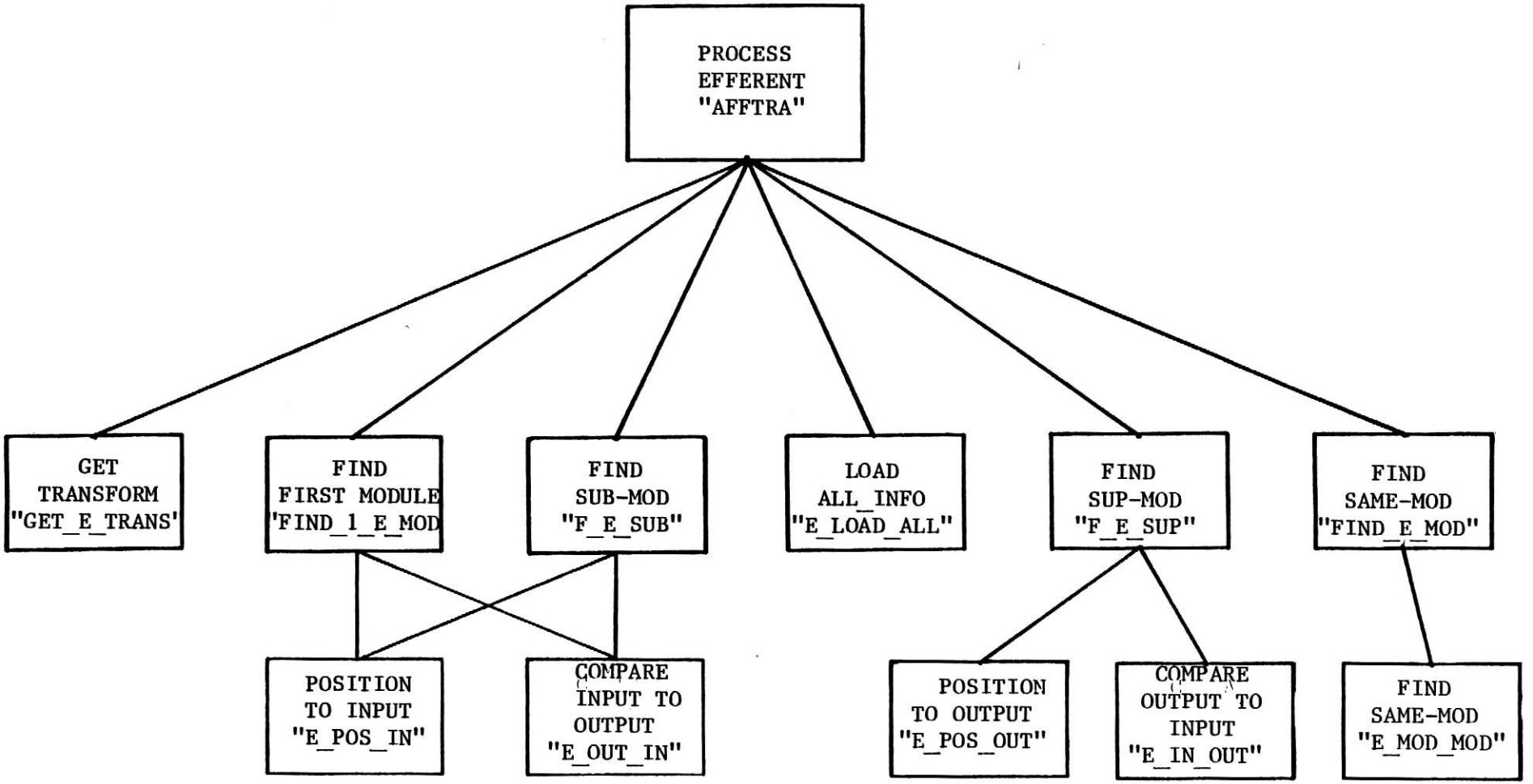

Figure 3.6 - PROCESS EFFERENT SUBSYSTEM

A search is made of all the remaining afferent modules to locate any other records that contain the same module name. It is possible for a module to have more than one input record. This occurs when a module has multiple inputs and/or outputs. The location of these modules is also stored and it will be used when the output is generated.

The process_afferent routine is repeated until all the modules in the afferent branch have been processed. As stated before, the routine to process the efferent branch records is similar to the afferent routine. It must be kept in mind that where the input is used in the afferent branch the output must be used in the efferent branch, and where the output is used in the afferent branch the input must be used in the efferent branch.

3.7 Print afferent, efferent and transform

The system produces two types of outputs: a) a textual output (Appx. C) of each individual module listed by afferent, transform and efferent, and b) a tabular output (Appx. D) listing in hierarchical order the modules' names under the headings of afferent, transform and efferent.

As in the previous processing, the modular output for the afferent and efferent modules is similar; therefore, only the afferent will be described. Figure 3.7 shows the hierarchical

structure of this branch.

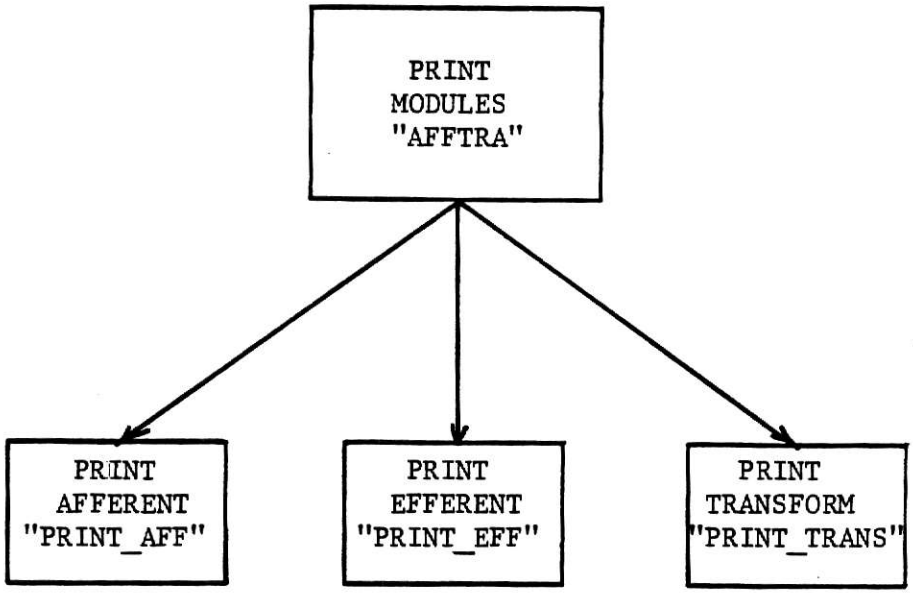

Figure 3.7 -- PRINT MODULES SUBSYSTEM

An array has been populated with the location of all afferent modules in the previous section. This information will be used to print each individual module with its input(s), output(s), superordinate(s) and subordinate(s). A location of a module is obtained from the above array. The module name, its input, output, superordinate and subordinate is loaded into holding areas by the function A_P_LOAD. Function A_P_MATCH accesses the module name in the next location and the module name of this module is compared with the module name in the hold area. If there is a match the module's other information the input, output, superordinate and subordinate

are matched. The information that matches is disregarded but the information that does not match is saved in hold areas. When there is a change in module name, the module and the associated information is printed. This process is repeated until all modules have been processed. Again, this procedure also applies to the printing of the modules and their information in the efferent branch.

The transform branch is processed and printed in the same fashion as the afferent and efferent; therefore, it will not be repeated.

3.8 Produce tabular output

The tabular output is processed by the function PRINT_ORDER. This function accesses the array where the locations of the afferent and efferent are stored, the array where the transform modules are stored, and sequentially print the module name under the heading of AFFERENT, TRANSFORM and EFFERENT.

3.9 Conclusion

In designing the system, attention has been given in designing a system that is easy to maintain and to modify. Functions and subfunctions were coded to perform redundant tasks. Each

branch of the transform has been designed to use separate functions. Some of these functions may appear to perform the same tasks and they do; this was done to keep the branches separate. If it becomes necessary to change one of the branches, the other two branches will not have to be changed. This design approach has resulted in having to write more code but I believe that the payoff will be in ease of maintaining the system.

Chapter 4 - Implementation

4.1 General implementation

The system developed in this project is composed of three (3) "C" language programs ADDTRAN.C, CHANTRAN.C, AFFTRA.C and three Shell programs TRANSFORM, COMB1 and COMB2. The system was developed and implemented on a computer using a UNIX operating system.

In designing the software for the system, a lot of time and effort was spent in making the software modular and easy to use. As discussed in various design methodologies especially the Yourdon-Constantine methodology, modular designed systems are easier to test and to maintain.

The main "C" program for the system (AFFTRA.C) is divided into three main parts each to process the afferent, efferent and transform. Some functions perform similar processing in each part. This was done so that if any of the parts change, only that portion of the software has to be changed.

The system is easy to execute, the user simply invokes the Shell program "TRANSFORM". The system leads the user through a series of questions and answers to see if transform information on the input file has to be added or changed. (See Appx. for complete user instructions).

Even though a lot of time and effort was spent in making the system flexible and easy to use, the system does have some limitations. The system expects that the input file be a correct textual representation of the data flow diagram. The system prompts the user if transform information is missing. However, it will not check the information for correctness.

The system expects a file that is 'chain-like' were the output of a module is the input to another module. If the input file does not represent a correct data flow diagram, the results are unpredictable.

The system can handle input from data flow diagrams that have a maximum of fifty (50) modules in each branch the afferent, efferent and transform. If there are systems with more modules than the above, the system can be easily modified to handle them. The names of the modules, the inputs and outputs is limited to twenty (20) characters each. Each module can have a maximum of five inputs and five outputs. The input file must be arranged in a specified format (see Appx. A). One of the system outputs is a textual representation of the software structure of the system. The structure represents a 'first' cut representation of the Yourdon-Constantine methodology. From this tabular output a final software structure can be manually derived.

4.2 Testing

The system was tested using input information taken from data flow diagrams of various sizes. The system was not tested with an input file which contained more than fifty (50) modules per branch. In such a case, it can be predicted that the system will fail on the basis that the arrays that hold the module names, inputs and outputs will overflow.

The size of the systems tested ranged from systems with 10 modules to systems with a total of 25 modules. Under the above circumstances, the system responded as expected with respect to the execution time and the output produced. Execution time varied from 2 to 5 seconds again depending on the size of the system and the number of users.

As previously mentioned the system expects an input file that is a true representation of the data flow diagram. The system only requires that each module contain whether the module belongs to the afferent, efferent or transform branch of the data flow diagram.

Chapter 5 - Conclusions

5.1 Conclusions

The initial requirements of this project were to develop a system that would take data from a data flow diagram and by using the Yourdon-Constantine design methodology produce a 'first' cut textual and tabular representation of the software structure of a system. The requirements have been met.

This project was one of a group of projects done at Kansas State University on mechanized software development tools. If the age of the Fifth Generation of computing is to occur, mechanized software development tools must be developed to obtain full use of the advanced hardware.

In researching this project, the major system design methodologies were reviewed. The methodologies included the Jackson's, the Warnier-Orr's and of course the Yourdon-Constantine's methodology. One thing that became clear is that no one methodology solved all system design problems. While both the Jackson's and the Warnier-Orr's are based on the concept that the structure of the system should be determined by the structure of the data, the Yourdon-Constantine methodology is based on the concept that the structure of a system should be determined by the flow of data through the system.

There are systems where the Jackson's and Warnier-Orr's methodologies work better and some where the Yourdon-Constantine's works better. With the advent of faster and less expensive computers, it may come a day when all of the methodologies will be mechanized and systems may be designed by using the best characteristics of all of the above methodologies.

Design methodologies may not be ideal for every type of system; but when properly applied, they result in systems that are easier to test and to maintain. However, because they are hard to learn and use these methodologies are often improperly used or not used at all. It is through mechanization that these methodologies will become easier to learn and to use.

5.2 Extensions

There are a few possible extension which would make this system easier to use.

The input for the system could be automatically generated by a system that takes the ERA specification and converts it into a data flow diagram. This would relieve the user of having to generate the input file which introduces the possibility of errors in describing the inputs and outputs for a module.

One of the outputs produced by the system is a tabular representation of the 'transformed' system showing the

afferent, efferent and transform modules. An extension to this project could be to take this tabular output and produce a graphical representation of the software structure.

The software structure produced by the system is a 'first' cut representation of the software structure. Each of the modules generated by the system can be 'exploded' into submodules until a final software structure is obtained.

Another extension would be to populate a data dictionary with the name of the modules and their inputs and outputs. This would ensure that a standard naming convention would be used for the modules, inputs and outputs.

These are some of the extensions which I feel would enhance the system, make it more useful and easier to use.

References

1. Stevens, W., G. Myers and L. Constantine
"Structured Design"
IBM System Journal, vol. 13 No. 2
1974, pp. 115-139

2. Yen R.
"Software Engineering"
Spectrum vol. 20 no. 11, pp. 91-94
IEEE, Nov. 1983

3. Pressman, R. S.
"Software Engineering:A Practitioners Approach"
McGraw-Hill, New York, N.Y., 1982

4. De Marco, T.
"Structured Analysis and System Specification"
Prentice-Hall, Englewood Cliffs, N.J., 1979

5. Yourdon E., Constantine L.L.
"Structured Design"
Prentice-Hall, Englewood Cliffs, N.J., 1979

6. Weinberg, V.
"Structured Analysis"
Yourdon Inc., New York, N.Y., 1978

7. Structured Analysis and Design
Infotech International Ltd.
Maidenhead, Berkshire, England
1978

8. Page-Jones, M.
"The Practical Guide to Structured System Design Methodologies"
Yourdon Press, New York, N.Y., 1980

9. Jensen, R.W; Tonie, C.C.
"Software Engineering"
Prentice-Hall, Englewood Cliffs, N.J., 1979

10. Peters L. J., Tripp, L. L.
"Comparing Software Design Methodologies"
Datamation, Vol. 23, No. 11
1977, pp. 89-94

11. Bergland, G. D.
"Structured Design Methodologies"
15th Annual Design Automation Conference
Proceedings; June 1978, pp. 475-493
IEEE, 1979

12. Bergland, G.D., Gordon, R.D.
"Software Design Strategies"
2nd edition
IEEE, 1981

Appendix A -- Input Record

```
module: sequence
inputs:
update_card
outputs:
seq_cards
module: edit
inputs:
seq_cards
outputs:
valid_card
module: reformat
inputs:
valid_card
outputs:
form_update
module: validate
inputs:
old_master
outputs:
ok_master
module: expand
inputs:
ok_master
outputs:
master_area
module: match
inputs:
master_area
form_update
outputs:
matched_set
unmatched_mast
module: update
inputs:
matched_set
outputs:
new_master_rec
module: format_output
inputs:
new_master_rec
unmatched_mast
outputs:
form_mast
module: add_check
inputs:
form_mast
outputs:
new_master
```

Appendix B -- Input Record with transforms

```
module: sequence a
inputs:
update_card
outputs:
seq_cards
module: edit a
inputs:
seq_cards
outputs:
valid_card
module: reformat a
inputs:
valid_card
outputs:
form_update
module: validate a
inputs:
old_master
outputs:
ok_master
module: expand a
inputs:
ok_master
outputs:
master_area
module: match t
inputs:
master_area
form_update
outputs:
matched_set
unmatched_mast
module: update t
inputs:
matched_set
outputs:
new_master_rec
module: format_output t
inputs:
new_master_rec
unmatched_mast
outputs:
form_mast
module: add_check e
inputs:
form_mast
outputs:
new_master
```

Appendix C -- Module textual output

```
     ** AFFERENT MODULES **

     module name: expand
        input(s): ok_master
       output(s): master_area
superordinate(s): main
  subordinate(s): validate

     module name: validate
        input(s): old_master
       output(s): ok_master
superordinate(s): expand
  subordinate(s):

     module name: reformat
        input(s): valid_card
       output(s): form_update
superordinate(s): main
  subordinate(s): edit

     module name: edit
        input(s): seq_cards
       output(s): valid_card
superordinate(s): reformat
  subordinate(s): sequence

     module name: sequence
        input(s): update_card
       output(s): seq_cards
superordinate(s): edit
  subordinate(s):


     ** EFFERENT MODULES **

     module name: add_check
        input(s): form_mast
       output(s): new_master
superordinate(s): main
  subordinate(s):
```

```
** TRANSFORM MODULES **

module name: format_output
   input(s): new_master_rec
             unmatched_mast
  output(s): form_mast

module name: match
   input(s): master_area
             form_update
  output(s): matched_set
             unmatched_mast

module name: update
   input(s): matched_set
  output(s): new_master_rec
```

Appendix D -- Module tabular output

CHECKSEQ

AFFERENT	TRANSFORM	EFFERENT
AFF	format_output	EFF
expand	match	add_check
validate	update	
AFF		
reformat		
edit		
sequence		

Appendix E -- User's guide

'USER'S GUIDE'

Before the system can be invoked, the user must build an input file containing the input information (see Appx. A). To invoke the system, the user shall enter: 'transform', this will invoke the main shell program.

The system will prompt the user: 'Please enter the name of the system that you want to transform:'. The user enters the name of the input file. If the file does not exist, the system will display: 'invalid system (file name) please re-enter'.

When a valid system name is entered, the program 'ADDTRAN' is executed and if any of the modules in the input file do not contain transform information, the user is prompted: 'enter transform information' At this time the user must enter 'a', 'e', or 't' depending on whether the module is part of the afferent, efferent or transform.

If all the modules in the input file contain transform information the user is prompted: 'do you want to change any transform information? (y or n):'. If the user enters 'y', the system will step through all the modules in the input file and the transform information for any or all modules may be

changed. To skip a module press 'carriage return'.

After the transform information has been changed, the input file is processed by the program 'AFFTRA'. This is the main program which reads the input file and produces the system's outputs consisting of the individual modular output and the tabular output format. (See Appx. C & D). The user is then prompted: 'do you want the output printed on the printer?: (y or n)'. If the user answers 'y' the output is printed on the system's printer otherwise it is printed on the user's terminal.

Appendix F -- BNF syntax

"Input syntax"

<input_module> ::= <in_mod_keyword> <mod_name>

<input_fixed> ::= <inp_keyword>

<input_text> ::= {<inp_description>}

<output_fixed> ::= <output_keyword>

<output_text> ::= {<out_description>}

<mod_keyword> ::= 'module: '

<mod_name> ::= {<alpha> <numerics>}

<inp_keyword> ::= 'inputs: '

<inp_description> ::= {<alpha> <numerics>}

<out_keyword> ::= 'outputs: '

<out_description> ::= {<alpha> <numerics>}

<alpha> ::= a |...| Z

<numerics> ::= 0 |...| 9

"Modular output syntax"

<transf_type> ::= <type_keyword>

<out_mod> ::= <out_mod_keyword>

<out_mod_name> ::= <mod_name>

<out_input_fixed> ::= <out_inp_keywoxDlsrd> <input_description>

<out_input_name> ::= {<input_description>}

<out_output_fixed> ::= <out_out_keyword> <output_description>

```
<out_output_name> ::= {<output_description>}

<sup_fixed> ::= <sup_keyword> <sup_description>

<sup_name> ::= {<sup_description>}

<sub_fixed> ::= <sub_keyword> <sub_de scription>

<sub_name> ::= {<sub_description>}

<type_keyword> ::= '** AFFERENT **' | '** EFFERENT **' |
                   '** TRANSFORM **'

<out_mod_keyword> ::= 'module name: '

<mod_name> ::= {<alpha> <numerics>}

<out_input_keyword> ::= 'input(s): '

<input_description> ::= {<alpha> <numerics>}

<out_out_keyword> ::= 'output(s): '

<output_description> ::= {<alpha> <numerics>}

<sup_keyword> ::= 'superordinate(s): '

<sup_description> ::= {<alpha> <numerics>}

<sub_keyword> ::= 'subordinate(s): '

<sub_description> ::= {<alpha> <numerics>}

<alpha> ::= a |...| Z ls(_ <numerics ::= 0 |...| 9
```

"Tabular output syntax"

```
<header> ::= <sys_name>

<sub_header> ::= <branch_name>

<sub_aff> ::= {<aff_sub_branch>}

<modules> ::= {<module_names>}

<sys_name> ::= <cap_letters>

<branch_name> 'AFFERENT' |
                  'TRANSFORM' |
```

'EFFERENT'

<aff_sub_branch> ::= 'AFF'

<eff_sub_branch> ::= 'EFF'

<module_names> ::= {<alpha> <numerics>}

<alpha> ::= a |...| Z

<numerics> ::= 0 |>>>| 9

Appendix G -- Program Specification

Program name: transform

"Transform" is the main shell program, it drives the entire system. The user executes the system by invoking the shell "transform". "Transform" prompts the user for the name of the system to be processed and verifies that the file exists. If the file does not exist, an error message is issued and the user is asked to reenter the name of the system to be processed.

"Transform" invokes the other programs, "addtran.out", "chantran.out", "comb1", "comb2" and "afftra.out" (see program specification for more information on the above programs). After the execution of "afftra.out", "transform" prompts the user if the system outputs are to be printed on the local printer or at the user"s terminal.

Program name: addtran.out

Input file: dataflow

Output file: allin.out

"Addtran.out" is the compiled version of program "addtran.c". The "addtran.out" program is used to check if all of the modules in the input file have transform information "a", "e" or "t". The program looks for the keyword "module:" and when it finds it the module is checked for the transform information "a", "e" or "t". If the transform information is found, the program continues to scan the input file. If the transform information is not present, then the program prompts the user to enter the transform information. If all of the modules contain the transform information, no messages are issued by the program.

Program name: chantran.out

Input file: dataflow

Output file: allin.out

Chantran.out is the compiled version of chantran.c. The chantran.out program is used to change the transform information associated with an input module. When the user indicates the transform information associated with a module is to be changed, chantran.out reads the input file and searches for the keyword "module:". When the keyword "module:" is found, the module name with its transform information is displayed and the user is prompted to enter the new transform information or a "carriage return". The program validates the entered information. If a "a", "e", "t" or "carriage return" have been entered, the program proceeds to the next module name. The above is repeated until all of the modules have been processed. If the wrong information is entered, the program issues an error message and prompts the user to reenter the information.

Program name: afftra.out

Input file: allin.in

Ouptut file: output

Afftra.out is the compiled version of the program afftra.c. Afftra.out is the main program of the system. The first part of afftra.out reads the input file and loads each module with its input(s) and output(s) into separate arrays. There is one array for each branch of the transform the afferent, the efferent and the transform branch.

After all modules have been loaded starting with the transform, each branch is processed. The transform branch is processed first since both the afferent and the efferent branches start at the transform branch. Once the transform is processed and the start of the afferent and the efferent has been determined, the afferent branch is processed next.

Each branch is processed in a "chainlike" fashion since the output of a module is the input to another module. This process is repeated for the efferent branch. When all modules have been processed, the program produces two types of outputs: 1) the listing of each module by branch with its input(s), output(s), sub-ordinate and sup-ordinate module(s) and (2) a tabular output which lists under the heading of "AFFERENT" "TRANSFORM" and "EFFERENT" all of the modules that make up the "transformed" system.

Appendix H — SHELL listings

"TRANSFORM"

```
this is the main driver of the
system to transform data flow to
a software structure

echo "Please enter the name of the system that
      you want to transform:"
read sys

caps='echo $sys | tr '[a-z]' '[A-Z]''
until test -r $sys
 do
  echo "invalid system $sys please reenter:"
  read sys
 done

cp $sys dataflow

addtran.out

echo " "
echo " "
echo "do you want to change any transform
information? (y or n):"
read ans

if [ "$ans" = y ]
then
cp allin.out dataflow
chantran.out
fi

cp allin.out $sys
cp allin.out dataflow
rm out1 out2 out3

comb1.in

sort out1 > out2
 comb2.in
cp out3 allin.in

afftra.out $caps > output
```

```
 echo "do you want the output printed on the printer ?  (y  or
n)
 (else it will be printed on the terminal):"
 read print

 if [ "$print" = y ]
 then
  lpr output
 else
  cat output
 fi
 THE END
```

"COMB1.IN"

```
 awk -s '
   BEGIN { mod=" ";
           inp=" ";
           out=" "}
         $3 == "inputs:" {inp=$4}
         $3 == "outputs:" {out=$4}
         $1 == mod {print mod "," inp "," out}
         $1 != mod {mod=$1}' out2 >> out3
```

"COMB2.IN"

```
 awk -s '
    BEGIN {mod='z' ;
           number=0 ;
           type='x'}
   /module:/ {mod=$2 ;
              tran=$3 ;
              number=0}
   /inputs:/ {number=0 ;
              type="inputs:"}
   /outputs:/ {number=0 ;
              type="outputs:"}
             {if (number>=1 && type=="outputs:") {print mod " "
number " " type " "  $1 "," tran  }
 else
              {if (number>=1 && type=="inputs:") {print mod " "
number " " type " "  $1}};
              number = number + 1}
    ' dataflow  >> out1
```

Appendix I — ADDTRAN listing

```
1   /* PROGRAM ADDTRAN.C */
2   #INCLUDE <STDIO.H>
3   CHAR STRING2[50];
4   MAIN()
5   {
6       INT I,N;
7       FILE *RPTR,*WRPTR;
8       RPTR=FOPEN("DATAFLOW","R");
9       WRPTR=FOPEN("ALLIN.OUT","W");
10  WHILE (1)
11  {

12   I = 0;
13   DO
14    STRING2 [++I] = '\0';
15    WHILE (I < 50);

16    FGETS (STRING2,50,RPTR);
17   IF (FEOF(RPTR))
18    BREAK;
19   IF (STRNCMP(STRING2,"MODULE:",7) != 0)
20    FPUTS (STRING2,WRPTR);
21   ELSE
22  {
23   IF (STRING2 [STRLEN(STRING2) - 3] == ' ')
24    FPUTS (STRING2,WRPTR);
25   ELSE
26   {
27    GET_TRANS();
28    FPUTS (STRING2, WRPTR);
29  }
30  }
31  }
32  }

33  /*THIS FUNCTION ADD THE TRANSFORM INFORMATION TO THE MODULE NAME
34  BY PROMPTING THE USER FOR IT */

35  GET_TRANS()
36  {

37  INT I;
38  CHAR C, D;
39  C = ' ';
40  D = ' ';
41  PRINTF ("\N");
42  PRINTF (" THE FOLLOWING MODULE DOES NOT CONTAIN TRANSFORM INFO.");
43  PRINTF ("\N\N");
44  PRINTF ("%S\N", STRING2);
45   PRINTF ("PLEASE ENTER AFFERENT, EFFERENT OR TRANSFORM (A, E OR T):");

46  ENTER:

47   C = GETCHAR();
48   IF (C == '\N')
49    RETURN;
50   D = GETCHAR();
51
52   IF ((C == 'A') || (C == 'T') || (C == 'E'))
53   {
```

```
54        I = STRLEN(STRING2);
55        STRING2 [I - 1] = ' ';
56        STRING2 [I] = C;
57        STRING2 [++I] = '\N';
58       }
59       ELSE
60       {
61        PRINTF ("PLEASE ENTER A, E OR T:");
62        GOTO ENTER;
63       }

64      }
```

Appendix J -- CHANTRAN listing

```
1    /* PROGRAM NAME : CHANTRAN.C */
2    #INCLUDE <STDIO.H>
3    CHAR STRING2[50];
4    MAIN()
5    {
6        INT I,N;
7        CHAR C, D;
8        FILE *RPTR,*WRPTR;
9        RPTR=FOPEN("DATAFLOW","R");
10       WRPTR=FOPEN("ALLIN.OUT","W");

11   PRINTF ("%S\N", "THE SYSTEM WILL STEP THROUGH ALL THE MODULES.");
12   PRINTF ("%S\N", "IF YOU WANT TO CHANGE THE TRANSFORM INFORMATION,");
13   PRINTF ("%S\N", "WHEN THE MODULE IS DISPLAYED ENTER A, E , OR T");
14   PRINTF ("%S\N", "CARR. RETURN ACCORDINGLY.");
15   WHILE (1)
16   {

17    I = 0;
18    DO
19     STRING2 [++I] = '\0';
20     WHILE (I < 50);

21     FGETS (STRING2,50,RPTR);
22    IF (FEOF(RPTR))
23     BREAK;
24    IF (STRNCMP(STRING2,"MODULE:",7) != 0)
25     FPUTS (STRING2,WRPTR);
26    ELSE
27   {
28    PRINTF ("\N\N");
29    PRINTF ("%S\N", STRING2);
30    PRINTF ("PLEASE ENTER NEW INFO. A, E, T OR CARR. RETURN :");

31    C = GETCHAR();
32    IF (C == '\N')
33    {
34     FPUTS (STRING2,WRPTR);
35     CONTINUE;
36    }
37    D = GETCHAR();
38    GET_TRANS(C);
39    FPUTS (STRING2, WRPTR);
40   }
41   }
42   }

43   /*THIS FUNCTION ADD THE TRANSFORM INFORMATION TO THE MODULE NAME
44   BY PROMPTING THE USER FOR IT */

45   GET_TRANS(D)
46   CHAR D;
47   {

48   INT I;
49   CHAR C;
50   C = D;

51   ENTER:
```

```
52
53      IF ((C == 'A') || (C == 'T') || (C == 'E'))
54      {
55       IF (STRING2 [STRLEN(STRING2) - 3] == ' ')
56        I = STRLEN(STRING2) - 2;
57       ELSE
58       I = STRLEN(STRING2);
59       STRING2 [I - 1] = ' ';
60       STRING2 [I] = C;
61       STRING2 [++I] = '\N';
62      }
63      ELSE
64      {
65       PRINTF ("PLEASE ENTER A, E OR T:");
66       C = GETCHAR();
67       IF (C == '\N')
68        RETURN;
69       D = GETCHAR();
70       GOTO ENTER;
71      }

72     }
```

Appendix K -- AFFTRA listing

```
/* PROGRAM NAME : AFFTRA.C */
#INCLUDE <STDIO.H>
    CHAR AARR[50][40];
    CHAR EARR[50][40];
    CHAR TARR[50][40];
    CHAR ARR[40];
CHAR H_MODULE [20];
CHAR H_INPUT [6][20];
CHAR H_OUTPUT [6][20];
CHAR H_SUPER [6][20];
CHAR H_SUB [6][20];
CHAR HOLD_MOD[40];
INT TAB_COUNT;
INT LIST_COUNT;

INT IN_COUNT, OUT_COUNT, SUP_COUNT, SUB_COUNT;
    INT MOD_ORD_TABLE [50][4];
    INT E_MOD_ORD_TABLE [50][4];
    INT A_COUNT, E_COUNT, T_COUNT;
    INT T_E_COUNT;
MAIN(ARGC,ARGV)
INT ARGC;
CHAR *ARGV[];
{
    FILE *RPTR,*WRPTR;
    INT I, N,X ;
    A_COUNT = 0;
    E_COUNT = 0;
    T_COUNT = 0;
    X=0;
    I=1;
    N=0;
    RPTR=FOPEN("ALLIN.IN","R");
    WRPTR=FOPEN("ARRAY.OUT","W");
WHILE (1)
{
    WHILE ((ARR[++N] = FGETC (RPTR)) != '\N')
    {
      IF (FEOF(RPTR))
        BREAK;
      CONTINUE;
    }
    IF (FEOF(RPTR))
      BREAK;
    IF (ARR[N - 1] == 'A')
      LOADA (N);
    ELSE IF (ARR[N-1] == 'E')
      LOADE (N);
    ELSE
      LOADT (N);
      N=0;
      I=I+1;
}

  TRANS_P ();

  AFF_P();

  PRINT_AFF();
```

```
57      EFF_P();

58      PRINT_EFF();
59      PRINT_TRANSF();

60     PRINTF ("\F");
61     FOR (I = 1; I < ARGC; I++)
62      PRINTF ("%48S%C", (ARGV[I]), (I < ARGC-1) ? ' ' : '\N');
63      PRINTF ("\N\N");

64      PRINT_ORDER();

65    }

66    /*************************************************************************/
67    /*************************************************************************/

68    /* LOAD FUNCTIONS */
69        LOADA (A)
70        INT A;
71    {
72        INT I;


73        A_COUNT = A_COUNT + 1;
74        I = 0;
75        WHILE (A >= ++I)
76        {
77          /* PUTCHAR ('A') */
78          AARR[A_COUNT][I] = ARR[I];
79        }
80    }
81        LOADE (A)
82        INT A;
83    {
84        INT I;


85        E_COUNT = E_COUNT + 1;
86        I = 0;
87        WHILE (A >= ++I)
88        {
89          /* PUTCHAR ('E') */
90          EARR[E_COUNT][I] = ARR[I];
91     /*   PRINTF ("%C", EARR[E_COUNT][I]); */
92        }
93    }
94        LOADT (A)
95        INT A;
96    {
97        INT I;

98        I=0;
99        T_COUNT=T_COUNT + 1;
100       WHILE (A > ++I)
101       {
102     /* PUTCHAR ('T'); */
103         TARR[T_COUNT][I] = ARR[I];
104       }
```

```
105     }

106     /*************************************************************/
107     /*************************************************************/

108     /*THIS IS THE MAIN TRANSFORM FUNCTION.
109     THE PURPOUSE OF THE FUNCTION AND ITS
110     SUBFUNCTIONS IS TO ACCESS THE TARR ARRAY
111     AND FIND THE START OF THE AFFERENT BRANCH(ES)
112     AND THE EFFERENT BRANCH(ES).*/

113     CHAR HOLD_IN[20],HOLD_OUT[20];
114     TRANS_P ()
115     {
116      INT MOD, F, G, I, K, N;


117      MOD = 0;
118      F = 0;
119      WHILE (++MOD <= T_COUNT)
120      {
121        IF (T_COUNT == 1)
122        {
123         TARR[1][0] = 'B';
124         BREAK;
125        }
126        T_LOAD_REC (MOD);
127        N = MOD;
128        F = 0;
129        K = 0;
130        /* PRINTF ("%S %D\N", "MOD1 = ", MOD); */
131
132        WHILE ((N != (MOD - 1)) && (F != 1))
133        {
134         /* PRINTF ("%S %D\N", "MOD2 = ", MOD); */
135         /* PRINTF ("%S %D\N", "N1 = ", N); */
136         IF (N == T_COUNT)
137          IF (MOD == 1)
138           BREAK;
139          ELSE
140           N = 0;
141         N = N + 1;
142         /* PRINTF ("%S %D\N", "N2 = ", N); */
143         K = T_POS_OUT (N);
144         F = T_IN_OUT (N,K);
145         /* PRINTF ("%S %D\N", "F = ", F); */
146        }

147         PUTCHAR ('N');
148        N = MOD;
149        G = 0;
150        WHILE ((N != (MOD - 1)) && (G != 1))
151        {
152         IF (N == T_COUNT)
153          {
154          IF (MOD == 1)
155           BREAK;
156          ELSE
157           N = 0;
```

```
158          }
159         N = N + 1;
160         I = T_POS_IN (N);
161         G = T_OUT_IN (N,I);
162        /* PRINTF ("%S %D\N", "G = ", G); */
163        }

164        IF (F == 1)
165        {
166         IF (G == 1)
167          /* I/O TRANSFORM */
168          TARR[MOD][O] = 'T';
169         ELSE
170          /* I-TRANSF, O-EFF. */
171          TARR[MOD][O] = 'D';
172        }
173        ELSE
174         IF (G == 1)
175           /* I-AFF., O-TRANSF */
176          TARR[MOD][O] = 'C';
177         ELSE
178          /* I-AFF., O-EFF. */
179          TARR[MOD][O] = 'B';

180     }

181    }

182    /*THIS FUNCTION POSITIONS POINTER TO THE
183    INPUT FIELD THEN LOAD INPUT AND OUTPUT INTO[
184    HOLD_IN AND HOLD_OUT ARRAYS  */

185    T_LOAD_REC (N)
186    INT N;
187    {
188     INT I, J, P;
189     CHAR C;
190

191     I = O;

192     J = O;
193     DO
194      C = TARR[N][++I];
195      WHILE (C != ','); /*POSITIONS I AFTER MOD. NAME*/
196
197     DO
198      HOLD_IN[++J] = TARR[N][++I];
199      WHILE (HOLD_IN[J] != ','); /*LOADS INPUT */

200      /* PRINTF ("%S\N", "HOLD_IN");
201      P = O;
202      DO
203       PRINTF ("%C", HOLD_IN[++P]);
204       WHILE (P <= J); */
205     J = O;
206     DO
207      HOLD_OUT[++J] = TARR[N][++I];
208      WHILE (HOLD_OUT[J] != ','); /* LOADS OUTPUT */
```

```
209       /* PRINTF ("%S\N", " HOLD_OUT ");
210       P = 0;
211       DO
212        PRINTF ("%C", HOLD_OUT[++P]);
213        WHILE (P <= J); */
214     }

215     /*THIS FUNCTION ACCESSES THE NEXT TRANSFORM
216     RECORD AND PLACES THE POINTER TO THE
217     BEGINNING OF THE OUTPUT FIELD */

218     T_POS_OUT (N)
219     INT N;

220     {
221      INT COMMA_COUNT, J, L;
222      CHAR C;

223      /* PRINTF ("%S\N", " ENTERED T_PO_OUT"); */
224      COMMA_COUNT = 0;
225      J = N;
226      L = 0;

227      WHILE (COMMA_COUNT < 2)
228      {
229       C = TARR[J][++L];
230       IF (C == ',')
231        COMMA_COUNT = COMMA_COUNT + 1;
232      }

233      RETURN (L);

234     }


235     /* FUNCTION TO MATCH INPUT TO ON
236     OF THE OUTPUTS.
237     POINTER IS SET AT START OF OUTPUT FROM
238     T_POS_OUT FUNCTION */

239     T_IN_OUT (N,L)
240     INT N,L;

241     {

242      INT I, J, C;

243      /* PRINTF ("%S\N", " ENTERED T_IN_OUT"); */

244      I = 0;
245      C = 0;
246      J = N;

247      WHILE (C == 0)
248      {
249       C = STRCMP (HOLD_IN[++I],TARR[J][++L]);
250       /* PRINTF ("%C %C\N", HOLD_IN[I], TARR[J][L]); */
251       IF ((HOLD_IN[I] == ',') && (TARR[J][L] == ','))
252        RETURN (1); /* MATCH FOUND */
253      }
```

```
254     RETURN (0); /* NO MATCH FOUND */
255    }

256    /* THIS FUNCTION ACCESSES THE NEXT
257    TRANSFORM RECORD AND PLACES THE
258    POINTER AT THE BEGINNING OF THE INPUT */

259    T_POS_IN (N)
260    INT N;
261    {

262     INT I, J;
263     CHAR C;

264     /* PRINTF ("%S\N", " ENTERED T_POS_IN"); */


265     I = 0;
266     J = N;

267     DO
268      (C = TARR[J][++I]);
269      WHILE (C != ',');

270     RETURN (I);

271    }


272    /* THIS FUNCTION LOOKS FOR A MATCH FOR THE OUTPUT
273    OF THE MODULE BEING REFERENCED WITN AN INPU
274    OF THE REST OF THE TRANSFORM MODULES */

275    T_OUT_IN (J, L)
276    INT J, L;

277    {

278     INT C, I;

279     /* PRINTF ("%S\N", " ENTERED T_OUT=IN"); */

280     I = 0;
281     C = 0;

282     WHILE (C == 0)
283     {
284      C = STRCMP(HOLD_OUT[++I],TARR[J][++L]);
285      /* PRINTF ("%C %C\N", HOLD_OUT[I], TARR[J][L]); */
286      IF ((HOLD_OUT[I] == ',') && (TARR[J][L] == ','))
287       RETURN (1); /*MATCH FOUND */
288      }

289     RETURN (0); /* NO MATCH FOUND */
290    }

291    /*************************************************************/
```

```
292.  /*************************************************************/

293   /* THIS FUNCTION AND ITS SUBORDINATE FUNCTIONS ARE THE
294   MAIN PROCESS OF THE AFFERENT BRANCH  */

295   INT ORD_COUNT, MOD_NBR;
296   INT SUB_COUNT, T_A_COUNT;
297   CHAR HOLD_MOD[40];

298   AFF_P ()

299   {
300    INT I,J;
301    INT R;
302    I = -1;
303    J = -1;
304    MOD_NBR = 0;
305    ORD_COUNT = 1;
306    T_A_COUNT = 0;
307    SUB_COUNT = 1;


308    WHILE (++J <= 50)
309    {
310     WHILE ( ++I <= 4)
311      MOD_ORD_TABLE [J][I] = 0;
312     I = -1;
313    }


314   WHILE (T_A_COUNT <= T_COUNT)
315   {
316    R = GET_TRANS (T_A_COUNT);
317    /* THIS MODULE GETS A TRANSF. INPUT */
318    IF (R == 0)
319     BREAK;

320    SUB_COUNT = ORD_COUNT;
321    MOD_ORD_TABLE [ORD_COUNT][1] = 99;
322    MOD_ORD_TABLE [ORD_COUNT][2] = 99;
323    MOD_ORD_TABLE [ORD_COUNT][3] = 99;

324    SUB_COUNT = SUB_COUNT + 1;
325    ORD_COUNT = ORD_COUNT + 1;



326     J = 0;
327     WHILE (++J <= A_COUNT)
328      AARR [J][0] = ' ';

329    FIND_1_MOD ();   /* FUNCTION TO FIND FIRST MODULE */
330
331    WHILE ((MOD_ORD_TABLE [ORD_COUNT][1] != 0) ||
332          (MOD_ORD_TABLE [SUB_COUNT][3] != 0))
333    {


334     IF (MOD_ORD_TABLE [ORD_COUNT][1] == 0)
335     {
```

```
336       MOD_ORD_TABLE [ORD_COUNT][1] = MOD_ORD_TABLE [SUB_COUNT][3];
337       SUB_COUNT = SUB_COUNT + 1;
338      }

339     /********* FOR TEST ONLY **********/
340     /*
341     IF (ORD_COUNT > 10)
342       BREAK;
343     */
344     /************** END ***************/

345      MOD_NBR = MOD_ORD_TABLE [ORD_COUNT][1];
346      IF (MOD_NBR == 0)
347       BREAK;

348      AARR [MOD_ORD_TABLE[ORD_COUNT][1]][0] = '*';
349      A_LOAD_ALL (MOD_ORD_TABLE [ORD_COUNT][1]);
350      F_A_SUBO (); /* THIS FUNCTION FINDS SUBORDINATE MODULE */

351      F_A_SUPE (); /* THIS FUNCTION FINDS SUPERORDINATE MODULE */
352      FIND_A_MOD ();
353     /* THE ABOVE MODULE FINDS ANY MODULES WITH THE SAME NAME */
354
355      ORD_COUNT = ORD_COUNT + 1;
356      }

357     IF (MOD_ORD_TABLE [ORD_COUNT][1] == 0)
358      SUB_COUNT = SUB_COUNT + 1;

359     }
360     }

361     /* THIS FUNCTIONS OBTAINS THE INPUT FROM ONE OF THE
362      TRANSFORM MODULES MARKED ¢B¢ OR ¢C¢ AND LOADS IT
363      INTO HOLD_IN */

364     GET_TRANS ()

365     {
366      CHAR C;
367      INT I, J;
368      I = 0;
369      J = 0;


370       WHILE (++T_A_COUNT <= T_COUNT)
371        {
372         IF ((TARR [T_A_COUNT][0] != 'B') &&
373             (TARR [T_A_COUNT][0] != 'C'))
374           CONTINUE;
375         ELSE
376         {
377          DO
378           (C = TARR[T_A_COUNT][++J]);
379           WHILE (C != ',');

380          DO
381           {
382           (HOLD_IN[++I] = TARR[T_A_COUNT][++J]);
383           /* PRINTF ("%C", HOLD_IN[I]); */
```

```
384           }
385           WHILE (HOLD_IN[I] != ',');
386          RETURN (1);
387         }
388       }
389      }

390     /* THIS MODULE FINDS THE FIRST AFFERENT MODULE THAT
391     IS ATTACHED TO THE TRANSFORM OR MAIN MODULE */

392     FIND_1_MOD ()

393     {
394      INT N,J,P;

395      N = MOD_NBR;

396      WHILE ((N != (MOD_NBR -1)) && (P != 1))
397      {
398       IF (N == A_COUNT)
399       {
400        IF (MOD_NBR == 1)
401         BREAK;
402        ELSE
403         N = 0;
404       }
405       N = N + 1;
406       J = A_POS_OUT (N);
407       P = A_IN_OUT (N,J);
408      }
409      IF (P == 1)
410      {
411       MOD_ORD_TABLE [ORD_COUNT][1] = N;
412       MOD_NBR = N;
413      }
414      ELSE
415       MOD_ORD_TABLE [ORD_COUNT][1] = 0;

416     }

417     /* THIS FUNCTION FINDS THE SUBMODULE TO THE
418     MODULE */
419     F_A_SUB0 ()

420     {
421      INT N,K,F;

422      N = MOD_NBR;

423
424      WHILE ((N != (MOD_NBR -1)) && (F != 1))
425      {
426       IF (N == A_COUNT)
427       {
428        IF (MOD_NBR == 1)
429         BREAK;
430        ELSE
431         N = 0;
432       }
433       N = N + 1;
```

```
434       K = A_POS_OUT (N);
435       F = A_IN_OUT (N,K);
436      }
437      IF (F == 1)
438       MOD_ORD_TABLE [ORD_COUNT][3] = N;
439      ELSE
440       MOD_ORD_TABLE [ORD_COUNT][3] = O;

441     }

442     /* THIS FUNCTION FINDS THE SUPERORDINATE OF A
443      MODULE */

444     F_A_SUPE ()

445     {
446      INT N,L,H;

447      N = MOD_NBR;
448      H = O;


449
450      WHILE ((N != (MOD_NBR -1)) && (H != 1))
451      {
452       IF (N == A_COUNT)
453       {
454        IF (MOD_NBR == 1)
455         BREAK;
456        ELSE
457         N = O;
458       }
459       N = N + 1;
460       L = A_POS_IN (N);
461       H = A_OUT_IN (N,L);
462      }
463      IF (H == 1)
464       MOD_ORD_TABLE [ORD_COUNT][2] = N;
465      ELSE
466       MOD_ORD_TABLE [ORD_COUNT][2] = 99;

467     }

468     /* THIS FUNCTION LOCATES ANY OTHER MODULE WITH THE
469      SAME NAME AS THE CURRENT ONE */

470     FIND_A_MOD ()

471     {
472      INT N,G;

473      N = MOD_NBR;
474
475      WHILE ((N != (MOD_NBR -1)) && (G != 1))
476      {
477       IF (N == A_COUNT)
478       {
479        IF (MOD_NBR == 1)
```

```
480         BREAK;
481        ELSE
482         N = 0;
483       }
484       N = N + 1;
485       G = A_MOD_MOD (N);
486      }
487      IF (G == 1)
488      {
489       MOD_ORD_TABLE [ORD_COUNT + 1][1] = N;
490      }

491     }

492     /* THIS FUNCTION LOADS MODULE NAME INPUT AND OUTPUT INTO
493      HOLD_MOD, HOLD_IN, HOLD_OUT */

494     A_LOAD_ALL(N)
495     INT N;
496     {
497     INT I, J;
498     CHAR C;
499     I = 0;
500     J = 0;


501     DO
502      HOLD_MOD[++J] = AARR[N][++I];
503      WHILE (HOLD_MOD[J] != ',');

504     J = 0;
505     DO
506      HOLD_IN[++J] = AARR[N][++I];
507      WHILE (HOLD_IN [J] != ',');

508     J = 0;
509     DO
510      HOLD_OUT[++J] = AARR[N][++I];
511      WHILE (HOLD_OUT[J] != ',');

512     }

513     /* THIS FUNCTION COMPARES TWO MODULES NAME TO
514      FIND OUT IF THEY MATCH */

515     A_MOD_MOD (N)
516     INT N;

517     {

518      INT I, L, J, C;

519      /* PRINTF ("%S\N", " ENTERED A_MOD_MOD"); */

520      I = 0;
521      L = 0;
522      C = 0;
523      J = N;

524      WHILE (C == 0)
```

```
525      {
526       C = STRCMP (HOLD_MOD[++I],AARR[J][++L]);
527       /* PRINTF ("%C %C\N", HOLD_MOD[I], AARR[J][L]); */
528       IF ((HOLD_MOD[I] == ',') && (AARR[J][L] == ','))
529       {
530        IF (AARR [J][O] == '*')
531           RETURN (O);
532        ELSE
533         RETURN (1);
534       }
535      }

536      RETURN (O); /* NO MATCH FOUND */

537     }

538     /* THIS MODULE POSITIONS THE POINTER TO THE
539      BEGINNING OF THE INPUT */


540     A_POS_IN (N)
541     INT N;
542     {

543      INT I, J;
544      CHAR C;

545      /* PRINTF ("%S\N", " ENTERED A_POS_IN"); */


546      I = O;
547      J = N;

548      DO
549       (C = AARR[J][++I]);
550       WHILE (C != ',');

551      RETURN (I);

552     }


553     /* THIS FUNCTION COMPARES THE INPUT OF ONE MODULE
554      WITH THE OUTPUT OF ANOTHER TO FIND A MATCH */

555     A_IN_OUT (N,L)
556     INT N,L;

557     {

558      INT I, J, C;

559      /* PRINTF ("%S\N", " ENTERED A_IN_OUT"); */

560      I = O;
561      C = O;
562      J = N;

563      WHILE (C == O)
564      {
```

```
565       C = STRCMP (HOLD_IN[++I],AARR[J][++L]);
566       /* PRINTF ("%C %C\N", HOLD_IN[I], AARR[J][L]); */
567       IF ((HOLD_IN[I] == ',') && (AARR[J][L] == ','))
568        RETURN (1); /* MATCH FOUND */
569      }

570      RETURN (0); /* NO MATCH FOUND */

571     }


572     /* THIS FUNCTION POSITIONS THE POINTER TO THE
573      BEGINNING OF THE OUTPUT FIELD */


574     A_POS_OUT (N)
575     INT N;

576     {
577      INT COMMA_COUNT, J, L;
578      CHAR C;

579      /* PRINTF ("%S\N", " ENTERED A_PO_OUT"); */
580      COMMA_COUNT = 0;
581      J = N;
582      L = 0;

583      WHILE (COMMA_COUNT < 2)
584      {
585       C = AARR[J][++L];
586       IF (C == ',')
587        COMMA_COUNT = COMMA_COUNT + 1;
588      }

589      RETURN (L);

590     }


591     /* THIS FUNCTION COMPARES OUTPUT TO INPUT TO FIND
592     A MATCH */

593     A_OUT_IN (J, L)
594     INT J, L;

595     {

596      INT C, I;

597      /* PRINTF ("%S\N", " ENTERED A_OUT_IN"); */

598      I = 0;
599      C = 0;

600      WHILE (C == 0)
601      {
602       C = STRCMP(HOLD_OUT[++I],AARR[J][++L]);
603       /* PRINTF ("%C %C\N", HOLD_OUT[I], AARR[J][L]); */
604       IF ((HOLD_OUT[I] == ',') && (AARR[J][L] == ','))
605        RETURN (1); /*MATCH FOUND */
```

```
606       }
607      RETURN (0); /* NO MATCH FOUND */
608     }

609     PRINT_AFF ()
610     {
611      TAB_COUNT = 1;
612      IN_COUNT = 0;
613      OUT_COUNT = 0;
614      SUP_COUNT = 0;
615      SUB_COUNT = 0;

616      PRINTF ("\F");
617      PRINTF ("      ** AFFERENT MODULES ** \N\N");
618      WHILE (1)
619      {
620       TAB_COUNT = TAB_COUNT + 1;

621       IF (MOD_ORD_TABLE[TAB_COUNT][1] == 0)
622        BREAK;
623       IF (MOD_ORD_TABLE[TAB_COUNT][0] == 100)
624        CONTINUE;

625       IF (MOD_ORD_TABLE[TAB_COUNT][1] == 99)
626        CONTINUE;

627       A_P_LOAD ();
628
629       LIST_COUNT = TAB_COUNT + 1;

630       WHILE (MOD_ORD_TABLE [LIST_COUNT][1] != 99)
631       {
632        IF (MOD_ORD_TABLE [LIST_COUNT][1] == 0)
633         BREAK;
634        IF (MOD_ORD_TABLE [LIST_COUNT][0] == 100)
635        {
636         LIST_COUNT = LIST_COUNT + 1;
637         CONTINUE;
638        }

639        A_P_MATCH ();


640        LIST_COUNT = LIST_COUNT + 1;

641       }

642       IF (MOD_ORD_TABLE [LIST_COUNT][1] == 0)
643        BREAK;
644       A_P_PRINT ();
645       IF (MOD_ORD_TABLE [TAB_COUNT + 1][1] == 99)
646        CONTINUE;
647      }
648     }
649     /* THIS FUNCTION LOADS THE FIRST RECORD ON
650     THE TABLE */
```

```
651    A_P_LOAD ()

652    {

653     INT I,J;

654     I = 0;
655     J = 0;

656     WHILE (H_MODULE [I] != ',')
657     {
658      H_MODULE [++I] = AARR [MOD_ORD_TABLE[TAB_COUNT][1]][++J];
659     }

660     I = 0;
661     IN_COUNT = 1;
662     WHILE (H_INPUT [IN_COUNT][I] != ',')
663      H_INPUT[IN_COUNT][++I] = AARR [MOD_ORD_TABLE[TAB_COUNT][1]][++J];

664     I = 0;
665     OUT_COUNT = 1;
666     WHILE (H_OUTPUT [OUT_COUNT][I] != ',')
667      H_OUTPUT [OUT_COUNT][++I] = AARR[MOD_ORD_TABLE[TAB_COUNT][1]][++J];

668     I = 0;
669     J = 0;
670     SUB_COUNT = 1;
671     IF (MOD_ORD_TABLE[TAB_COUNT][3] == 0)
672      H_SUB[SUB_COUNT][1] = ',';
673     ELSE
674     WHILE (H_SUB [SUB_COUNT][I] != ',')
675      H_SUB [SUB_COUNT][++I]=AARR[MOD_ORD_TABLE[TAB_COUNT][3]][++J];

676     SUP_COUNT = 1;
677     IF (MOD_ORD_TABLE [TAB_COUNT][2] == 99)
678     {
679      H_SUPER [SUP_COUNT][1] = 'M';
680      H_SUPER [SUP_COUNT][2] = 'A';
681      H_SUPER [SUP_COUNT][3] = 'I';
682      H_SUPER [SUP_COUNT][4] = 'N';
683      H_SUPER [SUP_COUNT][5] = ',';
684     }
685    ELSE
686     {
687      I =0;
688      J = 0;
689      WHILE (H_SUPER [1][I] != ',')
690       H_SUPER [1][++I] = AARR[MOD_ORD_TABLE[TAB_COUNT][2]][++J];
691     }


692    }
693    /* THIS FUNCTION MATCHES THE MODULES WITH SAME NAME
694    STORED IN THE TABLE AND GETS THEIR INPUTS AND
695    OUTPUTS IF THEY ARE DIFFERENT THAT THE OTHER ONES
696    FOR THE SAME MODULE */

697    A_P_MATCH ()
```

```
698    {
699    INT P, I, B, C,K,F,J,N,M,R,L,S;

700    P = LIST_COUNT - 1;

701     WHILE (MOD_ORD_TABLE [++P][1] != 99)
702     {
703      IF (MOD_ORD_TABLE[P][1] == 0)
704       BREAK;
705      C = 0;
706      I = 0;
707      S = 0;
708      WHILE (C == 0)
709      {
710       C = STRCMP(H_MODULE [++S],AARR[MOD_ORD_TABLE[P][1]][++I]);
711       IF ((H_MODULE [I] == ',') && (AARR[MOD_ORD_TABLE[P][1]][I] == ','))
712        BREAK;
713      }

714      IF (C != 0)
715       CONTINUE;

716      MOD_ORD_TABLE [P][0] = 100;


717      /* THE MODULE WAS FOUND AND NOW THE INPUT, OUTPUT,
718      SUB AND SUP WILL BE MATCHED */

719      K = 0;
720      F = IN_COUNT;
721      WHILE (++K <= F)
722      {
723       J = I;
724       C = 0;
725       L = 0;
726       WHILE (C == 0)
727       {
728        C = STRCMP (H_INPUT [K][++L], AARR[MOD_ORD_TABLE[P][1]][++J]);
729      IF ((H_INPUT [K][L] == ',') && (AARR[MOD_ORD_TABLE[P][1]][J] == ','))
730         BREAK;
731       }
732       IF (C == 0)
733        BREAK;

734       IF ((C != 0) && (K != F))
735        CONTINUE;

736       IN_COUNT = IN_COUNT + 1;
737       N = 0;
738       WHILE (H_INPUT [IN_COUNT][N] != ',')
739        H_INPUT [IN_COUNT][++N] = AARR[MOD_ORD_TABLE[P][1]][++I];
740      J = I;
741      }
742       M = J;

743    /* THIS IS TO MATCH THE OUTPUT */

744     K = 0;
```

```
745     F = OUT_COUNT;
746     WHILE (++K <= F)
747     {
748      R = M;
749      C = O;
750      L = O;
751      WHILE (C == O)
752      {
753       C = STRCMP (H_OUTPUT[K][++L],AARR[MOD_ORD_TABLE[P][1]][++R]);
754     IF ((H_OUTPUT [K][L] == ',') && (AARR[MOD_ORD_TABLE[P][1]][R] == ','))
755        BREAK;
756      }

757      IF (C == O)
758       BREAK;

759      IF ((C != O) && (K != F))
760       CONTINUE;

761      OUT_COUNT = OUT_COUNT + 1;
762      N = O;
763      WHILE (H_OUTPUT [OUT_COUNT][N] != ',')
764       H_OUTPUT [OUT_COUNT][++N] = AARR[MOD_ORD_TABLE[P][1]][++M];
765     }


766     B = SUP_COUNT;
767     K = O;

768     WHILE (++K <= B)
769     {
770      I = O;
771      C = O;
772      L = O;
773      IF (MOD_ORD_TABLE [P][2] == 99)
774       BREAK;

775      WHILE (C == O)
776      {
777       C = STRCMP(H_SUPER[K][++L],AARR[MOD_ORD_TABLE[P][2]][++I]);
778       IF ((H_SUPER[K][L] == ',') && (AARR[MOD_ORD_TABLE[P][2]][I] == ','))
779        BREAK;
780      }
781      IF (C == O)
782       BREAK;

783      IF ((C != O) && (K != B))
784       CONTINUE;

785      L = O;
786      I = O;
787      SUP_COUNT = SUP_COUNT + 1;
788      WHILE (H_SUPER [SUP_COUNT][L] != ',')
789       H_SUPER[SUP_COUNT][++L] = AARR[MOD_ORD_TABLE[P][2]][++I];

790     }

791     B = SUB_COUNT;
792     K = O;
```

```
793     WHILE (++K <= B)
794     {
795      L = 0;
796      I = 0;
797      C = 0;
798      IF (MOD_ORD_TABLE [P][3] == 0)
799       BREAK;

800      WHILE (C == 0)
801      {
802       C = STRCMP(H_SUB[K][++L],AARR[MOD_ORD_TABLE[P][3]][++I]);
803       IF ((H_SUB[K][L] == ',') && (AARR[MOD_ORD_TABLE[P][3]][I] == ','))
804        BREAK;
805      }
806      IF (C == 0)
807       BREAK;

808      IF ((C != 0) && (K != B))
809       CONTINUE;

810      L = 0;
811      I = 0;
812      SUB_COUNT = SUB_COUNT + 1;
813      WHILE (H_SUB [SUB_COUNT][L] != ',')
814       H_SUB[SUB_COUNT][++L] = AARR[MOD_ORD_TABLE[P][3]][++I];

815     }
816    }
817   }

818   /* THIS FUNCTION PRINTS THE MODULES IN
819    FORMAT 1 */

820   A_P_PRINT ()

821   {

822    INT I, K;

823    I = 0;
824    PRINTF ("%S", "      MODULE NAME: ");
825    WHILE (H_MODULE [I + 1] != ',')
826     PRINTF ("%C", H_MODULE [++I]);
827    PRINTF ("\N");

828    K = 0;
829    PRINTF("%S", "         INPUT(S): ");
830    WHILE (++K <= IN_COUNT)
831    {
832     I = 0;
833     WHILE (H_INPUT[K][I + 1] != ',')
834      PRINTF("%C", H_INPUT[K][++I]);
835     PRINTF ("\N");
836     IF (K != IN_COUNT)
837      PRINTF ("%S", "                    ");
838    }

839    K = 0;
840    PRINTF("%S","       OUTPUT(S): ");
```

```
841     WHILE (++K <= OUT_COUNT)
842     {
843      I = 0;
844      WHILE (H_OUTPUT [K][I + 1] != ',')
845       PRINTF ("%C", H_OUTPUT[K][++I]);
846      PRINTF ("\N");
847      IF (K != OUT_COUNT)
848       PRINTF("%S", "                    ");
849     }

850     K = 0;
851     PRINTF("%S", "SUPERORDINATE(S): ");
852     WHILE (++K <= SUP_COUNT)
853     {
854      I = 0;
855      WHILE (H_SUPER [K][I + 1] != ',')
856       PRINTF ("%C", H_SUPER[K][++I]);
857      PRINTF ("\N");
858      IF (K != SUP_COUNT)
859       PRINTF ("%S", "                    ");
860     }


861     K = 0;
862     PRINTF("%S", "  SUBORDINATE(S): ");
863     WHILE (++K <= SUB_COUNT)
864     {
865      I = 0;
866      WHILE (H_SUB[K][I + 1] != ',')
867       PRINTF ("%C",H_SUB [K][++I]);
868      PRINTF ("\N");
869      IF (K != SUB_COUNT)
870       PRINTF ("%S","                    ");
871     }

872       PRINTF ("\N");

873    }

874    /*********************************************************************/
875    /*********************************************************************/


876    EFF_P ()

877    {
878     INT I,J;
879     INT R;
880     I = -1;
881     J = -1;
882     MOD_NBR = 0;
883     ORD_COUNT = 1;
884     T_E_COUNT = 0;
885     SUB_COUNT = 1;


886     WHILE (++J <= 50)
887     {
888      WHILE ( ++I <= 4)
889       E_MOD_ORD_TABLE [J][I] = 0;
```

```
890       I = -1;
891      }


892     WHILE (T_E_COUNT <= T_COUNT)
893     {
894      R = GET_E_TRANS (T_E_COUNT);
895      /* THIS MODULE GETS A TRANSF. INPUT */
896      IF (R == 0)
897       BREAK;

898      SUB_COUNT = ORD_COUNT;

899      E_MOD_ORD_TABLE [ORD_COUNT][1] = 99;
900      E_MOD_ORD_TABLE [ORD_COUNT][2] = 99;
901      E_MOD_ORD_TABLE [ORD_COUNT][3] = 99;
902      SUB_COUNT = SUB_COUNT + 1;
903      ORD_COUNT = ORD_COUNT + 1;


904       J = 0;
905       WHILE (++J <= E_COUNT)
906        EARR [J][0] = ' ';

907      FIND_FIRST_E_MOD (.);    /* FUNCTION TO FIND FIRST MODULE */
908
909      WHILE ((E_MOD_ORD_TABLE [ORD_COUNT][1] != 0) ||
910            (E_MOD_ORD_TABLE [SUB_COUNT][3] != 0))
911      {


912       IF (E_MOD_ORD_TABLE [ORD_COUNT][1] == 0)
913       {
914        E_MOD_ORD_TABLE [ORD_COUNT][1] = E_MOD_ORD_TABLE [SUB_COUNT][3];
915        SUB_COUNT = SUB_COUNT + 1;
916       }

917     /********* FOR TEST ONLY **********/
918     /*
919     IF (ORD_COUNT > 10)
920       BREAK;
921     */
922     /************** END ***************/

923      MOD_NBR = E_MOD_ORD_TABLE [ORD_COUNT][1];
924      IF (MOD_NBR == 0)
925       BREAK;

926      EARR [E_MOD_ORD_TABLE[ORD_COUNT][1]][0] = '*';
927      E_LOAD_ALL (E_MOD_ORD_TABLE [ORD_COUNT][1]);
928      F_E_SUBO (); /* THIS FUNCTION FINDS SUBORDINATE MODULE */

929      F_E_SUPE (); /* THIS FUNCTION FINDS SUPERORDINATE MODULE */
930      FIND_E_MOD ();
931     /* THE ABOVE MODULE FINDS ANY MODULES WITH THE SAME NAME */
932
933      ORD_COUNT = ORD_COUNT + 1;
934      }

935     IF (E_MOD_ORD_TABLE [ORD_COUNT][1] == 0)
```

```
936      SUB_COUNT = SUB_COUNT + 1;

937     }
938     }

939     /* THIS FUNCTIONS OBTAINS THE OUTPUT FROM ONE OF THE
940      TRANSFORM MODULES MARKED ¢B¢ OR ¢D¢ AND LOADS IT
941      INTO HOLD_OUT */

942     GET_E_TRANS ()

943     {
944      CHAR C;
945      INT I, J, COMME_COUNT;
946      I = 0;
947      J = 0;
948      COMME_COUNT = 0;


949       WHILE (++T_E_COUNT <= T_COUNT)
950        {
951         IF ((TARR [T_E_COUNT][0] != 'B') &&
952             (TARR [T_E_COUNT][0] != 'D'))
953           CONTINUE;
954         ELSE
955         {
956          WHILE (COMME_COUNT < 2)
957          {
958           C = (TARR [T_E_COUNT][++J]);
959           IF (C == ',')
960            COMME_COUNT = COMME_COUNT + 1;
961          }
962          DO
963           {
964           HOLD_OUT[++I] = TARR[T_E_COUNT][++J];
965            /* PRINTF ("%C", HOLD_OUT[I]); */
966           }
967           WHILE (HOLD_OUT[I] != ',');
968          RETURN (1);
969         }
970       }
971      }

972     /* THIS MODULE FINDS THE FIRST EFFERENT MODULE THAT
973     IS ATTACHED TO THE TRANSFORM OR MAIN MODULE */

974     FIND_FIRST_E_MOD ()

975     {
976      INT N,J,P;

977      N = MOD_NBR;

978
979      WHILE ((N != (MOD_NBR -1)) && (P != 1))
980      {
981       IF (N == E_COUNT)
982       {
983        IF (MOD_NBR == 1)
984         BREAK;
```

```
985         ELSE
986          N = 0;
987        }
988        N = N + 1;
989        J = E_POS_IN (N);
990        P = E_OUT_IN (N,J);
991       }
992       IF (P == 1)
993       {
994        E_MOD_ORD_TABLE [ORD_COUNT][1] = N;
995        MOD_NBR = N;
996       }
997       ELSE
998        E_MOD_ORD_TABLE [ORD_COUNT][1] = 0;

999     }

1000    /* THIS FUNCTION FINDS THE SUBMODULE TO THE
1001    MODULE */
1002    F_E_SUB0 ()

1003    {
1004     INT N,K,F;

1005     N = MOD_NBR;

1006
1007     WHILE ((N != (MOD_NBR -1)) && (F != 1))
1008     {
1009      IF (N == E_COUNT)
1010      {
1011       IF (MOD_NBR == 1)
1012        BREAK;
1013       ELSE
1014        N = 0;
1015      }
1016      N = N + 1;
1017      K = E_POS_IN (N);
1018      F = E_OUT_IN (N,K);
1019     }
1020     IF (F == 1)
1021      E_MOD_ORD_TABLE [ORD_COUNT][3] = N;
1022     ELSE
1023      E_MOD_ORD_TABLE [ORD_COUNT][3] = 0;


1024    }

1025    /* THIS FUNCTION FINDS THE SUPERORDINATE OF A
1026     MODULE */

1027    F_E_SUPE ()

1028    {
1029     INT N,L,H;

1030     N = MOD_NBR;
1031     H = 0;
```

```
1032
1033     WHILE ((N != (MOD_NBR -1)) && (H != 1))
1034     {
1035      IF (N == E_COUNT)
1036      {
1037       IF (MOD_NBR == 1)
1038        BREAK;
1039       ELSE
1040        N = 0;
1041      }
1042      N = N + 1;
1043      L = E_POS_OUT (N);
1044      H = E_IN_OUT (N,L);
1045     }
1046     IF (H == 1)
1047      E_MOD_ORD_TABLE [ORD_COUNT][2] = N;
1048     ELSE
1049      E_MOD_ORD_TABLE [ORD_COUNT][2] = 99;

1050    }

1051    /* THIS FUNCTION LOCATES ANY OTHER MODULE WITH THE
1052     SAME NAME AS THE CURRENT ONE */

1053    FIND_E_MOD ()

1054    {
1055     INT N,G;

1056     N = MOD_NBR;
1057     /* PRINTF ("%S\N", " ENTERED MODULE FIND_E_MOD "); */
1058
1059     WHILE ((N != (MOD_NBR -1)) && (G != 1))
1060     {
1061      IF (N == E_COUNT)
1062      {
1063       IF (MOD_NBR == 1)
1064        BREAK;
1065       ELSE
1066        N = 0;
1067      }
1068      N = N + 1;
1069      G = E_MOD_MOD (N);
1070     }
1071     IF (G == 1)
1072     {
1073      E_MOD_ORD_TABLE [ORD_COUNT + 1][1] = N;
1074     }

1075    }

1076    /* THIS FUNCTION LOADS MODULE NAME INPUT AND OUTPUT INTO
1077     HOLD_MOD, HOLD_IN, HOLD_OUT */

1078    E_LOAD_ALL(N)
1079    INT N;
1080    {
1081    INT I, J;
1082    CHAR C;
```

```
1083    I = 0;
1084    J = 0;

1085    /* PRINTF ("%S\N", " ENTERED E_LOAD_ALL "); */

1086    DO
1087     HOLD_MOD[++J] = EARR[N][++I];
1088     WHILE (HOLD_MOD[J] != ',');

1089    J = 0;
1090    DO
1091     HOLD_IN[++J] = EARR[N][++I];
1092     WHILE (HOLD_IN [J] != ',');

1093    J = 0;
1094    DO
1095     HOLD_OUT[++J] = EARR[N][++I];
1096     WHILE (HOLD_OUT[J] != ',');

1097    }

1098    /* THIS FUNCTION COMPARES TWO MODULES NAME TO
1099     FIND OUT IF THEY MATCH */

1100    E_MOD_MOD (N)
1101    INT N;

1102    {

1103     INT I, L, J, C;

1104     /* PRINTF ("%S\N", " ENTERED E_MOD_MOD"); */

1105     I = 0;
1106     L = 0;
1107     C = 0;
1108     J = N;

1109     WHILE (C == 0)
1110     {
1111      C = STRCMP (HOLD_MOD[++I],EARR[J][++L]);
1112      /* PRINTF ("%C %C\N", HOLD_MOD[I], EARR[J][L]); */
1113      IF ((HOLD_MOD[I] == ',') && (EARR[J][L] == ','))
1114      {
1115       IF (EARR [J][0] == '*')
1116          RETURN (0);
1117       ELSE
1118        RETURN (1);
1119      }
1120     }

1121     RETURN (0); /* NO MATCH FOUND */

1122    }

1123    /* THIS MODULE POSITIONS THE POINTER TO THE
1124     BEGINNING OF THE INPUT */


1125    E_POS_IN (N)
```

```
1126    INT N;
1127    {

1128     INT I, J;
1129     CHAR C;

1130     /* PRINTF ("%S\N", " ENTERED E_POS_IN"); */


1131     I = 0;
1132     J = N;

1133     DO
1134      (C = EARR[J][++I]);
1135      WHILE (C != ',');

1136     RETURN (I);

1137    }


1138    /* THIS FUNCTION COMPARES THE INPUT OF ONE MODULE
1139     WITH THE OUTPUT OF ANOTHER TO FIND A MATCH */

1140    E_IN_OUT (N,L)
1141    INT N,L;

1142    {

1143     INT I, J, C;

1144      /* PRINTF ("%S\N", " ENTERED E_IN_OUT"); */

1145     I = 0;
1146     C = 0;
1147     J = N;

1148     WHILE (C == 0)
1149     {
1150      C = STRCMP (HOLD_IN[++I],EARR[J][++L]);
1151      /* PRINTF ("%C %C\N", HOLD_IN[I], EARR[J][L]); */
1152      IF ((HOLD_IN[I] == ',') && (EARR[J][L] == ','))
1153       RETURN (1); /* MATCH FOUND */
1154     }

1155     RETURN (0); /* NO MATCH FOUND */

1156    }

1157    /* THIS FUNCTION POSITIONS THE POINTER TO THE
1158     BEGINNING OF THE OUTPUT FIELD */


1159    E_POS_OUT (N)
1160    INT N;

1161    {
1162     INT COMME_COUNT, J, L;
1163     CHAR C;
```

```
1164     /* PRINTF ("%S\N", " ENTERED E_PO_OUT"); */
1165     COMME_COUNT = 0;
1166     J = N;
1167     L = 0;

1168     WHILE (COMME_COUNT < 2)
1169     {
1170      C = EARR[J][++L];
1171      IF (C == ',')
1172       COMME_COUNT = COMME_COUNT + 1;
1173     }

1174     RETURN (L);

1175    }


1176    /* THIS FUNCTION COMPARES OUTPUT TO INPUT TO FIND
1177    A MATCH */

1178    E_OUT_IN (J, L)
1179    INT J, L;

1180    {

1181     INT C, I;

1182     /* PRINTF ("%S\N", " ENTERED E_OUT_IN"); */

1183     I = 0;
1184     C = 0;

1185     WHILE (C == 0)
1186     {
1187      C = STRCMP(HOLD_OUT[++I],EARR[J][++L]);
1188      /* PRINTF ("%C %C\N", HOLD_OUT[I], EARR[J][L]); */
1189      IF ((HOLD_OUT[I] == ',') && (EARR[J][L] == ','))
1190.      RETURN (1); /*MATCH FOUND */
1191      }

1192     RETURN (0); /* NO MATCH FOUND */
1193    }

1194    PRINT_EFF ()

1195    {

1196     TAB_COUNT = 1;
1197     IN_COUNT = 0;
1198     OUT_COUNT = 0;
1199     SUP_COUNT = 0;
1200     SUB_COUNT = 0;

1201     PRINTF ("\F");
1202     PRINTF ( "      ** EFFERENT MODULES ** \N\N");
1203     WHILE (1)
1204     {
1205      TAB_COUNT = TAB_COUNT + 1;
```

```
1206      IF (E_MOD_ORD_TABLE[TAB_COUNT][1] == 0)
1207       BREAK;
1208      IF (E_MOD_ORD_TABLE[TAB_COUNT][0] == 100)
1209       CONTINUE;

1210      IF (E_MOD_ORD_TABLE[TAB_COUNT][1] == 99)
1211       CONTINUE;

1212      E_P_LOAD ();
1213
1214      LIST_COUNT = TAB_COUNT + 1;

1215      WHILE (E_MOD_ORD_TABLE [LIST_COUNT][1] != 99)
1216      {
1217       IF (E_MOD_ORD_TABLE [LIST_COUNT][1] == 0)
1218        BREAK;
1219       IF (E_MOD_ORD_TABLE [LIST_COUNT][0] == 100)
1220       {
1221        LIST_COUNT = LIST_COUNT + 1;
1222        CONTINUE;
1223       }

1224       E_P_MATCH ();


1225       LIST_COUNT = LIST_COUNT + 1;

1226      }

1227      IF (E_MOD_ORD_TABLE [LIST_COUNT][1] == 0)
1228       BREAK;
1229      E_P_PRINT ();
1230      IF (E_MOD_ORD_TABLE [TAB_COUNT + 1][1] == 99)
1231       CONTINUE;
1232     }
1233    }
1234    /* THIS FUNCTION LOADS THE FIRST RECORD ON
1235    THE TABLE */

1236    E_P_LOAD ()

1237    {

1238     INT I,J;

1239     I = 0;
1240     J = 0;

1241     WHILE (H_MODULE [I] != ',')
1242     {
1243      H_MODULE [++I] = EARR [E_MOD_ORD_TABLE[TAB_COUNT][1]][++J];
1244     }

1245     I = 0;
1246     IN_COUNT = 1;
1247     WHILE (H_INPUT [IN_COUNT][I] != ',')
1248      H_INPUT[IN_COUNT][++I] = EARR [E_MOD_ORD_TABLE[TAB_COUNT][1]][++J];

1249     I = 0;
1250     OUT_COUNT = 1;
```

```
1251     WHILE (H_OUTPUT [OUT_COUNT][I] != ',')
1252      H_OUTPUT [OUT_COUNT][++I] = EARR[E_MOD_ORD_TABLE[TAB_COUNT][1]][++J];

1253     I = 0;
1254     J = 0;
1255     SUB_COUNT = 1;
1256     IF (E_MOD_ORD_TABLE[TAB_COUNT][3] == 0)
1257      H_SUB[SUB_COUNT][1] = ',';
1258     ELSE
1259     WHILE (H_SUB [SUB_COUNT][I] != ',')
1260      H_SUB [SUB_COUNT][++I]=EARR[E_MOD_ORD_TABLE[TAB_COUNT][3]][++J];

1261     SUP_COUNT = 1;
1262     IF (E_MOD_ORD_TABLE [TAB_COUNT][2] == 99)
1263     {
1264      H_SUPER [SUP_COUNT][1] = 'M';
1265      H_SUPER [SUP_COUNT][2] = 'A';
1266      H_SUPER [SUP_COUNT][3] = 'I';
1267      H_SUPER [SUP_COUNT][4] = 'N';
1268      H_SUPER [SUP_COUNT][5] = ',';
1269     }
1270    ELSE
1271     {
1272      I =0;
1273      J = 0;
1274      WHILE (H_SUPER [1][I] != ',')
1275       H_SUPER [1][++I] = EARR[E_MOD_ORD_TABLE[TAB_COUNT][2]][++J];
1276     }


1277    }
1278    /* THIS FUNCTION MATCHES THE MODULES WITH SAME NAME
1279    STORED IN THE TABLE AND GETS THEIR INPUTS AND
1280    OUTPUTS IF THEY ARE DIFFERENT THAT THE OTHER ONES
1281    FOR THE SAME MODULE */

1282    E_P_MATCH ()

1283    {

1284    INT P, I, B, C,K,F,J,N,M,R,L,S;


1285    P = LIST_COUNT - 1;

1286     WHILE (E_MOD_ORD_TABLE [++P][1] != 99)
1287     {
1288      IF (E_MOD_ORD_TABLE[P][1] == 0)
1289       BREAK;
1290      C = 0;
1291      I = 0;
1292      S = 0;
1293      WHILE (C == 0)
1294      {
1295       C = STRCMP(H_MODULE [++S],EARR[E_MOD_ORD_TABLE[P][1]][++I]);
1296       IF ((H_MODULE [I] == ',') && (EARR[E_MOD_ORD_TABLE[P][1]][I] == ','))
1297        BREAK;
1298      }

1299      IF (C != 0)
```

```
1300        CONTINUE;
1301       E_MOD_ORD_TABLE [P][O] = 100;

1302       /* THE MODULE WAS FOUND AND NOW THE INPUT, OUTPUT,
1303       SUB AND SUP WILL BE MATCHED */

1304       K = O;
1305       F = IN_COUNT;
1306       WHILE (++K <= F)
1307       {
1308        J = I;
1309        C = O;
1310        L = O;
1311        WHILE (C == O)
1312        {
1313         C = STRCMP (H_INPUT [K][++L], EARR[E_MOD_ORD_TABLE[P][1]][++J]);
1314      IF ((H_INPUT [K][L] == ',') && (EARR[E_MOD_ORD_TABLE[P][1]][J] == ','))
1315           BREAK;
1316        }
1317        IF (C == O)
1318         BREAK;

1319        IF ((C != O) && (K != F))
1320         CONTINUE;

1321        IN_COUNT = IN_COUNT + 1;
1322        N = O;
1323        WHILE (H_INPUT [IN_COUNT][N] != ',')
1324         H_INPUT [IN_COUNT][++N] = EARR[E_MOD_ORD_TABLE[P][1]][++I];
1325       J = I;
1326       }
1327        M = J;

1328     /* THIS IS TO MATCH THE OUTPUT */

1329      K = O;
1330      F = OUT_COUNT;
1331      WHILE (++K <= F)
1332      {
1333       R = M;
1334       C = O;
1335       L = O;
1336       WHILE (C == O)
1337       {
1338        C = STRCMP (H_OUTPUT[K][++L],EARR[E_MOD_ORD_TABLE[P][1]][++R]);
1339     IF ((H_OUTPUT [K][L] == ',') && (EARR[E_MOD_ORD_TABLE[P][1]][R] == ','))
1340         BREAK;
1341       }

1342       IF (C == O)
1343        BREAK;

1344       IF ((C != O) && (K != F))
1345        CONTINUE;

1346       OUT_COUNT = OUT_COUNT + 1;
1347       N = O;
1348       WHILE (H_OUTPUT [OUT_COUNT][N] != ',')
```

```
1349        H_OUTPUT [OUT_COUNT][++N] = EARR[E_MOD_ORD_TABLE[P][1]][++M];
1350      }


1351      B = SUP_COUNT;
1352      K = 0;

1353      WHILE (++K <= B)
1354      {
1355       I = 0;
1356       C = 0;
1357       L = 0;
1358       IF (E_MOD_ORD_TABLE [P][2] == 99)
1359        BREAK;

1360       WHILE (C == 0)
1361       {
1362        C = STRCMP(H_SUPER[K][++L],EARR[E_MOD_ORD_TABLE[P][2]][++I]);
1363      IF ((H_SUPER[K][L] == ',') && (EARR[E_MOD_ORD_TABLE[P][2]][I] == ','))
1364        BREAK;
1365       }
1366       IF (C == 0)
1367        BREAK;

1368       IF ((C != 0) && (K != B))
1369        CONTINUE;

1370       L = 0;
1371       I = 0;
1372       SUP_COUNT = SUP_COUNT + 1;
1373       WHILE (H_SUPER [SUP_COUNT][L] != ',')
1374        H_SUPER[SUP_COUNT][++L] = EARR[E_MOD_ORD_TABLE[P][2]][++I];

1375      }

1376      B = SUB_COUNT;
1377      K = 0;

1378      WHILE (++K <= B)
1379      {
1380       L = 0;
1381       I = 0;
1382       C = 0;
1383       IF (E_MOD_ORD_TABLE [P][3] == 0)
1384        BREAK;

1385       WHILE (C == 0)
1386       {
1387        C = STRCMP(H_SUB[K][++L],EARR[E_MOD_ORD_TABLE[P][3]][++I]);
1388        IF ((H_SUB[K][L] == ',') && (EARR[E_MOD_ORD_TABLE[P][3]][I] == ','))
1389         BREAK;
1390       }
1391       IF (C == 0)
1392        BREAK;

1393       IF ((C != 0) && (K != B))
1394        CONTINUE;

1395       L = 0;
1396       I = 0;
```

```
1397        SUB_COUNT = SUB_COUNT + 1;
1398        WHILE (H_SUB [SUB_COUNT][L] != ',')
1399         H_SUB[SUB_COUNT][++L] = EARR[E_MOD_ORD_TABLE[P][3]][++I];

1400        }
1401       }
1402      }

1403      /* THIS FUNCTION PRINTS THE MODULES IN
1404       FORMAT 1 */


1405      E_P_PRINT ()

1406      {

1407       INT I, K;

1408       I = 0;
1409       PRINTF ("%S", "      MODULE NAME: ");
1410       WHILE (H_MODULE [I + 1] != ',')
1411        PRINTF ("%C", H_MODULE [++I]);
1412       PRINTF ("\N");

1413       K = 0;
1414       PRINTF("%S", "         INPUT(S). ");
1415       WHILE (++K <= IN_COUNT)
1416       {
1417        I = 0;
1418        WHILE (H_INPUT[K][I + 1] != ',')
1419         PRINTF("%C", H_INPUT[K][++I]);
1420        PRINTF ("\N");
1421        IF (K != IN_COUNT)
1422         PRINTF ("%S", "                    ");
1423       }

1424       K = 0;
1425       PRINTF("%S","        OUTPUT(S): ");
1426       WHILE (++K <= OUT_COUNT)
1427       {
1428        I = 0;
1429        WHILE (H_OUTPUT [K][I + 1] != ',')
1430         PRINTF ("%C", H_OUTPUT[K][++I]);
1431        PRINTF ("\N");
1432        IF (K != OUT_COUNT)
1433         PRINTF("%S", "                    ");
1434       }

1435       K = 0;
1436       PRINTF("%S", "SUPERORDINATE(S): ");
1437       WHILE (++K <= SUP_COUNT)
1438       {
1439        I = 0;
1440        WHILE (H_SUPER [K][I + 1] != ',')
1441         PRINTF ("%C", H_SUPER[K][++I]);
1442        PRINTF ("\N");
1443        IF (K != SUP_COUNT)
1444         PRINTF ("%S", "                    ");
1445       }
```

```
1446     K = 0;
1447     PRINTF("%S", "  SUBORDINATE(S): ");
1448     WHILE (++K <= SUB_COUNT)
1449     {
1450      I = 0;
1451      WHILE (H_SUB[K][I + 1] != ',')
1452       PRINTF ("%C",H_SUB [K][++I]);
1453      PRINTF ("\N");
1454      IF (K != SUB_COUNT)
1455       PRINTF ("%S","                    ");
1456     }

1457       PRINTF ("\N");

1458    }
1459    /*****************************************************************/
1460    /*****************************************************************/

1461    /* THIS FUNCTION AND ITS SUBFUNCTIONS PROCESS AND PRINT THE
1462      TRANSFORM MODULES */

1463    PRINT_TRANSF()

1464    {


1465    TAB_COUNT = 0;
1466    LIST_COUNT = 0;
1467    IN_COUNT = 0;
1468    OUT_COUNT = 0;

1469    PRINTF ("\F");
1470    PRINTF ("     ** TRANSFORM MODULES ** \N\N");
1471    WHILE (TAB_COUNT < T_COUNT)
1472    {
1473     TAB_COUNT = TAB_COUNT + 1;
1474     IF (TARR [TAB_COUNT][0] == 'U')
1475      CONTINUE;

1476     T_P_LOAD();

1477     LIST_COUNT = TAB_COUNT;

1478     T_P_MATCH();

1479     T_P_PRINT();


1480    }

1481    }

1482    /* THIS FUNCTION LOAD THE MODULE NAME, INPUT AND
1483     OUTPUT IN HOLD AREAS */

1484    T_P_LOAD()

1485    {
1486     INT I,J;
```

```
1487      I = 0;
1488      J = 0;


1489      WHILE (H_MODULE [I] != ',')
1490       H_MODULE [++I] = TARR [TAB_COUNT][++J];

1491      I = 0;
1492      IN_COUNT = 1;
1493      WHILE (H_INPUT [IN_COUNT][I] != ',')
1494       H_INPUT [IN_COUNT] [++I] = TARR [TAB_COUNT][++J];

1495      I = 0;
1496      OUT_COUNT = 1;
1497      WHILE (H_OUTPUT [OUT_COUNT][I] != ',')
1498       H_OUTPUT [OUT_COUNT] [++I] = TARR [TAB_COUNT][++J];

1499     }

1500     T_P_MATCH()

1501     {

1502     INT P, I, B, C,K,F,J,N,M,R,L,S;


1503     P = LIST_COUNT - 1;

1504      WHILE (++LIST_COUNT <= T_COUNT)
1505      {
1506       C = 0;
1507       I = 0;
1508       S = 0;
1509       WHILE (C == 0)
1510       {
1511        C = STRCMP(H_MODULE [++S],TARR[LIST_COUNT][++I]);
1512        IF ((H_MODULE [I] == ',') && (TARR[LIST_COUNT][I] == ','))
1513         BREAK;
1514       }

1515       IF (C != 0)
1516        CONTINUE;

1517       TARR [LIST_COUNT][0] = 'U';


1518       /* THE MODULE WAS FOUND AND NOW THE INPUT, OUTPUT,
1519       WILL BE MATCHED */

1520       K = 0;
1521       F = IN_COUNT;
1522       WHILE (++K <= F)
1523       {
1524        J = I;
1525        C = 0;
1526        L = 0;
1527        WHILE (C == 0)
1528        {
1529         C = STRCMP (H_INPUT [K][++L], TARR[LIST_COUNT][++J]);
```

```
1530        IF ((H_INPUT [K][L] == ',') && (TARR[LIST_COUNT][J] == ','))
1531         BREAK;
1532       }
1533       IF (C == 0)
1534        BREAK;

1535       IF ((C != 0) && (K != F))
1536        CONTINUE;

1537       IN_COUNT = IN_COUNT + 1;
1538       N = 0;
1539       WHILE (H_INPUT [IN_COUNT][N] != ',')
1540        H_INPUT [IN_COUNT][++N] = TARR[LIST_COUNT][++I];
1541      J = I;
1542      }
1543       M = J;

1544     /* THIS IS TO MATCH THE OUTPUT */

1545      K = 0;
1546      F = OUT_COUNT;
1547      WHILE (++K <= F)
1548      {
1549       R = M;
1550       C = 0;
1551       L = 0;
1552       WHILE (C == 0)
1553       {
1554        C = STRCMP (H_OUTPUT[K][++L],TARR[LIST_COUNT][++R]);
1555        IF ((H_OUTPUT [K][L] == ',') && (TARR[LIST_COUNT][R] == ','))
1556         BREAK;
1557       }

1558       IF (C == 0)
1559        BREAK;

1560       IF ((C !='0) && (K != F))
1561        CONTINUE;

1562       OUT_COUNT = OUT_COUNT + 1;
1563       N = 0;
1564       WHILE (H_OUTPUT [OUT_COUNT][N] != ',')
1565        H_OUTPUT [OUT_COUNT][++N] = TARR[LIST_COUNT][++M];
1566      }
1567     }
1568     }

1569     T_P_PRINT()


1570     {

1571      INT I, K;

1572      I = 0;
1573      PRINTF ("%S", "      MODULE NAME: ");
1574      WHILE (H_MODULE [I + 1] != ',')
1575       PRINTF ("%C", H_MODULE [++I]);
1576      PRINTF ("\N");
```

```
1577      K = 0;
1578      PRINTF("%S", "         INPUT(S): ");
1579      WHILE (++K <= IN_COUNT)
1580      {
1581       I = 0;
1582       WHILE (H_INPUT[K][I + 1] != ',')
1583        PRINTF("%C", H_INPUT[K][++I]);
1584       PRINTF ("\N");
1585       IF (K != IN_COUNT)
1586        PRINTF ("%S", "                    ");
1587      }

1588      K = 0;
1589      PRINTF("%S","        OUTPUT(S): ");
1590      WHILE (++K <= OUT_COUNT)
1591      {
1592       I = 0;
1593       WHILE (H_OUTPUT [K][I + 1] != ',')
1594        PRINTF ("%C", H_OUTPUT[K][++I]);
1595       PRINTF ("\N");
1596       IF (K != OUT_COUNT)
1597        PRINTF("%S", "                    ");
1598      }
1599      PRINTF ("\N");
1600     }

1601     /* THIS FUNCTION PRINTS THE AFFERENT, TRANSFORM AND
1602     EFFERENT IN A TABULAR FORMAT */

1603     PRINT_ORDER()

1604     {

1605      INT I, N, AFF_COUNT, TRANSF_COUNT, EFF_COUNT;
1606      INT AFF_NO, EFF_NO;
1607      CHAR PRINT_LINE[90];
1608      I = 0;
1609      AFF_COUNT = 0;
1610      TRANSF_COUNT = 0;
1611      EFF_COUNT = 0;
1612      AFF_NO = 1;
1613      EFF_NO = 1;


1614      PRINTF ("%13S %36S %27S\N \N", "AFFERENT", "TRANSFORM", "EFFERENT");
1615      WHILE ((MOD_ORD_TABLE [AFF_COUNT][1] != 0) ||
1616            (TRANSF_COUNT < T_COUNT) ||
1617            (E_MOD_ORD_TABLE [EFF_COUNT][1] != 0))
1618      {
1619       ++AFF_COUNT;
1620       ++TRANSF_COUNT;
1621       ++EFF_COUNT;
1622       /* PRINTF ("%S\N", "MAIN WHILE LOOP"); */
1623       I = 0;
1624       DO
1625        PRINT_LINE[++I] = ' ';
1626       WHILE (I < 90);

1627       I = 0;
1628       WHILE ((I == 0) && (MOD_ORD_TABLE[AFF_COUNT][1] != 0))
```

```
1629      {
1630       N = 5;
1631       /* PRINTF ("%S\N", " AFF. LOOP");
1632      PRINTF ("%S %D %S %D\N", "I = ", I, "AFF_COUNT = ", AFF_COUNT);
1633      PRINTF ("%S %D\N", "MOD_ORD_TABLE", MOD_ORD_TABLE [AFF_COUNT][0]);
1634      PRINTF ("%S %D\N", "MOD_ORD_TABLE1", MOD_ORD_TABLE [AFF_COUNT][1]); */
1635       IF (MOD_ORD_TABLE [AFF_COUNT][0] == 100)
1636       {
1637        AFF_COUNT = AFF_COUNT + 1;
1638        CONTINUE;
1639       }
1640       IF (MOD_ORD_TABLE [AFF_COUNT][1] == 99)
1641        IF (MOD_ORD_TABLE [AFF_COUNT + 1][1] == 0)
1642         BREAK;

1643       ELSE
1644       {
1645        /* PRINTF ("%S\N", "LOADING INIT. AFF."); */
1646        PRINT_LINE [N] = 'A';
1647        PRINT_LINE [++N] = 'F';
1648        PRINT_LINE [++N] = 'F';
1649        I = 1;
1650        BREAK;
1651       }

1652       I = 0;
1653       N = 5;
1654       WHILE (AARR[MOD_ORD_TABLE[AFF_COUNT][1]][I + 1] != ',')
1655       {
1656         /* PRINTF ("%C\N", PRINT_LINE[N]); */
1657        PRINT_LINE[++N] = AARR[MOD_ORD_TABLE[AFF_COUNT][1]][++I];
1658       }

1659      }

1660     /* COLUMN OF TRANSFORMS */

1661      I = 0;
1662
1663      WHILE ((I == 0) && (TRANSF_COUNT <= T_COUNT))
1664      {
1665       /* PRINTF ("%S\N", " MAIN TRANSF. LOOP ");
1666      PRINTF ("%S %D %S %D\N", "I = ", I, "TRANSF_COUNT = ", TRANSF_COUNT);
1667      PRINTF ("%C\N", TARR[TRANSF_COUNT][0]); */
1668       IF (TARR[TRANSF_COUNT][0] == 'U')
1669       {
1670        TRANSF_COUNT = TRANSF_COUNT + 1;
1671        CONTINUE;
1672       }

1673       I = 0;
1674       N = 40;
1675       WHILE (TARR[TRANSF_COUNT][++I] != ',')
1676       {
1677        /* PRINTF ("%C\N", PRINT_LINE[N]); */
1678        PRINT_LINE [++N] = TARR [TRANSF_COUNT][I];
1679       }
1680      }
```

```
1681    /* COLUMN OF EFFERENTS */
1682      I = 0;
1683      WHILE ((I == 0) && (E_MOD_ORD_TABLE[EFF_COUNT][1] != 0))
1684      {
1685       N = 70;
1686       /* PRINTF ("%S\N", " EFF. LOOP ");
1687      PRINTF ("%S %D %S %D\N", "I = ", I, "EFF_COUNT = ", EFF_COUNT);
1688      PRINTF ("%S %D\N", "E_MOD_1", E_MOD_ORD_TABLE[EFF_COUNT][0]);
1689      PRINTF ("%S %D\N", "E_MOD_2", E_MOD_ORD_TABLE[EFF_COUNT][1]); */
1690       IF (E_MOD_ORD_TABLE [EFF_COUNT][0] == 100)
1691       {
1692        EFF_COUNT = EFF_COUNT + 1;
1693        CONTINUE;
1694       }
1695       IF (E_MOD_ORD_TABLE [EFF_COUNT][1] == 99)
1696        IF (E_MOD_ORD_TABLE [EFF_COUNT + 1][1] == 0)
1697         BREAK;
1698       ELSE
1699       {
1700        /* PRINTF ("%S\N", "LOADING INIT EFF"); */
1701        PRINT_LINE [N] = 'E';
1702        PRINT_LINE [++N] = 'F';
1703        PRINT_LINE [++N] = 'F';
1704        PRINT_LINE [++N] = EFF_NO;
1705        I = 1;
1706        EFF_NO = EFF_NO + 1;
1707        BREAK;
1708       }
1709       I = 0;
1710       N = 70;
1711       WHILE (EARR[E_MOD_ORD_TABLE[EFF_COUNT][1]][I + 1] != ',')
1712       {
1713        /* PRINTF ("%C\N", PRINT_LINE[N]); */
1714        PRINT_LINE[++N] = EARR[E_MOD_ORD_TABLE[EFF_COUNT][1]][++I];
1715       }
1716    }
1717      PRINT_LINE [++N] = '\0';
1718      I = 0;
1719      DO
1720      PRINTF ("%C", PRINT_LINE[++I]);
1721      WHILE (I <= N);
1722      PRINTF ("\N");
1723     }
1724    }
```

TRANSFORMING DATA FLOW DIAGRAMS TO SOFTWARE STRUCTURE USING THE YOURDON-CONSTANTINE METHODOLOGY

by

FRANCO ANTONUCCI

B.S.E.E., Fairleigh Dickenson University, 1975

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

The Yourdon-Constantine design methodology was introduced in 1974. The methodology is based on the concept that the structure of a system should be determined by the flow of data through the system. The flow of data through a system is represented by a data flow diagram.

By using the concepts of transform analysis of the Yourdon-Constantine methodology and information derived from the data flow diagram, this project will generate a 'first' cut software structure of a system. This project is part of a group of projects at Kansas State University to develop automated tools for the system development life cycle.

1430-60
CD-53