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:

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

Chapter 1 - Overview.............................................. 1
1.1 Introduction.................................................. 1
1.2 Fifth Generation............................................. 3
1.3 Design Approaches.......................................... 4
1.4 Yourdon-Constantine system design methodology....... 5
1.5 Transform Analysis........................................... 7
1.6 Summary....................................................... 14

Chapter 2 - Requirements.................................... 17
2.1 Overview...................................................... 17
2.2 Input.......................................................... 17
2.3 User interaction.............................................. 18
2.4 Output......................................................... 19
2.5 General characteristics................................... 20

Chapter 3 - Design............................................. 21
3.1 Introduction.................................................. 21
3.2 Process input................................................ 21
3.3 Reformatting the input file............................... 24
3.4 Load input..................................................... 25
3.5 Process transform......................................... 27
3.6 Process afferent and efferent............................ 28
3.7 Print afferent, efferent and transform................... 31
3.8 Produce tabular output.................................... 33
3.9 Conclusion................................................... 33
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

Figure 1.1 -- DATA FLOW DIAGRAM............................. 6
Figure 1.2 -- PARTITIONED DATA FLOW DIAGRAM................. 9
Figure 1.3 -- FIRST LEVEL FACTORING........................... 11
Figure 1.4 -- SECOND LEVEL FACTORING........................... 12
Figure 1.5 -- SOFTWARE STRUCTURE.............................. 13
Figure 3.1 -- HIERARCHICAL DIAGRAM OF TRANSFORM SYSTEM... 22
Figure 3.2 -- PROCESS INPUT SUBSYSTEM.......................... 23
Figure 3.3 -- LOAD INPUT SUBSYSTEM............................. 25
Figure 3.4 -- PROCESS TRANSFORM SUBSYSTEM..................... 26
Figure 3.5 -- PROCESS AFFERENT SUBSYSTEM..................... 29
Figure 3.6 -- PROCESS EFFERENT SUBSYSTEM..................... 30
Figure 3.7 -- PRINT MODULES SUBSYSTEM........................ 32
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 gone 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 module 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).

![Diagram of process input subsystem]

**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 formatted 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 COMBI 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 reformatted 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.

![Diagram of LOAD_INPUT subsystem]

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 FA_SUP and FA_SUB
with other functions that position and match the two fields.
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.

![Diagram](image)

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 ADDIRAN.C, CHANIRAN.C, AFFIRA.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 (AFFIRA.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"

4. De Marco, T.
"Structured Analysis and System Specification"

5. Yourdon E., Constantine L.L.
"Structured Design"

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"

"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

"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:
k_master
module: expand
inputs:
k_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
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

<table>
<thead>
<tr>
<th>CHECKSEQ</th>
<th>AFFERENT</th>
<th>TRANSFORM</th>
<th>EFFERENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFF</td>
<td>format_output</td>
<td>EFF</td>
<td></td>
</tr>
<tr>
<td>expand</td>
<td>match</td>
<td>add_check</td>
<td></td>
</tr>
<tr>
<td>validate</td>
<td>update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AFF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reformat</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>edit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sequence</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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_keywordDlsrd>
<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_definition>
<sub_name> ::= {<sub_description>}
<type_keyword> ::= '** AFFECTIVE **' | '** EFFECTIVE **' | '** 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 1s(_ <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> 'AFFECTIVE' | '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", "combl", "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
Output 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
addtrans.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 chantrans.out
fi

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

```c
/* PROGRAM ADDTRAN.C */

#include <stdio.h>

char STRING2[50];

main()
{
    int i,n;
    file *rptr,*wrptr;
    rptr=fopen("DATAFLOW", "r");
    wrptr=fopen("ALLIN.OUT", "w");
    while (1)
    {
        i = 0;
        do
        {
            STRING2[++i] = 'o';
        } while (i < 50);
        fgets (STRING2,50,rptr);
        if (feof(rptr))
            break;
        if (strncmp(STRING2,"MODULE: ",7) != 0)
            fputs (STRING2,wrptr);
        else
            {if (STRING2[strlen(STRING2) - 3] == ' ')
             fputs (STRING2,wrptr);
             else
             {get_trans();
             fputs (STRING2,wrptr);
             }}
    }

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

    get_trans()
    {
        int i;
        char c, d;
        c = ' ';
        d = ' ';
        printf("\n");
        printf("THE FOLLOWING MODULE DOES NOT CONTAIN TRANSFORM INFO.");
        printf("\n");
        printf("%s\n", STRING2);
        printf("PLEASE ENTER DIFFERENT EFFERENT OR TRANSFORM (A, E OR T):");
        enter:
        c = getchar();
        if (c == '\n')
            return;
        d = getchar();
        if ((c == 'A') || (c == 'T') || (c == 'E'))
```
\begin{verbatim}
54     L = 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 }
\end{verbatim}
/* PROGRAM NAME : CHANTRAN.C */
#include <stdio.h>
char STRING2[50];
main()
{
    int i, n;
    char c, d;
    file *=Rptr, *WRptr;
    Rptr Femme "DATAFLOW", "R";
    WRptr Femme "ALLIN.OUT", "W"
;
    printf ("/5S\n", "THE SYSTEM WILL STEP THROUGH ALL THE MODULES.");
    printf ("/5S\n", "IF YOU WANT TO CHANGE THE TRANSFORM INFORMATION.");
    printf ("/5S\n", "WHEN THE MODULE IS DISPLAYED ENTER A ,E , OR T");
    printf ("/5S\n", "CARR. RETURN ACCORDINGLY.");
    while (1)
    {
        i = 0;
        do
            string2[++i] = '\0';
            while (i < 50);
        fgets (string2.50, Rptr);
        if (feof(Rptr))
            break;
        if (strcmp(string2, "MODULE: *,,7") != 0)
            fputs (STRING2, WRptr);
        else
            {
                printf ("\n\n" );
                printf("/5S\n", string2);
                printf ("PLEASE ENTER NEW INFO. A, E, T OR CARR. RETURN :");
            }
        c = getchar();
        if (c == '\n')
            {
                fputs (STRING2, WRptr);
                continue;
            }
        d = getchar();
        getTrans(c);
        fputs (STRING2, WRptr);
    }
    /* THIS FUNCTION ADD THE TRANSFORM INFORMATION TO THE MODULE NAME
    BY PROMPTING THE USER FOR IT */
    getTrans(d)
    char d;
    {
    }
    int i;
    char c;
    c = d;
    enter:
IF ((C == 'A') || (C == 'T') || (C == 'E'))
{
  IF (STRING2[STRLEN(STRING2) - 3] == ' ')
    I = STRLEN(STRING2) - 2;
  ELSE
    I = STRLEN(STRING2);
  STRING2[I - 1] = ' ';
  STRING2[I] = C;
  STRING2[++I] = '\n';
}
ELSE
{
  PRINTF("PLEASE ENTER A, E OR T:");
  C = GETCHAR();
  IF (C == '\n')
    RETURN;
  D = GETCHAR();
  GOTO ENTER;
}

Appendix K — AFFTRA listing

```c
/* PROGRAM NAME : AFFTRA.C */

#include <stdio.h>

char arr[50][40];
char err[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];

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[];
{
    int i, n, x;
    int a_count = 0;
    int e_count = 0;
    int t_count = 0;
    x = 0;
    i = 1;
    n = 0;
    rp = fopen("allin.in","r");
    wrp = fopen("array.out","w");

    while (i)
    {
        while ((arr[++n] = fgetc(rp)) != '\n')
        {
            if (feof(rp))
                break;
            continue;
        }
        if (feof(rp))
            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_TRANSFORM();
60      PRINTF ("\n");
61      FOR (I = 1; I < ARGV; I++)
62      PRINTF ("%4SS%5c", (ARGV[I]), (I < ARGV-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  }  LOADE (A)
81  INT A;
82  {
83      INT I;
84      E_COUNT = E_COUNT + 1;
85      I = 0;
86      WHILE (A >= ++I)
87      {
88          /* PUTCHAR ('E') */
89          EARR[E_COUNT][I] = ARR[I];
90      }  /* PRINTF ("%c", EARR[E_COUNT][I]); */
91      LLOADT (A)
92      INT A;
93  {
94      INT I;
95      T_COUNT = T_COUNT + 1;
96      I = 0;
97      WHILE (A >= ++I)
98      {
99          /* PUTCHAR ('T') */
100         TARR[T_COUNT][I] = ARR[I];
101      }  */
*/

INT MOD, F, G, I, K, N;

MOD = 0;
F = 0;

WHILE (++MOD <= T_COUNT)
{
    IF (T_COUNT == 1)
    {
        TARR[1][0] = 'B';
        BREAK;
    }
    T_LOAD_REC (MOD);
    N = MOD;
    F = 0;
    K = 0;
    /* PRINTF ("%s %d\n", "MOD1 = ", MOD); */
    WHILE ((N != (MOD - 1)) && (F != 1))
    {
        /* PRINTF ("%s %d\n", "MOD2 = ", MOD); */
        /* PRINTF ("%s %d\n", "N1 = ", N); */
        IF (N == T_COUNT)
            IF (MOD == 1)
                BREAK;
            ELSE
                N = 0;
        N = N + 1;
        /* PRINTF ("%s %d\n", "N2 = ", N); */
        K = T_POS_OUT (N);
        F = T_IN_OUT (N,K);
        /* PRINTF ("%s %d\n", "F = ", F); */
    }
    PUTCHAR ('N');
    N = MOD;
    G = 0;
    WHILE ((N != (MOD - 1)) && (G != 1))
    {
        IF (N == T_COUNT)
        {
            IF (MOD == 1)
                BREAK;
            ELSE
                N = 0;
N = N + 1;
I = T_POS_IN (N);
G = T_OUT_IN (N, I);

/* PRINTF ("%S \n", "G = ", G); */

IF (F == 1)
{
  IF (G == 1)
    /* I/O TRANSFORM */
    TARR[MOD][0] = 'T';
  ELSE
    /* I-TRANSF, O-EFF. */
    TARR[MOD][0] = 'O';

  ELSE
    /* I-AFF., O-TRANSF */
    TARR[MOD][0] = 'C';
  ELSE
    /* I-AFF., O-EFF. */
    TARR[MOD][0] = 'B';
}

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

T_LOAD_REC (N)
INT N:
{
  INT I, J, P;
  CHAR C;

  I = 0;
  J = 0;

  DO
    C = TARR[N][++I];
    WHILE (C != ' ', ' '): /POSITIONS I AFTER MOD. NAME*/
  DO
    HOLD_IN[+J] = TARR[N][+I];
    WHILE (HOLD_IN(J) != ' ', ' '): /LOADS INPUT */
  /* PRINTF ("%S \n", "HOLD_IN");
  P = 0;
  DO
    PRINTF ("%C", HOLD_IN[+P]);
    WHILE (P <= J): */
  J = 0;
  DO
    HOLD_OUT[+J] = TARR[N][+I];
    WHILE (HOLD_OUT(J) != ' ', ' '): /LOADS OUTPUT */
209        /* PRINTF ("%S\n", " HOLD_OUT ");
210        P = O;
211        DO
212            PRINTF ("%C", HOLD_OUT[++P]);
213            WHILE (P <= J); */
214    
215    /*THIS FUNCTION ACCESES 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 = O;
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    
236    /* FUNCTION TO MATCH INPUT TO ON
237    OF THE OUTPUTS.
238    POINTER IS SET AT START OF OUTPUT FROM
239    T_POS_OUT FUNCTION */
240    T_IN_OUT (N,L)
241    INT N,L:
242    
243    { INT I, J, C;
244        /* PRINTF ("%S\n", " ENTERED T_IN_OUT"); */
245        I = O;
246        C = O;
247        J = N;
248        WHILE (C == O)
249        {
250            C = STRCMP (HOLD_IN[++I],TARR[j][++L]);
251            /* PRINTF ("%C %S\n", HOLD_IN[I], TARR[j][L]); */
252            IF ((HOLD_IN[I] == ",") && (TARR[j][L] == ","))
253                RETURN (T); /* MATCH FOUND */
254     RETURN (O); /* NO MATCH FOUND */
255 }

256 /* THIS FUNCTION ACCESSES THE NEXT
257 TRANSFORM RECORD AND PLACES THE
258 T_POS_IN (N)
259 INT N:
260 {
261     INT I, J;
262     CHAR C;
263     /* PRINTF ("%S\n", " ENTERED T_POS_IN"); */
264     I = O;
265     J = N;
266     DO
267         (C = TARR[J][++I]);
268         WHILE (C != ',');
269     RETURN (I);
270 }

272 /* THIS FUNCTION LOOKS FOR A MATCH FOR THE OUTPUT
273 OF THE MODULE BEING REFERENCED WITH 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 = O;
281     C = O;
282     WHILE (C == O)
283         { C = STRCMP(HOLD_OUT[++I], TARR[J][++L]);
284             /* PRINTF ("%C %C\n", HOLD_OUT[I], TARR[J][L]); */
285             IF ((HOLD_OUT[I] == ',') & (TARR[J][L] == ',')
286                 RETURN (I); /*MATCH FOUND */
287         }
288     RETURN (O); /* NO MATCH FOUND */
289 }

291 /*********************************************************************************/
/* THIS FUNCTION AND ITS SUBORDINATE FUNCTIONS ARE THE MAIN PROCESS OF THE AFFERENT BRANCH */

INT ORD_COUNT, MOD_NBR;
INT SUB_COUNT, T_A_COUNT;
CHAR MOLD_MOD[40];

AFF_P ()
{
    INT I, J;
    INT R;
    I = -1;
    J = -1;
    MOD_NBR = 0;
    ORD_COUNT = 1;
    T_A_COUNT = 0;
    SUB_COUNT = 1;

    WHILE (++J <= 50)
    {
        WHILE ( ++I <= 4)
            MOD_ORD_TABLE [J][I] = 0;
        I = -1;
    }

    WHILE (T_A_COUNT <= T_COUNT)
    {
        R = GET_TRANS (T_A_COUNT);
        /* THIS MODULE GETS A TRANSF. INPUT */
        IF (R == 0)
            BREAK;
        SUB_COUNT = ORD_COUNT;
        MOD_ORD_TABLE [ORD_COUNT][1] = 99;
        MOD_ORD_TABLE [ORD_COUNT][2] = 99;
        MOD_ORD_TABLE [ORD_COUNT][3] = 99;
        SUB_COUNT = SUB_COUNT + 1;
        ORD_COUNT = ORD_COUNT + 1;

        J = 0;
        WHILE (++J <= A_COUNT)
            AARR [J][0] = 7;

        FIND_1_MOD (); /* FUNCTION TO FIND FIRST MODULE */
        WHILE ((MOD_ORD_TABLE [ORD_COUNT][1] != 0) ||
              (MOD_ORD_TABLE [SUB_COUNT][3] != 0))
        {
            IF (MOD_ORD_TABLE [ORD_COUNT][1] == 0)
                
            
    
}
MOD_ORD_TABLE [ORD_COUNT][1] = MOD_ORD_TABLE [SUB_COUNT][3];
SUB_COUNT = SUB_COUNT + 1;
}

/*************** FOR TEST ONLY **************/
/*
IF (ORD_COUNT > 10)
BREAK;
*/
/*************** END ***************

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

AARR [MOD_ORD_TABLE[ORD_COUNT][1]][0] = '*';
A_LOAD_ALL (MOD_ORD_TABLE [ORD_COUNT][1]);
F_A_SUBO (); /* THIS FUNCTION FINDS SUBORDINATE MODULE */
F_A_SUPE (); /* THIS FUNCTION FINDS SUPERORDINATE MODULE */
FIND_A_MOD (); /* THE ABOVE MODULE FINDS ANY MODULES WITH THE SAME NAME */
ORD_COUNT = ORD_COUNT + 1;
}

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

/* THIS FUNCTIONS OBTAINS THE INPUT FROM ONE OF THE
TRANSFORM MODULES MARKED #B# OR #C# AND LOADS IT
INTO HOLD_IN */
GET_TRANS ()
{
CHAR C;
INT I, J;
I = 0;
J = 0;

WHILE (++T_A_COUNT <= T_COUNT)
{
IF ((TARR [T_A_COUNT][0] != 'B') &
    (TARR [T_A_COUNT][0] != 'C'))
    CONTINUE;
ELSE
{
    DO
    (C = TARR[T_A_COUNT][++J]);
    WHILE (C != ' '); 
    DO
    (HOLD_IN[++I] = TARR[T_A_COUNT][++J]);
    /* PRINTF ("%c", HOLD_IN[I]); */
384 }  
385          } while (HOLD_IN[I] != ' ', '');
386          return (1);
387      }
388  
389  
390  /* THIS MODULE FINDS THE FIRST AFFECTED 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_subo ()
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;
K = A_POS_OUT(N);
F = A_IN_OUT(N,K);
}

IF (F == 1)
MOD_ORD_TABLE[ORD_COUNT][3] = N;
ELSE
MOD_ORD_TABLE[ORD_COUNT][3] = 0;

/* THIS FUNCTION FINDS THE SUPERORDINATE OF A
MODULE */
F_A_SUPE()
{
INT N,L,H;
N = MOD_NBR;
H = O;

WHILE ((N != (MOD_NBR - 1)) && (H != 1))
{
IF (N == A_COUNT)
{
IF (MOD_NBR == 1)
BREAK;
ELSE
N = O;
}
N = N + 1;
L = A_POS_IN(N);
H = A_OUT_IN(N,L);
}
IF (H == 1)
MOD_ORD_TABLE[ORD_COUNT][2] = N;
ELSE
MOD_ORD_TABLE[ORD_COUNT][2] = 99;

/* THIS FUNCTION LOCATES ANY OTHER MODULE WITH THE
SAME NAME AS THE CURRENT ONE */
FIND_A_MOD()
{
INT N,G;
N = MOD_NBR;

WHILE ((N != (MOD_NBR - 1)) && (G != 1))
{
IF (N == A_COUNT)
{
IF (MOD_NBR == 1)
BREAK;
ELSE
   N = 0;
}
N = N + 1;
G = A_MOD_MOD(N);
}
IF (G == 1)
{
   MODORD_TABLE[ORD_COUNT + 1][1] = N;
}

/* THIS FUNCTION LOADS MODULE NAME INPUT AND OUTPUT INTO
   HOLD_MOD, HOLD_IN, HOLD_OUT */
A_LOAD_ALL(N)
INT N:
{
   INT I, J:
   CHAR C:
   I = 0;
   J = 0:

   DO
      HOLD_MOD[++J] = AARR[N][++I];
      WHILE (HOLD_MOD[J] != ',');
   J = 0;
   DO
      HOLD_IN[++J] = AARR[N][++I];
      WHILE (HOLD_IN[J] != ',');
   J = 0;
   DO
      HOLD_OUT[++J] = AARR[N][++I];
      WHILE (HOLD_OUT[J] != ',');
   }

/* THIS FUNCTION COMPARSES TWO MODULES NAME TO
   FIND OUT IF THEY MATCH */
A_MOD_MOD(N)
INT N:
{
   INT I, L, J, C:
   /* PRINTF ("%s\n", "ENTERED A_MOD_MOD"); */
   I = 0;
   L = 0;
   C = 0;
   J = N;
   WHILE (C == 0)
{ 
    C = STRCMO (HOLD_MOD[++I], AARR[J][++L]);
    /* PRINTF ("%C %C\n", HOLD_MOD[I], AARR[J][L]); */
    IF ((HOLD_MOD[I] == ', ') && (AARR[J][L] == ', '))
    {
        IF (AARR[J][O] == ' *')
            RETURN (O);
        ELSE
            RETURN (1);
    }
    RETURN (0); /* NO MATCH FOUND */
}

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

A_POS_IN (N)
INT N:
{
    INT I, J:
    CHAR C:
    /* PRINTF ("%S\n", " ENTERED A_POS_IN"); */
    I = O;
    J = N;
    DO
        (C = AARR[J][++I]);
        WHILE (C != ', ');
    RETURN (I);
}

/* THIS FUNCTION COMPARES THE INPUT OF ONE MODULE
WITH THE OUTPUT OF ANOTHER TO FIND A MATCH */
A_IN_OUT (N,L)
INT N,L:
{
    INT I, J, C:
    /* PRINTF ("%S\n", " ENTERED A_IN_OUT"); */
    I = O;
    C = O;
    J = N;
    WHILE (C == O)
    {    

C = STRCMP (HOLD_IN[++I], AARR[J][++]L);

/* PRINTF ("%C %C\n", HOLD_IN[I], AARR[J][L]); */

IF ((HOLD_IN[I] == ' ') && (AARR[J][L] == ' '))

RETURN (I); /* MATCH FOUND */

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

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

A_POS_OUT (N)
INT N:
{
    INT COMMA_COUNT, J, L;
    CHAR C;

    /* PRINTF ("%s\n", " ENTERED A_PO_OUT"); */
    COMMA_COUNT = 0;
    J = N;
    L = O;

    WHILE (COMMA_COUNT < 2)
    {
        C = AARR[J][++]L;
        IF (C == ' ')
            COMMA_COUNT = COMMA_COUNT + 1;
    }

    RETURN (L);

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

A_OUT_IN (J, L)
INT J, L:
{
    INT C, I:

    /* PRINTF ("%s\n", " ENTERED A_OUT_IN"); */
    I = O;
    C = O;

    WHILE (C == O)
    {
        C = STRCMP (HOLD_OUT[++I], AARR[J][++]L);

        /* PRINTF ("%C %C\n", HOLD_OUT[I], AARR[J][L]); */

        IF ((HOLD_OUT[I] == ' ') && (AARR[J][L] == ' '))
            RETURN (I); /* MATCH FOUND */
RETURN (0); /* NO MATCH FOUND */
}

PRINT_AFF ()
{
    IN_COUNT = 1;
    OUT_COUNT = 0;
    SUP_COUNT = 0;
    SUB_COUNT = 0;

    PRINTF ("\n");
    PRINTF ("** AFFECTED MODULES ** \n\n");
    WHILE (1)
    {
        TAB_COUNT = TAB_COUNT + 1;
        IF (MOD_ORD_TABLE[TAB_COUNT][1] == 0)
            BREAK;
        IF (MOD ORD_TABLE[TAB COUNT][0] == 100)
            CONTINUE;
        IF (MOD ORD_TABLE[TAB COUNT][1] == 99)
            CONTINUE;
        A_P_LOAD ();
        LIST_COUNT = TAB_COUNT + 1;
        WHILE (MOD_ORD_TABLE [LIST_COUNT][1] != 99)
        {
            IF (MOD ORD_TABLE [LIST_COUNT][1] == 0)
                BREAK;
            IF (MOD ORD_TABLE [LIST_COUNT][0] == 100)
            {
                LIST_COUNT = LIST_COUNT + 1;
                CONTINUE;
            }
            A_P_MATCH ();
            LIST_COUNT = LIST_COUNT + 1;
        }
        IF (MOD ORD_TABLE [LIST_COUNT][1] == 0)
            BREAK;
        A_P_PRINT ();
        IF (MOD_ORD_TABLE [TAB_COUNT + 1][1] == 99)
            CONTINUE;
    }
    /* THIS FUNCTION LOADS THE FIRST RECORD ON THE TABLE */
A_P_LOAD() 
{
    INT I,J;
    I = 0;
    J = 0;
    WHILE (H_MODULE[I] != '.')
    {
        H_MODULE[++I] = AARR[MOD_ORD_TABLE[TAB_COUNT][1]][++J];
    }
    I = 0;
    IN_COUNT = 1;
    WHILE (H_INPUT[IN_COUNT][I] != ' ')
    {
        H_INPUT[IN_COUNT][++I] = AARR[MOD_ORD_TABLE[TAB_COUNT][1]][++J];
    }
    I = 0;
    OUT_COUNT = 1;
    WHILE (H_OUTPUT[OUT_COUNT][I] != ' ')
    {
        H_OUTPUT[OUT_COUNT][++I] = AARR[MOD_ORD_TABLE[TAB_COUNT][1]][++J];
    }
    I = 0;
    J = 0;
    SUB_COUNT = 1;
    IF (MOD_ORD_TABLE[TAB_COUNT][3] == 0)
    {
        H_SUB[SUB_COUNT][I] = ' '; 
    }
    ELSE
    {
        H_SUB[SUB_COUNT][I] = AARR[MOD_ORD_TABLE[TAB_COUNT][3]][++J];
    }
    SUB_COUNT = 1;
    IF (MOD_ORD_TABLE[TAB_COUNT][2] == 99)
    {
        H_SUPER[SUB_COUNT][1] = 'M';
        H_SUPER[SUB_COUNT][2] = 'A';
        H_SUPER[SUB_COUNT][3] = 'i';
        H_SUPER[SUB_COUNT][4] = 'N';
        H_SUPER[SUB_COUNT][5] = 'I';
    }
    ELSE
    {
        H_SUPER[SUB_COUNT][1] = ' '; 
    }
    I = 0;
    J = 0;
    WHILE (H_SUPER[I][J] != ' ')
    {
        H_SUPER[I][++J] = AARR[MOD_ORD_TABLE[TAB_COUNT][2]][++J];
    }
}

/* THIS FUNCTION MATCHES THE MODULES WITH SAME NAME 
STORED IN THE TABLE AND GETS THEIR INPUTS AND 
OUTPUTS IF THEY ARE DIFFERENT THAT THE OTHER ONES 
FOR THE SAME MODULE */
A_P_MATCH()
{ 
    P = LIST_COUNT - 1;
    WHILE (MOD_ORD_TABLE[++P][1] != 99) 
    { 
        IF (MOD_ORD_TABLE[P][1] == 0) 
            BREAK;
        C = 0;
        I = 0;
        S = 0;
        WHILE (C == 0) 
        { 
            C = STRCMP(H_MODULE[++S], AARR[MOD_ORD_TABLE[P][1]][++I]);
            IF ((H_MODULE[I] == ',' || AARR[MOD_ORD_TABLE[P][1]][I] == ',') 
                BREAK;
        } 
        IF (C != 0) 
            CONTINUE;
    } 

    MOD_ORD_TABLE[P][0] = 100;

    /* THE MODULE WAS FOUND AND NOW THE INPUT, OUTPUT, 
        SUB AND SUP WILL BE MATCHED */
    K = 0;
    F = IN_COUNT;
    WHILE (++K <= F) 
    { 
        J = I;
        C = 0;
        L = 0;
        WHILE (C == 0) 
        { 
            C = STRCMP (H_INPUT[K][++L], AARR[MOD_ORD_TABLE[P][1]][++J]);
            IF ((H_INPUT[K][L] == ',' || AARR[MOD_ORD_TABLE[P][1]][J] == ',') 
                BREAK;
        } 
        IF (C == 0) 
            BREAK;
        IF (((C != 0) && (K != F)) 
            CONTINUE;
    } 

    IN_COUNT = IN_COUNT + 1;
    N = 0;
    WHILE (H_INPUT[IN_COUNT][N] != ',') 
    { 
        H_INPUT[IN_COUNT][++N] = AARR[MOD_ORD_TABLE[P][1]][++I];
        J = I;
    } 
    M = J;

    /* THIS IS TO MATCH THE OUTPUT */
    K = 0;
F = OUT_COUNT;

WHILE (++k <= F)
{
    R = N;
    C = O;
    L = O;
    WHILE (C == O)
    {
        C = STRCMP (H_OUTPUT[K][++]L, AARR[MOD_ORD_TABLE[P][1]][++]R);
        IF ((H_OUTPUT[K][L] == ' ') && (AARR[MOD_ORD_TABLE[P][1]][R] == ' '))
            BREAK;
    }

    IF (C == O)
        BREAK;

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

    OUT_COUNT = OUT_COUNT + 1;
    N = O;
    WHILE (H_OUTPUT[OUT_COUNT][N] != ' ')
        H_OUTPUT[OUT_COUNT][++]N = AARR[MOD_ORD_TABLE[P][1]][++]M;
}

B = SUP_COUNT;

K = O;

WHILE (++k <= B)
{
    I = O;
    C = O;
    L = O;
    IF (MOD_ORD_TABLE[P][2] == 99)
        BREAK;

    WHILE (C == O)
    {
        C = STRCMP (H_SUPER[K][++]L, AARR[MOD_ORD_TABLE[P][2]][++]I);
        IF ((H_SUPER[K][L] == ' ') && (AARR[MOD_ORD_TABLE[P][2]][I] == ' '))
            BREAK;
    }

    IF (C == O)
        BREAK;

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

    L = O;
    I = O;
    SUP_COUNT = SUP_COUNT + 1;
    WHILE (H_SUPER[SUP_COUNT][L] != ' ')
        H_SUPER[SUP_COUNT][++]L = AARR[MOD_ORD_TABLE[P][2]][++]I;
}

B = SUB_COUNT;

K = O;
WHILE (++K <= B) {
  L = 0;
  I = 0;
  C = 0;
  IF (MOD_ORD_TABLE[P][3] == 0) BREAK:
  WHILE (C == 0) {
    C = STRCMP(H_SUB[K][+L], AARR[MOD_ORD_TABLE[P][3][++I]]);
    IF ((H_SUB[K][L] == ',') && (AARR[MOD_ORD_TABLE[P][3][I] == ','))) {
      BREAK;
    }
  }
  IF (C != 0) {
    BREAK;
  }
  IF ((C != 0) && (K != B)) {
    CONTINUE;
  }
  L = 0;
  I = 0;
  SUB_COUNT = SUB_COUNT + 1;
  WHILE (H_SUB[SUB_COUNT][L] != ',') { H_SUB[SUB_COUNT][++L] = AARR[MOD_ORD_TABLE[P][3][+I]]; }
}

/* THIS FUNCTION PRINTS THE MODULES IN FORMAT 1 */
A_P_PRINT() {
  INT I, K;
  I = 0;
  PRINTF("%S", " MODULE NAME: ");
  WHILE (H_MODULE[I + 1] != ',') {
    PRINTF("%C", H_MODULE[++I]);
    PRINTF("\n");
  }
  K = 0;
  PRINTF("%S", " INPUT(S): ");
  WHILE (++K <= IN_COUNT) {
    I = 0;
    WHILE (H_INPUT[K][I + 1] != ',') {
      PRINTF("%C", H_INPUT[K][++I]);
      PRINTF("\n");
    }
    IF (K != IN_COUNT) {
      PRINTF("%S", " ");
    }
  }
  K = 0;
  PRINTF("%S", " OUTPUT(S): ");
while (++k <= out_count)
{
    i = 0;
    while (h_output[k][i + 1] != ',')
    {
        printf("%c", h_output[k][i++]);
        printf("\n");
        if (k != out_count)
            printf("%s", "");
    }
    k = 0;
    printf("%s", "superordinate(s): ");
    while (++k <= sup_count)
    {
        i = 0;
        while (h_super[k][i + 1] != ',')
        {
            printf("%c", h_super[k][i++]);
            printf("\n");
            if (k != sup_count)
                printf("%s", "");
        }
        k = 0;
        printf("%s", "subordinate(s): ");
        while (++k <= sub_count)
        {
            i = 0;
            while (h_sub[k][i + 1] != ',')
            {
                printf("%c", h_sub[k][i++]);
                printf("\n");
                if (k != sub_count)
                    printf("%s", "");
            }
            printf("\n");
        }
    }
    printf("\n");
/
*/
*/

```

```
I = -1;
}

WHILE (T_E_COUNT <= T_COUNT)
{
    R = GET_E_TRANS(T_E_COUNT);
    /* THIS MODULE GETS A TRANSF. INPUT */
    IF (R == O)
        BREAK;

    SUB_COUNT = ORD_COUNT;
    E_MOD_ORD_TABLE [ORD_COUNT][1] = 99;
    E_MOD_ORD_TABLE [ORD_COUNT][2] = 99;
    SUB_COUNT = SUB_COUNT + 1;
    ORD_COUNT = ORD_COUNT + 1;

    J = 0;
    WHILE (++J <= E_COUNT)
        EARR [J][0] = " ";

    FIND_FIRST_E_MOD ().; /* FUNCTION TO FIND FIRST MODULE */
    WHILE ((E_MOD_ORD_TABLE [ORD_COUNT][1] != O) ||
           (E_MOD_ORD_TABLE [SUB_COUNT][3] != O))
        {

        IF (E_MOD_ORD_TABLE [ORD_COUNT][1] == O)
            {
                E_MOD_ORD_TABLE [ORD_COUNT][1] = E_MOD_ORD_TABLE [SUB_COUNT][3];
                SUB_COUNT = SUB_COUNT + 1;
            }

        /*************** FOR TEST ONLY ***************
        /*
        IF (ORD_COUNT > 10)
            BREAK;
        */
        /*************** END ***************

        MOD_NBR = E_MOD_ORD_TABLE [ORD_COUNT][1];
        IF (MOD_NBR == O)
            BREAK;

        EARR [E_MOD_ORD_TABLE[ORD_COUNT][1]][0] = '*';
        E_LOAD_ALL (E_MOD_ORD_TABLE [ORD_COUNT][1]);
        F_E_SUBO (); /* THIS FUNCTION FINDS SUBORDINATE MODULE */
        F_E_SUPER (); /* THIS FUNCTION FINDS SUPERORDINATE MODULE */
        FIND_E_MOD ();
        /* THE ABOVE MODULE FINDS ANY MODULES WITH THE SAME NAME */
        ORD_COUNT = ORD_COUNT + 1;
        }

    IF (E_MOD_ORD_TABLE [ORD_COUNT][1] == O)
SUB_COUNT = SUB_COUNT + 1;
}

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

GET_E_TRANS ()
{
CHAR C;
INT I, J, COMME_COUNT;
I = 0;
J = 0;
COMME_COUNT = 0;

WHILE (++T_E_COUNT <= T_COUNT)
{
IF ((TARR [T_E_COUNT][0] != 'B') &&
(TARR [T_E_COUNT][0] != 'D'))

CONTINUE;
ELSE

WHILE (COMME_COUNT < 2)
{
C = (TARR [T_E_COUNT][++J]):
IF (C == ',')

COMME_COUNT = COMME_COUNT + 1;
}

DO

HOLD_OUT[++I] = TARR[T_E_COUNT][++J]:

/* PRINTF ("%C", HOLD_OUT[I]): */

WHILE (HOLD_OUT[I] != ',');
RETURN (1);

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

FIND_FIRST_E_MOD ()
{
INT N, J, P;
N = MOD_NBR;

WHILE ((N != (MOD_NBR -1)) && (P != 1))
{
IF (N == E_COUNT)

IF (MOD_NBR == 1)

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_SUPO ()
1003    {
1004        INT N,K,F;
1005        N = MOD_NBR;
1006    WHILE ((N != (MOD_NBR - 1)) && (F != 1))
1007    {
1008        IF (N == E_COUNT)
1009        {
1010            IF (MOD_NBR == 1)
1011                BREAK;
1012            ELSE
1013                N = 0;
1014        }
1015        N = N + 1;
1016        K = E_POS_IN (N);
1017        F = E_OUT_IN (N,K);
1018    }
1019    IF (F == 1)
1020        E_MOD_ORD_TABLE [ORD_COUNT][3] = N;
1021    ELSE
1022        E_MOD_ORD_TABLE [ORD_COUNT][3] = 0;
1023    }
1024    /* THIS FUNCTION FINDS THE SUPERORDINATE OF A
1025 MODULE */
1026    F_E_SUPE ()
1027    {
1028        INT N,L,H;
1029        N = MOD_NBR;
1030        H = 0;
1032    WHILE ((N != (MOD_NBR - 1)) && (H != 1))
1033    {
1034       IF (N == E_COUNT)
1035       {
1036          IF (MOD_NBR == 1)
1037          BREAK;
1038       }
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    WHILE ((N != (MOD_NBR - 1)) && (G != 1))
1059    {
1060       IF (N == E_COUNT)
1061       {
1062          IF (MOD_NBR == 1)
1063          BREAK;
1064       }
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;
I = 0;
J = 0;

/* PRINTF ("%s\n", "ENTERED E_LOAD_ALL"); */

DO
HOLD_MOD[++J] = EARR[N][++I];
WHILE (HOLD_MOD[J] != ',');
J = 0;
DO
HOLD_IN[++J] = EARR[N][++I];
WHILE (HOLD_IN[J] != ',');
J = 0;
DO
HOLD_OUT[++J] = EARR[N][++I];
WHILE (HOLD_OUT[J] != ',');

}

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

E_MOD_MOD (N)
INT N:

{   

   INT I, L, J, C;

   /* PRINTF ("%s\n", "ENTERED E_MOD_MOD"); */

   I = 0;
   L = 0;
   C = 0;
   J = N;

   WHILE (C == 0)
   {
      C = STRCMP (HOLD_MOD[+I], EARR[J][+L]);
      /* PRINTF ("%c %c\n", HOLD_MOD[I], EARR[J][L]); */
      IF ((HOLD_MOD[I] == ',') && (EARR[J][L] == ','))
      {
         IF (EARR[J][O] == '*')
            RETURN (O);
         ELSE
            RETURN (1);
      }
   }
   RETURN (O); /* NO MATCH FOUND */

}   

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

E_PDS_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 = O;
1147    J = N:
1148    WHILE (C == O)
1149    {
1150        C = STRCMP (HOLD_IN[++I],EARR[J][++L]);
1151    /* PRINTF ("%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 A MATCH */
1177  E_OUT_IN (J, L)
1178  INT J, L;
1179  {
1180      INT C, I;
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 %S\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 ("\n\n*");
1202      PRINTF ( "** EFFERENT MODULES ** \n\n*");
1203      WHILE (1)
1204      {  
1205          TAB_COUNT = TAB_COUNT + 1;
IF (E_MOD_ORD_TABLE[TAB_COUNT][1] == 0)  
BREAK;
IF (E_MOD_ORD_TABLE[TAB_COUNT][0] == 100)  
CONTINUE;
IF (E_MOD_ORD_TABLE[TAB_COUNT][1] == 99)  
CONTINUE:
E_P_LOAD ();
LIST_COUNT = TAB_COUNT + 1;

WHILE (E_MOD_ORD_TABLE [LIST_COUNT][1] != 99)  
{
  IF (E_MOD_ORD_TABLE [LIST_COUNT][1] == 0)  
    BREAK;
  IF (E_MOD_ORD_TABLE [LIST_COUNT][0] == 100)  
    LIST_COUNT = LIST_COUNT + 1;
    CONTINUE:
}
E_P_MATCH ();
LIST_COUNT = LIST_COUNT + 1;
}
IF (E_MOD_ORD_TABLE [LIST_COUNT][1] == 0)  
BREAK;
E_P_PRINT ();
IF (E_MOD_ORD_TABLE [TAB_COUNT + 1][1] == 99)  
CONTINUE:
/
/* THIS FUNCTION LOADS THE FIRST RECORD ON 
THE TABLE */
E_P_LOAD ()
{
INT I,J;
I = 0;
J = 0;
WHILE (H_MODULE [I] != ' ')
{
H_MODULE [++I] = EARR [E_MOD_ORD_TABLE[TAB_COUNT][1]][++J];
}
I = 0;
IN_COUNT = 1;
WHILE (H_INPUT [IN_COUNT][I] != ' ')
H_INPUT[IN_COUNT][++I] = EARR [E_MOD_ORD_TABLE[TAB_COUNT][1]][++J];
I = 0;
OUT_COUNT = 1;
WHILE (H_OUTPUT [OUT_COUNT][I] != ',')
H_OUTPUT [OUT_COUNT][++I] = ERR[E_MOD_ORD_TABLE[TAB_COUNT][I]][++J]:
I = O;
J = O;
SUB_COUNT = 1;
IF (E_MOD_ORD_TABLE[TAB_COUNT][3] == O)
H_SUB[SUB_COUNT][1] = '.';
ELSE
WHILE (H_SUB [SUB_COUNT][I] != ',')
H_SUB [SUB_COUNT][++I] = ERR[E_MOD_ORD_TABLE[TAB_COUNT][3]][++J]:
SUB_COUNT = 1;
IF (E_MOD_ORD_TABLE [TAB_COUNT][2] == 99)
{
H_SUPER [SUP_COUNT][1] = 'M';
H_SUPER [SUP_COUNT][2] = 'A';
H_SUPER [SUP_COUNT][3] = 'I';
H_SUPER [SUP_COUNT][4] = 'N';
H_SUPER [SUP_COUNT][5] = ' ';
}
ELSE
{
I = O;
J = O;
WHILE (H_SUPER [1][I] != ',')
H_SUPER [1][++I] = ERR[E_MOD_ORD_TABLE[TAB_COUNT][2]][++J]:
}

/* THIS FUNCTION MATCHES THE MODULES WITH SAME NAME
STORED IN THE TABLE AND GETS THEIR INPUTS AND
OUTPUTS IF THEY ARE DIFFERENT THAN THE OTHER ONES
FOR THE SAME MODULE */
E_P_MATCH ()
{
P = LIST_COUNT - 1:
WHILE (E_MOD_ORD_TABLE [++]P[1] != 99)
{
IF (E_MOD_ORD_TABLE[P][1] == O)
BREAK;
C = O;
I = O;
S = O;
WHILE (C == O)
{
C = STRCMP(H_MODULE [++]S), ERR[E_MOD_ORD_TABLE[P][1]][++I]);
IF ((H_MODULE [I] == ',') & & (ERR[E_MOD_ORD_TABLE[P][1]][I] == ','))
BREAK;
}
IF (C != O)
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][I]][++J]);
1314           IF ((H_INPUT [K][L] == ',')) & & (EARR[E_MOD_ORD_TABLE[P][I]][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][I]][++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][I]][++R]);
1339           IF ((H_OUTPUT[K][L] == ',') & & (EARR[E_MOD_ORD_TABLE[P][I]][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] != ',')
    H_OUTPUT[OUT_COUNT][++N] = ERR[E_MOD_ORD_TABLE[P][1]][++M];
}

    B = SUP_COUNT;
    K = 0;

    while (++K <= B)
    {
        I = 0;
        C = 0;
        L = 0;
        if (E_MOD ORD_TABLE [P][2] == 99)
            break;

        while (C == 0)
        {
            C = STRCMP(H_SUPER[K][++L],ERR[E_MOD ORD_TABLE[P][2]][++I]);
            if ((H_SUPER[K][L] == ' ') && (ERR[E_MOD ORD_TABLE[P][2]][I] == ' '))
                break;
        }

        if (C == 0)
            break;

        if ((C != 0) && (K != B))
            continue;

        L = 0;
        I = 0;
        SUP_COUNT = SUP_COUNT + 1;
        while (H_SUPER[SUP_COUNT][L] != ' ')
            H_SUPER[SUP_COUNT][++L] = EARR[E_MOD ORD_TABLE[P][2]][++I];
    }

    B = SUB_COUNT;
    K = 0;

    while (++K <= B)
    {
        I = 0;
        C = 0;
        if (E_MOD ORD_TABLE [P][3] == 0)
            break;

        while (C == 0)
        {
            C = STRCMP(H_SUB[K][++L],ERR[E_MOD ORD_TABLE[P][3]][++I]);
            if ((H_SUB[K][L] == ' ') && (ERR[E_MOD ORD_TABLE[P][3]][I] == ' '))
                break;
        }

        if (C == 0)
            break;

        if ((C != 0) && (K != B))
            continue;

        L = 0;
        I = 0;
SUB_COUNT = SUB_COUNT + 1;

while (H_SUB[SUB_COUNT][L] != ' , ')
    H_SUB[SUB_COUNT][++L] = EARR[E_MOD_ORD_TABLE[P][3]][++I];

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

E_P_PRINT()
{
    int I, K;
    I = 0;
    printf("%s", " MODULE NAME: ");
    while (H_MODULE[I + 1] != ' , ')
        printf("%c", H_MODULE[++I]);
    printf("\n ");
    K = 0;
    printf("%s", " INPUT(S). ");
    while (++K <= IN_COUNT)
    {
        I = 0;
        while (H_INPUT[K][I + 1] != ' , ')
            printf("%c", H_INPUT[K][++I]);
        printf("\n ");
        if (K != IN_COUNT)
            printf("%s", " ");
    }
    K = 0;
    printf("%s", " OUTPUT(S): ");
    while (++K <= OUT_COUNT)
    {
        I = 0;
        while (H_OUTPUT[K][I + 1] != ' , ')
            printf("%c", H_OUTPUT[K][++I]);
        printf("\n ");
        if (K != OUT_COUNT)
            printf("%s", " ");
    }
    K = 0;
    printf("%s", " SUPERORDINATE(S): ");
    while (++K <= SUP_COUNT)
    {
        I = 0;
        while (H_SUPER[K][I + 1] != ' , ')
            printf("%c", H_SUPER[K][++I]);
        printf("\n ");
        if (K != SUP_COUNT)
            printf("%s", " ");
    }
K = 0;
PRINTF("%s", "SUBORDINATE(S): ");
WHILE (++K <= SUB_COUNT)
{
    I = 0;
    WHILE (H_SUB[K][I + 1] != ',')
    PRINTF ("%c", H_SUB[K][I++]);
    PRINTF ("\n");
    IF (K != SUB_COUNT)
    PRINTF ("%s", " ");
}
PRINTF ("\n");

/***************************************************************************/
/***************************************************************************/

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

PRINT_TRANSF()
{

TAB_COUNT = 0;
LIST_COUNT = 0;
IN_COUNT = 0;
OUT_COUNT = 0;

PRINTF ("\n");
PRINTF (" ** TRANSFORM MODULES ** \n\n");
WHILE (TAB_COUNT < T_COUNT)
{
    TAB_COUNT = TAB_COUNT + 1;
    IF (TARR[TAB_COUNT][0] == 'U')
    CONTINUE;
    T_P_LOAD();
    LIST_COUNT = TAB_COUNT;
    T_P_MATCH();
    T_P_PRINT();
}

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

T_P_LOAD()
{
    INT I,J;
I = 0;
J = 0;

WHILE (H_MODULE[I] != '.')
H_MODULE[++I] = TARR[TAB_COUNT][++J];

I = 0;
IN_COUNT = 1;
WHILE (H_INPUT[IN_COUNT][I] != '.')
H_INPUT[IN_COUNT][++I] = TARR[TAB_COUNT][++J];

I = 0;
OUT_COUNT = 1;
WHILE (H_OUTPUT[OUT_COUNT][I] != '.
H_OUTPUT[OUT_COUNT][++I] = TARR[TAB_COUNT][++J];

}
T_P_MATCH();


P = LIST_COUNT - 1;

WHILE (++LIST_COUNT <= T_COUNT)
{
C = 0;
I = 0;
S = 0;
WHILE (C == 0)
{
C = STRCMP(H_MODULE[++S], TARR[LIST_COUNT][++I]);
IF ((H_MODULE[I] == '.',) & & (TARR[LIST_COUNT][I] == '.
BREAK;

IF (C != 0)
CONTINUE;
TARR[LIST_COUNT][0] = 'U';

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

K = 0;
F = IN_COUNT;
WHILE (++K <= F)
{
J = I;
C = 0;
L = 0;
WHILE (C == 0)
{
C = STRCMP (H_INPUT[K][++L], TARR[LIST_COUNT][++J]);
IF ((H_INPUT[K][L] == '"') & (TARR[List_COUNT][J] == '"'))
    BREAK;
}
IF (C == O)
    BREAK;
IF ((C != O) & (K != F))
    CONTINUE;

IN_COUNT = IN_COUNT + 1;
N = O;
WHILE (H_INPUT[IN_COUNT][N] != '"')
    H_INPUT[IN_COUNT][++N] = TARR[List_COUNT][++I];
    J = I;
    M = J;
/* THIS IS TO MATCH THE OUTPUT */

K = O;
F = OUT_COUNT;
WHILE (++K <= F)
{
    R = M;
    C = O;
    L = O;
    //
    WHILE (C == O)
    {
        C = STRCMP (H_OUTPUT[K][++L], TARR[List_COUNT][++R]);
        IF ((H_OUTPUT[K][L] == '"') & (TARR[List_COUNT][R] == '"'))
            BREAK;
    }
}
IF (C == O)
    BREAK;
IF ((C != 'O') & (K != F))
    CONTINUE;

OUT_COUNT = OUT_COUNT + 1;
N = O;
WHILE (H_OUTPUT[OUT_COUNT][N] != '"')
    H_OUTPUT[OUT_COUNT][++N] = TARR[List_COUNT][++M];
}
}
T_P_PRINT()

{
    //
    INT I, K;
    I = O;
    PRINTF("%s", " MODULE NAME: ");
    WHILE (H_MODULE[I + 1] != '"')
        PRINTF("%c", H_MODULE[++I]);
    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] != ',')
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] != ',')
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 (1)
1616    (TRANSF_COUNT < T_COUNT) ||
1617    (EFF_COUNT[EFF_NO] != 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[++] = ' ';
1626    WHILE (I < 90);
1627    I = 0;
1628    WHILE (I <= 0) && (AFF_COUNT[AFF_NO][I] != 0))
N = 5;

/* PRINTF ("%s\n", "AFF. LOOP");
printf ("%s %d %s %d\n", "I = ", i, "AFF_COUNT = ", AFF_COUNT);
printf ("%s %d\n", "MOD_ORD_TABLE", MOD_ORD_TABLE[AFF_COUNT][0]);
printf ("%s %d\n", "MOD_ORD_TABLE1", MOD_ORD_TABLE[AFF_COUNT][1]); */
if (MOD_ORD_TABLE[AFF_COUNT][0] == 100)
{
    AFF_COUNT = AFF_COUNT + 1;
    continue;
}
if (MOD_ORD_TABLE[AFF_COUNT][1] == 99)
if (MOD_ORD_TABLE[AFF_COUNT + 1][1] == 0)
break;
else
{
    /* PRINTF ("%s\n", "LOADING INIT. AFF."); */
    printf ("%s\n", "LOADING INIT. AFF.");
    print_line[N] = 'A';
    print_line[++N] = 'F';
    print_line[++N] = 'F';
    i = 1;
    break;
}

i = 0;
N = 5;
while (aarr[mod_ord_table[aff_count][1]][i + 1] != ',')
{
    /* PRINTF ("%s\n", print_line[N]); */
    print_line[++N] = aarr[mod_ord_table[aff_count][1]][++i];
}

/* COLUMN OF TRANSFORMS */

i = 0;

while ((i == 0) && (transf_count <= t_count))
{
    /* PRINTF ("%s\n", "MAIN TRANSF. LOOP");
printf ("%s %d %s %d\n", "I = ", i, "TRANSF_COUNT = ", TRANSF_COUNT);
printf ("%s\n", tarr[transf_count][0]); */
    if (tarr[transf_count][0] == 'U')
    {  
        transf_count = transf_count + 1;
        continue;
    }
    i = 0;
    n = 40;
    while (tarr[transf_count][i] != ',')
    {
        /* PRINTF ("%s\n", print_line[N]); */
        print_line[++N] = tarr[transf_count][i];
    }
}

}
/* COLUMN OF EFFERENTS */

I = 0;

while ((I == 0) && (E_MOD_ORD_TABLE[EFF_COUNT][1] != 0))
{
    N = 70;
    /* printf ("%S\n", "EFF. LOOP ");
    printf ("%S %d %S %d\n", "I = ", I, "EFF. COUNT = ", EFF_COUNT);
    printf ("%S %d\n", "E_MOD_1", E_MOD_ORD_TABLE[EFF_COUNT][0]);
    printf ("%S %d\n", "E_MOD_2", E_MOD_ORD_TABLE[EFF_COUNT][1]); */
    if (E_MOD_ORD_TABLE[EFF_COUNT][0] == 100)
    {
        EFF_COUNT = EFF_COUNT + 1;
        continue;
    }
    if (E_MOD_ORD_TABLE[EFF_COUNT][1] == 99)
    {
        if (E_MOD_ORD_TABLE[EFF_COUNT + 1][1] == 0)
        {
            break;
            else
            {
                /* printf ("%S\n", "LOADING INIT EFF"); */
                print_line[N] = 'E';
                print_line[N+N] = 'F';
                print_line[N+N+N] = 'F';
                print_line[N+N+N+N] = 'F_NO';
                i = 1;
                EFF_NO = EFF_NO + 1;
            break;
            }
        }
    }
    I = 0;
    N = 70;
    while (EARR[E_MOD_ORD_TABLE[EFF_COUNT][1]][I + 1] != '.
    {
    /* printf ("%C\n", PRINT_LINE[N]); */
    print_line[N+N] = EARR[E_MOD_ORD_TABLE[EFF_COUNT][1]][I+N];
}

print_line[N+N] = '\0';
I = 0;
do
{
    printf ("%C", PRINT_LINE[N+N]);
    while (I <= N);
    printf ("\n");
}
}
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.