

3-2
A DATAFLOW DIAGRAM GENERATOR

by

ALICIA ELLEN SPECHT

B. A., University of Georgia, 1977

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

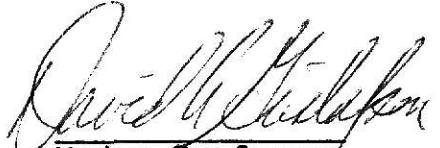
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Approved by:


Major Professor

LD
2668
.R4
1986
.S63
C.2

A11207 233566

CONTENTS

1. Literature Survey	1
1.1 Introduction	1
1.2 The Need for Requirements Engineering	2
1.3 Characteristics of a Good Requirement Specification	4
1.4 Formal Requirement Specifications	8
1.5 Types of Requirements Methodologies	11
1.6 Dataflow Models	14
1.6.1 Dataflow Diagrams	14
1.6.2 SADT	15
1.6.3 PSL/PSA	16
1.7 Future Trends in Requirements Methodologies ...	18
1.8 Conclusion	20
2. Requirements	22
2.1 General	22
2.2 Input	22
2.3 Output	23
3. Design	26
3.1 Introduction	26
3.2 Overview of System Structure	26
3.3 Design of the Dataflow Diagram	27
3.4 Detailed Design	31
3.4.1 Parse Requirement Spec Module	33
3.4.2 Compare Inputs/Outputs Module	35
3.4.3 Column Marking Module	38
3.4.4 Row Marking Module	42
3.4.5 Row Spacing Module	44
3.4.6 Create Graph Module	45
3.5 Implementation and Testing	52
4. Conclusions and Extensions	54
4.1 Conclusions	54
4.2 Extensions	54
References	56
Appendix A - BNF Description of ERA Specification	59
Appendix B - Module Specifications	62
Appendix C - Error Messages	72
Appendix D - User Guide	76
Appendix E - C Language Code for DFD Generator	77
Appendix F - Sample Output from DFD Generator	111

LIST OF FIGURES

Figure 1.	Format of dataflow diagram	24
Figure 2.	Dataflow diagram design	29
Figure 3.	Hierarchy diagram of main modules	32
Figure 4.	Parse_requirements_specs	34
Figure 5.	Compare_inputs_outputs dataflow data structure	37
Figure 6.	Column_marking	39
Figure 7.	Row_marking - column 1	43
Figure 8.	Row_marking - columns 2 through n	44
Figure 9.	Dataflow diagram design	47
Figure 10.	Reserved lines data structure	50

Chapter 1 - Introduction

1.1 Overview

This report describes a tool that produces a dataflow diagram from a requirements specification in textual form. This project is one tool in a prototype system for an advanced software development environment. The dataflow diagram produced by this tool should be useful in the requirements analysis phase of the software development cycle.

The need for adequate requirements analysis in the beginning of the software development process has received considerable attention in recent years [YE82] [R085a] [R077b] [TE77]. This attention has arisen in response to the growing concern of software developers over the inability to develop software systems in a predictable time frame that accurately perform the intended functions of the system. It has been noted that problems in system development have become less traceable to the hardware or the programming [R077b], inferring that the problems resulted from inaccurate requirement specifications.

This paper surveys the literature on requirements engineering with particular emphasis on dataflow analysis as a tool in requirements analysis. This paper discusses:

1) the need for adequate requirements engineering 2) the characteristics of a good requirements specification 3) the benefits of formal requirements specification languages 4) a survey of existing types of requirements modeling methodologies 5) a description of specific dataflow model requirements specification techniques and 6) future goals for requirements specification methodologies.

1.2 The Need for Requirements Engineering

Requirements engineering is an essential step in the software development cycle. Requirements engineering is the iterative process of analyzing the problem to be solved by the proposed software system, documenting the resulting requirements, and checking the documents for accuracy [RZ85]. The resulting requirement specification is then used as a guideline in the software design process. A requirement specification states the intended functional and performance attributes of the system without specifying an implementation plan. Without an accurate requirements specification, the software designer may solve the wrong problem, resulting in a system that does not satisfy the user's needs [RO85a].

Discrepancies between the system requirements and its implementation can result in significant cost overruns and

schedule delays. There are documented cases of two large command control systems that needed to have 67% and 97% of the system rewritten in order to perform their intended functions. It has been estimated that "design errors" comprise 36% to 74% to the total error count, with design errors costing far more to fix than coding errors [YE82]. In large scale systems, a requirements discrepancy detected in a completed system may cost 100 times more to correct than a mistake detected in the systems definition process [RO85]. In severe cases, entire projects have been canceled due to the lack of proper requirements and feasibility analysis. Even changes made under the heading of system "maintenance" are often repairs to correct requirements discrepancies. The lack of adequate requirements documents also makes the task of project control more difficult, resulting in waste and duplication of effort [RO77b].

To prevent the aforementioned problems in software development, requirements engineering can be used to reach an understanding between the user and the developer. Requirement specifications provide a tool for management review and project control. The benefits of requirements engineering include project justification, reduced development cost, reduction in schedule delays and easier system maintenance. In situations requiring contracts

between the system requester and the developer, requirement specifications can be used as a basis for contractual agreements [IE81] [EV80].

Good requirements documents provide the additional benefit of being useful throughout the software life cycle. Since the design of a system is not necessarily an implicit statement of its requirements [RO85a], the requirements document can be used to trace the necessary functional attributes of the system design when the need for system modifications arises.

1.3 Characteristics of a Good Requirement Specification

As stated earlier, a "good" requirements specification can be used to improve the quality of the software being developed and reduce the overall cost of the software. This section of the paper presents guidelines for evaluating the quality of a requirements specification.

The IEEE Guide to Software Requirements [IE81] offers the following fundamental description of a software requirements specification(SRS):

The SRS shall clearly and precisely describe each of the essential requirements (functions, performances, design constraints, and attributes) of the software and the external interfaces. Each requirement shall be defined such that its achievement is

capable of being objectively verified by a prescribed method, for example, inspection, demonstration, analysis, or test.

This description lays the foundation for the qualities of a good software requirements specification which will be discussed in detail in this chapter. This section will cover the kind of information that should be in a requirement specification, the kind of information that should not be in a requirement specification, and general characteristics of a good requirement specification.

A software requirement specification should specify the results that must be achieved by the software without specifying design or implementation details. Such implementation details include partitioning software into modules, allocating functions to modules, and indicating the flow of control between modules. By omitting design details, with the exception of design constraints, the developer is given maximum flexibility to choose the most appropriate means to satisfy the requirements [IE81].

The requirements specification should not specify verification and project management details. Project management details such as the management of the software development, delivery schedules, and documentation should be described in a separate project management document [IE81].

The requirements that should be specified in the software requirements specification fall into two categories, functional requirements and non-functional requirements. Functional requirements describe the behavior of the software. They provide a conceptual model of the software without specifying the design or implementation. Non-functional requirements restrict the solutions that may be considered for implementation. Non-functional requirements may increase the complexity of the design. Examples of some common non-functional requirements are as follows:

1. Interface constraints - Interface constraints define the interaction between the software and its environment.
2. Performance constraints - Examples of performance constraints include time requirements, reliability and security.
3. Operating constraints - Examples of operating constraints are personnel availability, skill levels of personnel, and environmental considerations such as temperature.
4. Life cycle constraints - Life cycle constraints include maintainability, enhancability, portability, and flexibility.

5. Economic constraints - Economic constraints identify considerations about immediate and long term costs.
6. Political constraints - Political constraints address policy and legal issues such as unwillingness to use a competitor's product [R085a].

The following section identifies and explains some characteristics of a good requirement specification [IE81] [R085a]:

- | | |
|-------------|---|
| Unambiguous | - Each stated requirement should have only one interpretation. |
| Complete | - All significant requirements should be included. |
| Verifiable | - For every stated requirement, there exists a cost effective way to see that the final product meets the requirements. |
| Consistent | - There should be no conflict between requirements in a specification. |
| Modifiable | - The structure of the requirements specification should allow changes to be easily made while preserving the |

completeness and consistency.

- | | |
|---------------|---|
| Traceable | - The requirement specification should record the origin of each requirement and identify the essential requirements. |
| Constructable | - The structure should allow people or machines to easily build the specification. |
| Analyzable | - The higher the degree of formality, the more adaptable a software requirement specification is to analysis by mechanical means. |

1.4 Formal Requirement Specifications

Constructing a requirement specification that embodies the qualities listed above is a difficult task. The complexity of large scale systems makes them difficult to define without oversights and inconsistencies. The matter is further complicated by the fact that system developers and users often do not use the same vocabulary. This inadequate communication can result in misunderstandings in the requirements that are not discovered until late in the development process [R077b].

Requirements specifications in a natural language are

particularly prone to problems since natural language is inherently ambiguous. When writing a requirement specification in a natural language, it is necessary to define common words per their usage in the requirements specification. Arriving at a set of terms understandable to all parties can be complicated. Requirements written in a natural language prose can be lengthy and time consuming to read. Without proper structuring, the requirements document can easily become incomplete, excessive and inconsistent [R077b].

The structure of the requirements document should also enforce the exclusion of certain types of information such as design details. This feature is important since the requirements document is often written by system analysts who find it difficult not to embed design decisions in their requirements definition [R077b].

These problems with constructing a proper requirements specification have lead to the development of formal requirement specification languages. Formal requirement specification languages were derived as a solution to the ambiguity and imprecision of natural language specifications. Requirements specification languages provide formal syntax and semantics which reduce ambiguity. The formal structure of the language provides a structured

way of thinking for the requirements analyst [R077b].

The higher the degree of formality of the language, the more adaptable the language is to being automated [R085a]. The automated requirements language methodologies have analyzers that mechanically detect syntax errors and enforce proper structuring of the requirement specification. With automated methodologies, the requirements are stored in a database which allows modifications to be more easily made and provides the basis for automatic report generation. Automation provides the means to display aspects of the requirements in tabular or graphical form. The nouns and verbs in the requirements that correspond to entities/actions in the formal requirements language can easily be tabulated to form a cross-reference table.

A possible disadvantage of using a formal requirements specification language is the high learning curve associated with some methodologies. Many formal methodologies require extensive training in order to use the methodology correctly and gain the expected benefits from the methodology. There are some considerations for determining whether a formal requirements methodology may be useful. The size and complexity of the target system should be considered. If the target system is relatively small or simple, a formal requirements methodology may not be worth the investment.

If a contract between the specifier and the customer is necessary, the requirements specification becomes more important. The specification can be used as a basis for developing the customer contract. The computer resources available to support a given methodology are a factor for consideration. Many automated methodologies require considerable database processing facilities in order to implement the methodology [IE81].

1.5 Types of Requirements Methodologies

Many different types of requirement specification techniques have evolved to solve different classes of problems. Most methodologies provide a means of formulating a conceptual model of the target system and its environment. These formal modeling methodologies use abstractions to suppress unnecessary details while specifying the essential properties of the system [YE82].

While requirements models may be partitioned into many very specific categories, this paper discusses three general types of requirements models defined in Yeh's paper on requirements analysis [YE82]. Those models are data models, dataflow models, and process models.

A data model specification specifies the different states of the system and its environment in terms of data structures.

The data structures in the requirements specification need not be implemented, since the data structures are derived strictly to specify the functional behavior of the system. Data model requirement specifications have been used successfully for data processing and business information systems.

The dataflow model requirement specification is the most commonly used requirements model. It uses dataflow diagrams in conjunction with the data model data structures specification to identify the major processing activities in the system, and indicate which data are inputs and outputs to each activity. The dataflow model is not only suited to model software requirements, it is also well suited to model the structure and behavior of most human organizations [R085a]. The dataflow model is the formal foundation for the SADT [R077a] [R077b] [R085b], and PSL/PSA [TE77] requirements methodologies which will be discussed in the dataflow methodology section of this chapter.

Dataflow models have deficiencies in modeling embedded systems. The activities of embedded systems are continuous, they are not activated by the appearance of input. The activities consist of complex computations which process concurrently. The dataflow model is not powerful enough to specify the behavior of a system with continuous, concurrent

operations [YE82].

The process model is appropriate for embedded systems. It is used to specify processes rather than activities. A process is "an autonomous computational unit which is understood to operate in parallel with, and interact asynchronously with other processes" [YE82]. The specification of a process defines the set of possible states of the process and a "successor relation" which maps the predecessor state into the possible successor states. The processes can represent objects in the target system and objects in the environment. Performance constraints can also be included for the processes.

SREM is a well known example of a process model methodology. It provides an executable model that specifies processing paths from the input stimulus to the output response. SREM includes a simulation feature to evaluate performance [Ye82] [BE77].

No one methodology is equally suitable to all applications. The success of a methodology ultimately depends on human factors. A methodology should be easy to learn and provide an understandable way of constructing a specification before it will be a benefit to most users [RO85a].

1.6 Dataflow Models

This section of the paper gives a description of dataflow diagrams in general, and explains their usage in requirements analysis. It then presents a summary of two of the most commonly used commercial dataflow model requirements methodologies, SADT and PSL/PSA [R077a] [R077b] [R085b] [TE77].

1.6.1 Dataflow Diagrams

A dataflow diagram is a tool used to analyze the information flow of a system, portraying the system information flow in graphic form for increased understandability. It defines the transformations data undergoes as it flows through the system, without specifying control information or the sequence of activities. Transformations are represented usually by labeled circles or boxes, with the dataflow between transformations represented by directed lines proceeding from left to right. All data is clearly labeled.

While a dataflow diagram is not a system flowchart, it is procedural in that it shows the sequence of steps as data is transformed from input to output. Because it is procedural, it can be used to assign performance constraints to information flow paths. A dataflow diagram can also be used as a starting point for establishing a structure diagram

showing the hierarchy of modules in the system [JE79].

1.6.2 SADT

The following information about SADT comes from the Ross references [RO77a] [RO77b] [RO85b]. The SADT structured analysis and design technique methodology uses the format of dataflow diagrams to define the functional architecture of a system. SADT is a blueprint-like language that stresses accurate communication of ideas through graphical representation. It combines the nouns and verbs of any other language with its rigorous graphical structure to present a gradual decomposition of detail.

The SADT system model consists of a set of diagrams that are organized in a hierarchical, top-down structure. There are two types of diagrams in the SADT methodology, one that shows the decomposition of activities, the other that shows the decomposition of data in the system. In the SADT activity decomposition diagrams, activities are represented by labeled boxes with the data flowing between activities represented by labeled lines. In the data decomposition graphs, the data is represented by labeled boxes and the activities that produce and receive the data are represented by labeled arrows.

Both types of graphs are generated with a rigorous syntax.

A given graph may have no more than 6 boxes, each box may decompose to another graph. A graph must include all of the information about the viewpoint that it is supposed to express. There are rigorous labeling standards for the graphs. These standards stress labeling a box in a graph and the graph generated from that box with the same identifier. This feature simplifies tracing through the layers of the graphs.

SADT presents an orderly, structured decomposition of the subject and provides a structured way of thinking about the subject. The SADT requirements methodology is adaptable to many applications and it incorporates any other language. One of SADT's primary drawbacks is that it is not an automated system.

1.6.3 PSL/PSA

The following information on PSL/PSA is derived from the Teichroew reference [TE77]. PSL/PSA is one of the most commonly used commercial requirements methodologies. It is an automated system that provides PSL, the Problem Statement Language, and PSA, the Problem Statement Analyzer. The system records requirements written in PSL into an internal database. The PSA software performs syntax validation and provides the capabilities to generate reports from the

database.

The PSL language offers a predefined set of objects commonly needed in software requirements definition. The objects are defined to have properties, and the properties to have values. The objects are interrelated with a predefined set of relationships. The PSA software provides consistency checking to enforce adherence to the relationships.

Within this framework, PSL/PSA allows the specification of system requirements using eight general categories: system input and output flow, system structure, data structure, data derivation, system size and volume, system properties and project management.

The developers of PSL/PSA claim that it improves the quality of the requirements documentation and reduces the development and maintenance cost of the system. Errors are detected early in the requirements definition process when they can be corrected at less expense. The effects of a proposed modification to the system can be more easily isolated using an automated system.

The aforementioned dataflow requirements methodologies have become commercially popular due to their broad applicability, especially to common business processing. However, they can be deficient for some applications in

their ability to specify knowledge about the environment of which the software system will be a part. The next section discusses a new direction in requirements modeling that focuses on "capturing more world knowledge" [GR82].

1.7 Future Trends in Requirements Methodologies

The information for this section comes from the Borgida, Greenspan, and Roman references [BO85] [GR82] [RO85a]. A new direction in requirements methodologies has emerged from the field of artificial intelligence work on knowledge representation. It addresses the problem of traditional functional requirements specifications that specify the behavior of a software system, but provide little information about the environment in which the software will function. This approach assumes that the software developer is knowledgeable in the applications area of the target system, which often is not the case. There is a need to represent "world knowledge" in the requirements specification to give the software developer a framework in which to properly interpret the functional requirements.

This need has lead to the development of the RML requirements modeling language. The RML language allows the requirements analyst to model the world by defining a real-world set of objects that are relevant to the environment in

which the software will be functioning. An object can be an entity, activity, assertion, or any concept of the real world.

A given object is defined to have certain properties, the objects are grouped into classes on the basis of common properties. The relationships between the classes are defined by decomposing classes into subclasses. Each subclass inherits the properties of its class; in defining a subclass, the specifier need only state how the subclass differs from the class. This technique of generalization hierarchies allows the specifier to begin by defining the most general classes, and by "stepwise refinement by specialization", decompose the classes into more specialized classes.

It is suggested that the specifier first model the total environment in which the target system will function. The software system can then be modeled by establishing the boundaries of the world model that the software system is supposed to address.

This "world knowledge" approach allows the specifier maximum flexibility. Inherent in this flexibility is the difficulty of determining which concepts of the real world are relevant to the model. The specifier must also deal with the problem

of over-abstraction. Useful generalizations in the natural world will often have contradictions embedded in them, for example "birds fly" and "penguins don't fly." For this reason, the designer is allowed to specify exceptional classes and objects for certain exceptions, with the reason for the exception also being specified.

The real-world approach to requirements modeling is still in its infancy. While the RML language has been used to specify at least one large software system, it does not enjoy widespread commercial acceptance. However, the artificial intelligence knowledge representation based requirements research does appear to hold interesting promise for the future.

1.8 Conclusion

Requirements engineering is a vital part of the software development life cycle. Good requirements specifications reduce the cost of system development and can be used throughout the life cycle to improve maintainability of the system.

The purpose of requirements specifications is to specify the essential behavior of the target system, not a design for the target system. Requirements specifications specify both functional and non-functional requirements such as

performance constraints. A good, usable, specification is unambiguous, complete, consistent and traceable.

Formal requirements specification methodologies have been developed to improve the quality of requirements specifications. The requirement specification methodologies founded on the dataflow model enjoy the most wide spread usage. They are particularly appropriate for business information systems applications. The dataflow diagram is a graphical aid used by these methodologies to depict the dataflow of a system for increased understandability.

Research for future requirements specification methodologies centers around the use of artificial intelligence knowledge representation theory to specify not only basic functional requirements, but knowledge of the application area as well.

Chapter 2 - Requirements

2.1 General Requirements

This software project produces a dataflow diagram from an entity- relationship-attribute requirements specification. It was developed to be used as a tool in an experimental software development environment that uses software engineering techniques to improve the quality and efficiency of the software development process.

This tool reads an input requirement specification in textual form and produces a file of graphics commands that represent a diagram of the flow of data between the activities in the specified system. The diagram is produced from the input specification automatically, requiring no additional input from the user.

It was a constraint that the tool be developed to execute on the UNIX III operating system at Kansas State University, and that the programs be implemented in Pascal or C language.

2.2 Input

The input to this tool is a textual requirement specification that specifies the activities in a system, and each activity's inputs and outputs. The syntax of the input

specification was defined by Dr. David A. Gustafson of Kansas State University; a BNF description of the input syntax is provided in Appendix A. All input to this tool is assumed to be syntactically correct.

This tool is required to parse the input specification, identifying the activities and their inputs and outputs. It must then match input names to output names to calculate the dataflow between activities.

Since there is no limit to the number of activities, inputs and outputs that this tool is required to accommodate, this tool uses a dynamically allocated, linked list data structure to store the data flow information.

2.3 Output

The dataflow diagram produced by this product is intended to be used as a graphical aid in requirements analysis. Therefore, the general requirements for the output dataflow diagram follow the rules of a good graphical aid. The dataflow diagram produced by this tool must be a complete, accurate representation of the dataflow of the system. All activities, inputs and outputs should be labeled. The dataflow diagram must be legible, allowing the user to easily read activity and data names, and to readily distinguish dataflow lines.

**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.**

The display format of the output dataflow diagram is illustrated in Figure 1. The activities are represented by labeled boxes, the dataflow is represented by labeled arrows. The data flows from left to right unless there is a loop in the dataflow. It is not a requirement for this project that the output dataflow diagram be an optimal solution with minimal crossing of lines.

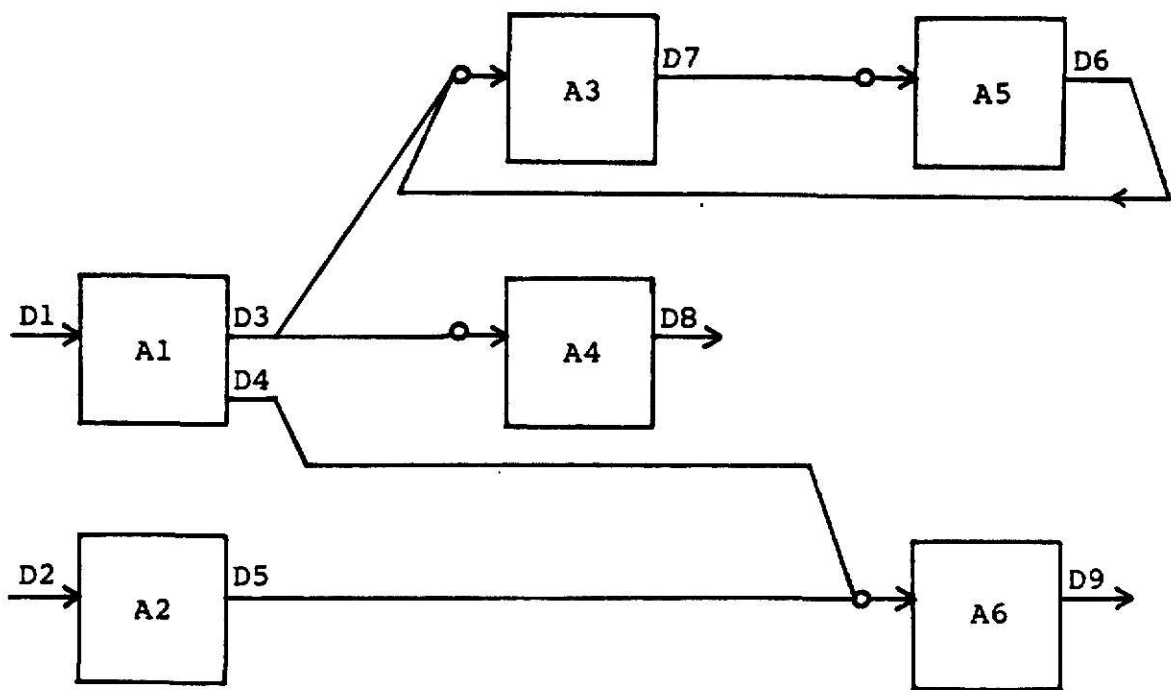


Figure 1: Format of dataflow diagram

Since this tool is required to produce a legible, accurate diagram no matter how complex the dataflow, the format in which the dataflow diagram is displayed is designed to accommodate these requirements.

The activity boxes are arranged in rows and columns with enough space between the rows and columns for dataflow lines. This design insures the ability to draw a line from a source activity, an activity that produces a given data file, to a destination activity, an activity that receives a given data file, without the lines crossing through activity boxes or merging with other lines.

To improve the readability of the output dataflow diagram, all inputs go into the left side of the activity box, all outputs come from the right side of the activity box. There is only one line for a given input going into a given activity box. The input line is preceded by an input collection node into which the dataflow lines from any number of source activities for that input may flow. There is only one output line for a given output coming out of an activity box. An output line begins with the output label line. From the end of the label line, output lines can diverge to reach any number of destination activities. This design reduces the number of lines going in and out of activity boxes and allows the user to readily distinguish inputs from outputs.

Chapter 3 - Design

3.1 Introduction

This chapter begins with an overview of the structure of this software product and the logical flow of its main functions. This paper then discusses issues that needed to be addressed in designing the display format of an automatically generated dataflow diagram and explains how those issues are resolved in the design. An analysis of the main modules in the system follows which describes the functions of the main modules and the usage of data structures.

3.2 Overview of System Structure

This software project is designed as one program that creates a dynamically allocated, linked list data structure to contain the data flow information in the input requirement specification. From this data structure, it produces a file of graphics commands that depict a dataflow diagram for the requirement specification. The program is organized into six main modules, one for each major function.

The dynamically allocated, linked list data structure was chosen for this application because of its flexibility in

the number of activities, inputs and outputs that it can accommodate. The data structure contains one node for each activity and one node for each data name in the requirements specification.

The basic functional flow of the program begins with reading the requirement specification input file, to extract the activities and their inputs and outputs. The input data names are matched to the output data names to calculate the dataflow between activities. Once the data structure containing the dataflow information has been created, the information in the data structure is used by the graphing algorithms to determine how the dataflow will be graphed. A file of graphics commands depicting the activity boxes and dataflow lines is created according to the graph placement algorithms.

3.3 Design of the Dataflow Diagram

This section of the paper explains the format in which the dataflow diagrams produced by the tool are displayed, and the reasons behind the chosen format.

When drawing a dataflow diagram manually, there are many types of conflicts that are resolved intuitively. These conflicts must be resolved in the design of the graph in an automatic tool. The potential conflicts that are resolved in

the design of the dataflow diagram format are listed below.

The design must guarantee the ability to get a dataflow line from a source activity to a destination activity without crossing through an activity marker. The dataflow lines must not merge so as to become indistinguishable. The design must control the placement of input lines into an activity and the placement of output lines out of an activity so that the lines remain discernible and the data labels remain readable.

To assure the ability to produce an accurate, legible dataflow diagram, the graph design displayed in Figure 2 was adopted. Activities are displayed as labeled boxes arranged in rows and columns. The program allows enough space between the rows and columns to graph the lines depicting dataflow.

Input into an activity is graphed in the space between columns as a horizontal arrow going into the left side of the activity box. All input lines are the same length, with input lines in the same column having the same x coordinates. Except for initial inputs, input lines are preceded by a collection node. There is one input line for a given input into an activity; dataflow lines from any number of source activities can feed into the input

collection node.

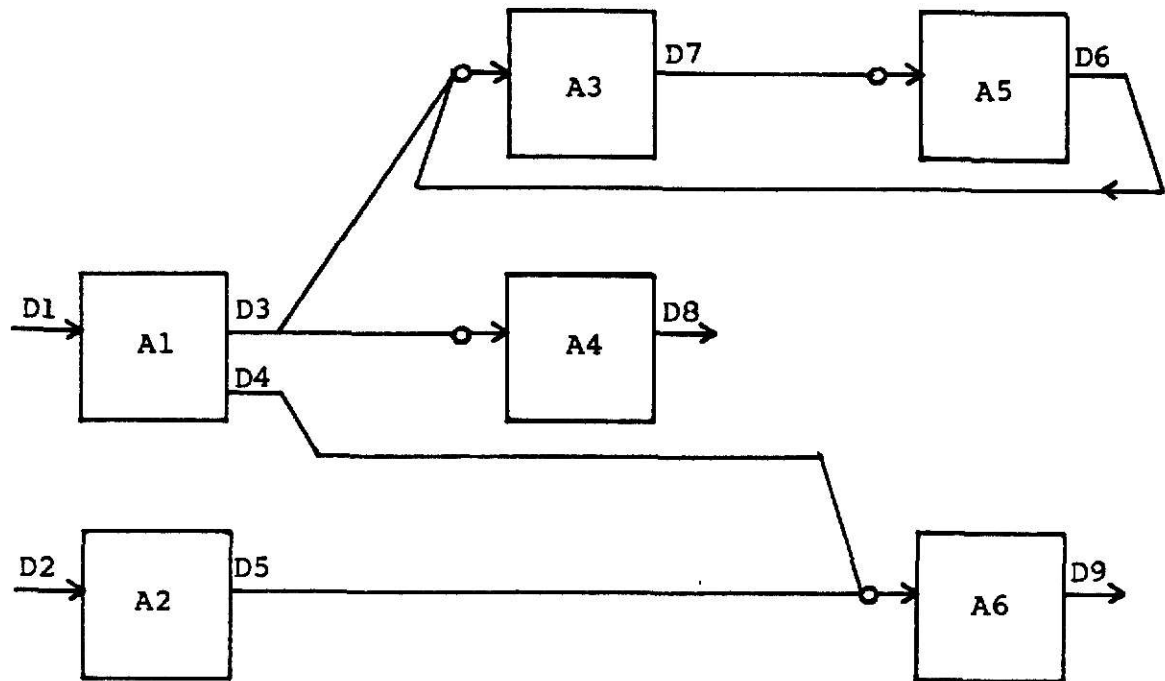


Figure 2: Dataflow diagram design

Output lines are graphed in the space between columns as labeled horizontal lines coming out of the right side of the activity box. All output lines have the same length, with output lines in the same column having the same x coordinates. There is one output line for a given output out of an activity; the dataflow lines to an output's destination activities diverge from the end of the output label line.

This design for displaying the inputs and outputs of activities keeps the inputs easily distinguishable from the

outputs, and prevents input lines from crossing the labeled portion of the output lines.

The design must assure the ability to graph a dataflow line from the output label line of the source activity to the input collection node of the destination activity. These lines must be graphed without crossing through activity boxes or merging with other lines. To accommodate this constraint, the following line placement design was developed.

To graph a line that has no potential for crossing through an activity box, the dataflow line is graphed directly from the end of the output label line to the collection node for the input into the destination activity. A line has no potential for crossing through an activity box if it is going to a destination activity in the next column to the right, or if the destination activity is the next activity in the same row.

For a line that must span more than one column, the line is graphed as a horizontal line in the space between rows. The y coordinate of the horizontal line is assigned an unused y coordinate in the space immediately above or below the source activity's row. The x coordinate of the starting point of the horizontal line is assigned based on a uniform

distance to the right of the output label line. The x coordinate of the end point of the horizontal line is assigned based on a uniform distance to the left of the input collection node. The dataflow line is thus graphed in three segments: 1) from the end of the output label line to the beginning of the horizontal line between the rows 2) from the starting point of the horizontal line to the end point of the horizontal line 3) and finally from the horizontal line end point to the input collection node.

This graph design, while not boasting the optimal solution for a given dataflow diagram, does insure the ability to provide a legible, accurate dataflow diagram.

3.4 Detailed Design

This section of the paper explains the functions of the main modules of this tool and the data structures used. The program is divided into the six main modules displayed in Figure 3.

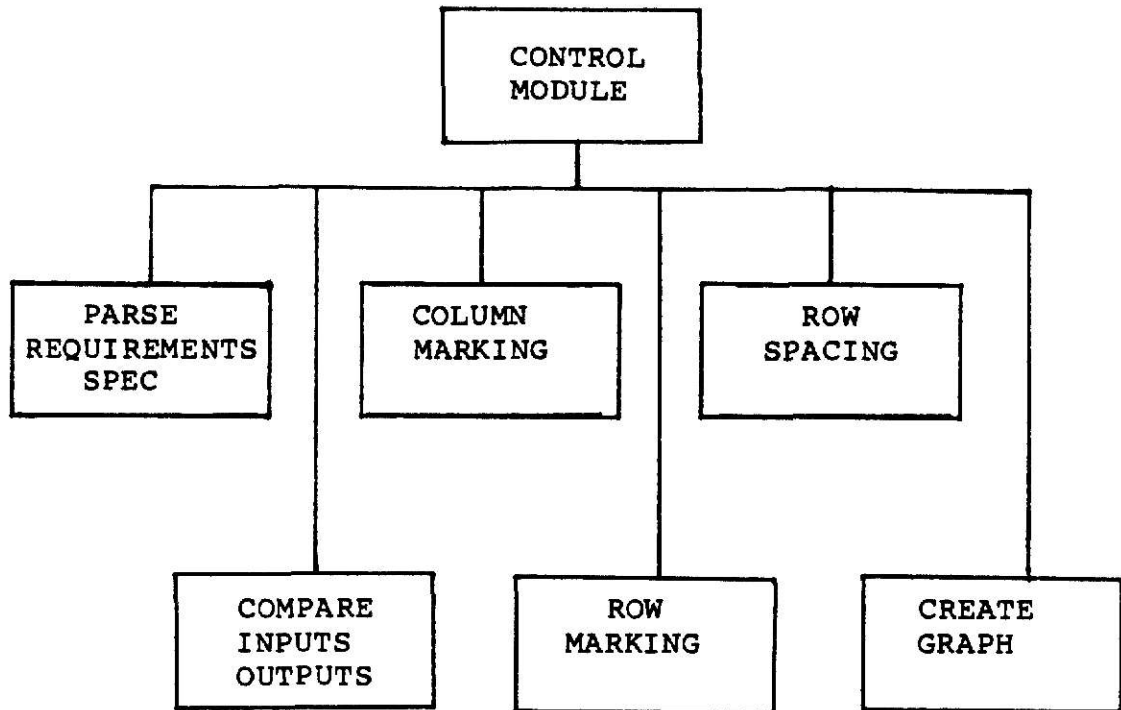


Figure 3: Hierarchy diagram of main modules

The six functions of the six main modules are as follows:

- 1) Parse_Requirements_Spec - This module parses the requirement specification for activities and their inputs and outputs.
- 2) Compare_Inputs_Outputs - This module compares the input names to the output names to calculate the data flow and create the data structure containing this information.
- 3) Column_Marking - This module assigns column numbers to each activity.
- 4) Row_Marking - This module assigns row numbers to each

activity.

5) Row_Spacing - This module calculates the amount of space needed between rows for dataflow lines.

6) Create_Graph - This module calculates the coordinates for the activity boxes and dataflow lines, and creates the file of graphics commands to graph them.

3.4.1 Parse_Requirements_Spec Module

The Parse_Requirement_Spec module extracts the activities and their inputs and outputs from the input requirement specification in the following manner. The module scans the input specification for activity frames. When an activity frame is encountered, the module allocates an activity node and stores the activity name in the name field of the activity node. The activity node is then linked into a singly linked list of activity nodes. The module parses the activity frame for the activity's inputs and outputs until it encounters the beginning of the next frame.

When an input is found, the module writes a record to a sort_inputs file; the record contains the input name and a pointer to the current activity node. If no inputs are found for an activity, a sort_inputs record is created with the reserved word "NONE" in the input name to be used by the

graphing algorithms to identify activities that should be placed in the first column of the graph.

The same process is used for outputs. A `sort_outputs` file is created that contains a record for each output, and destination activity combination. Figure 4 shows the activity nodes and `sort_inputs` and `sort_outputs` records that are created from the example requirement specification in the figure.

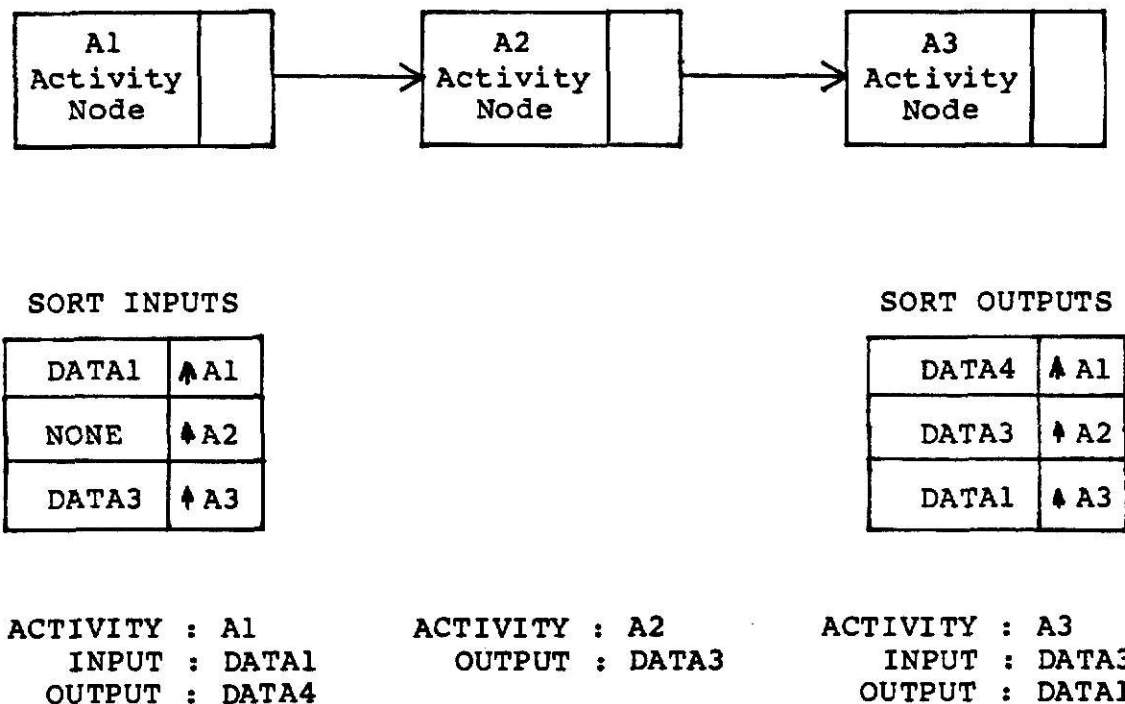


Figure 4: `Parse_requirements_spec`

The files of inputs and outputs are sorted by a system sort

on the data name field. The sorted files are used in the input-output matching algorithm of the Compare_Inputs_Outputs module.

3.4.2 Compare_Inputs_Outputs Module

The Compare_Inputs_Outputs module compares the input records from the sort_inputs file with the output records from the sort_outputs file. When a match on the data name field between the sort_inputs and sort_outputs records is encountered, the module updates the data structure to reflect dataflow from the source activity in the sort_outputs record to the destination activity in the sort_inputs record. If no match is found for a data name in the files, then the data name is either an initial input or a final output.

To contain this information, the module constructs a data structure which consists of initial input nodes, output nodes, and pointer nodes. The pointer nodes are used to point to activity nodes for listing the destination activities of data, and the source activities for the destination activities.

The list of initial input nodes is constructed from those input names in the sort_inputs file that do not have a match in the sort_outputs file. The list of initial input nodes

contains one node for each initial input name. The initial input node contains the input name, a pointer to the next initial input node, and a pointer to a list of pointer nodes which specify the destination activities for that input. The list of destination activities for an initial input is built from the activity pointers associated with the input name on the `sort_inputs` record.

If the `sort_inputs` file contains records with the input name "NONE", indicating an activity that receives no input, an initial input node is allocated with "NONE" in the name field. The "NONE" node points to a list of pointer nodes that reference all activities that receive no input. The result is a list of inputs that identify the activities receiving initial inputs or no input. This list is used as a consideration for placing an activity in the first column of the dataflow diagram.

For any data name that is not an initial input, an output node is allocated and linked into the list of all outputs. Figure 5 demonstrates how the initial input nodes, activity nodes and output nodes are linked together via pointer nodes to represent the dataflow of the requirement specification. Each activity node points to a list of output pointer nodes. Each output pointer node points to an output node of an output that is produced by that activity. For each record

in the `sort_outputs` file, the activity pointer associated with the output name is used to update the activity node's list of output pointer nodes. Thus, each activity points to a list of its outputs.

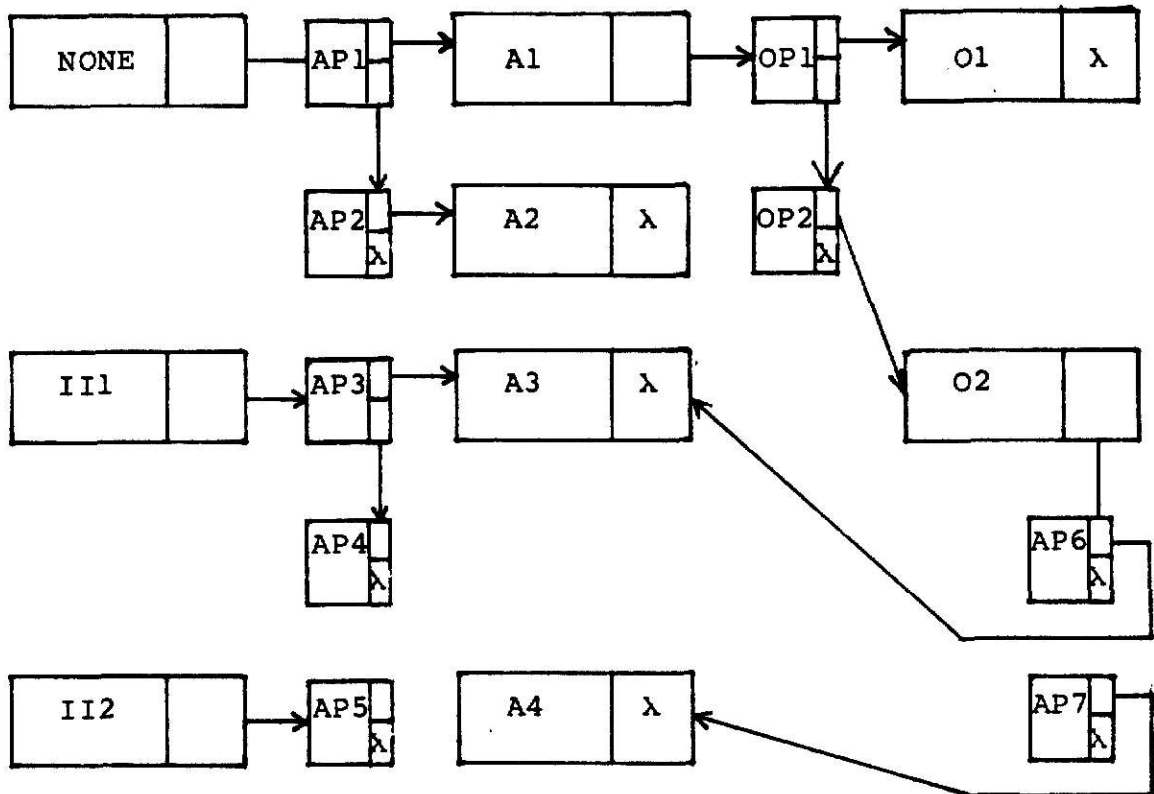


Figure 5: Compare_inputs_outputs dataflow data structure

To represent the flow of data of an output to its destination activities, each output node points to a list of activity pointer nodes. Each activity pointer node points to an activity node receiving that output. The output node's pointer to its destination list will be null if the output

is a final output.

At this point, the representation of the dataflow of the specified system is complete, with the initial input nodes pointing to their destination activities, the activity nodes pointing to their outputs, and the outputs pointing to their destination activities.

A list of each activity's source activity nodes is also created in this module. Each activity node points to a list of pointer nodes which identify all of the activities which produce input for the given activity. This structure provides a quick reference for the graphing algorithms to determine where to place an activity in the graph.

Thus, at the end of the Compare_Inputs_Outputs module, the data structure consists of: 1) a list of initial input nodes, each of which points to a list of pointers to its destination activity nodes 2) a list of output nodes, each of which points to a list of pointers to its destination activity nodes and 3) a list of activity nodes, each of which points to a list of its output nodes, and to a list of pointers to its source activity nodes.

3.4.3 Column_Marking Module

The next module to execute is the Column_Marking module.

This module establishes the column number in which each activity will be graphed, and creates a list of activities comprising each column. Each activity node has a column number field in which the column marking algorithm stores the activity's column number.

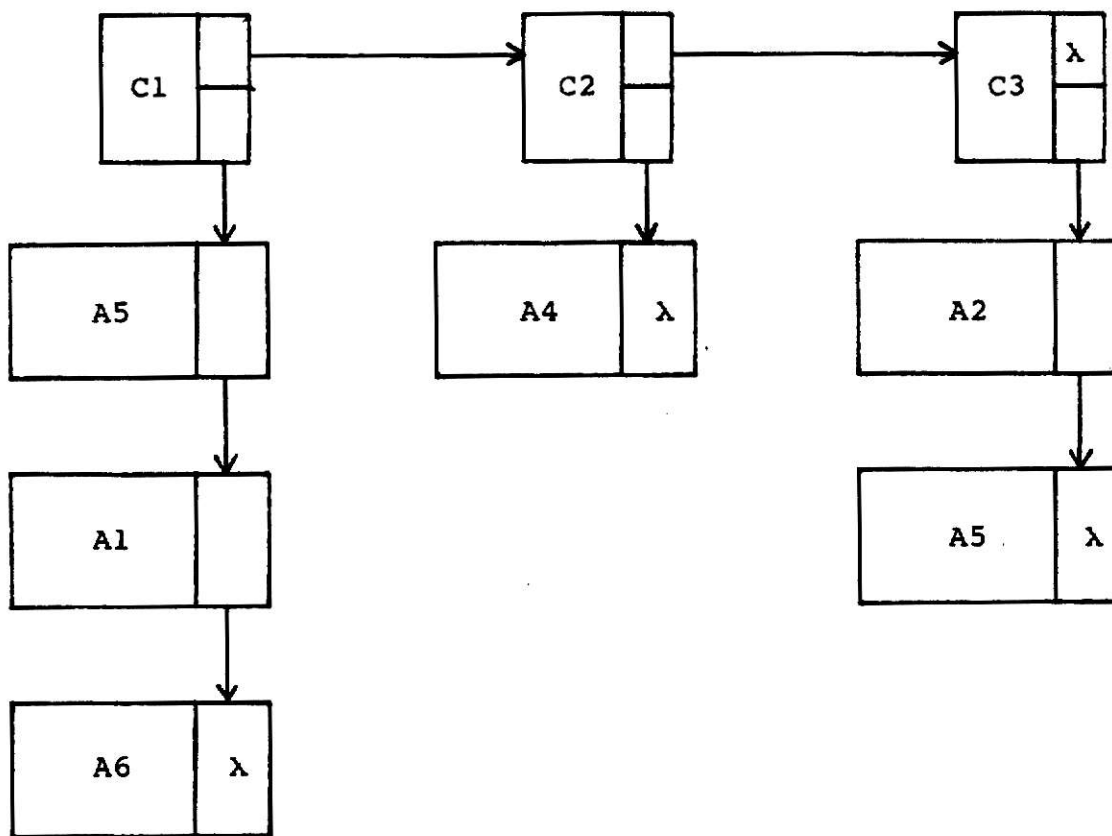


Figure 6: Column_marking

As Figure 6 illustrates, each activity node contains a pointer to the next activity in the column which is used to establish lists of the activities by column. Each column list is pointed to by a column header node which contains

the column number and a pointer to the next column header node. The column header nodes are linked together in order by column number to allow traversal of consecutive columns.

As a starting point for placing the activities in columns, the module selects the activities to be placed in the first column. These activities are the activities that have no input or only initial inputs. To select these activities, the activity node list is traversed and any activity node having an empty source activity list is marked with a "1" in the column number field and linked into the list of column one activities.

The subsequent columns are established in a loop that builds the next column of activities, from the previous column of activities' destination activities. The output list is traversed for every activity in the previous column. The output nodes referenced by those output lists are visited, and their destination activity lists are traversed to identify all of the activities fed by the previous column's activities. Each destination activity node is visited, if the activity has not previously been marked, it is marked with the current column number and added into the current column list.

If the destination activity node has already been marked as

a member of a previous column, it is added to the current column and removed from its former column unless a loop is detected in the dataflow. If the activity node being "bumped" to the current column is in a dataflow loop with the source activity triggering it to be bumped, the activity is not remarked. Thus, when dataflow loops exist between activities in the specified system, the first activity in the loop encountered by the column marking module will remain the first activity of the loop to be graphed unless it is bumped forward by another source activity.

Loop detection is accomplished by maintaining an input path list for each activity as it is placed in a column. The input path list for a given activity is composed of a list of pointers to activities that are in an input path to the given activity. The input path list for an activity is accumulated from the source activities that directly feed an activity, plus the activities listed in those source activities' input path list.

When an activity is being visited for the purpose of being bumped, the input path of the source activity triggering the potential bump is scanned for a match against the activity to be bumped. If the activity to be bumped exists in the input path of the source activity, a loop exists in the dataflow and the activity will not be remarked. The process

of marking the next column from the previous column continues until there are no more activities that can be bumped.

3.4.4 Row Marking Module

After the Column_Marking module places the activities in columns, the Row_Marking module assigns each activity a row number within its column and creates lists of the activities in each row. The activity node contains a field to record the row number and a pointer to the next activity node in the row.

For the first column of activities, the row numbers are assigned with the intention of grouping together activities that produce the same output and therefore, feed the same destination activities. The module builds a table containing every output produced by a first column activity (Figure 7). Associated with each output is a list of activities that produce the output. The activities having no common outputs are assigned consecutive row numbers beginning with one. The module then uses the table of output-activity relationships to mark activities producing a common output in consecutive rows. An activity belonging to more than one output grouping retains the first row number with which it was marked.

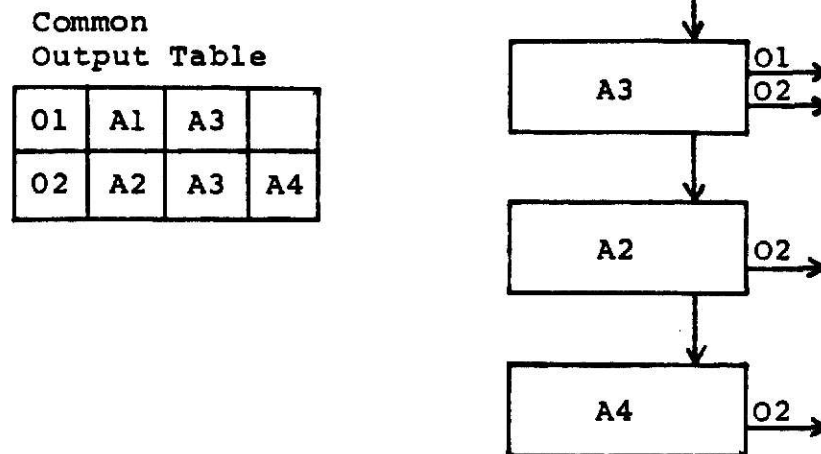


Figure 7: Row_marking - column 1

For the subsequent columns, the activities are assigned row numbers on the basis of having common source activities in the previous column (Figure 8). For each activity in the column being marked, the activity node's source activity list is scanned to determine the range of row numbers in which its source activities in the previous column were placed. The row placement algorithm attempts to center the activity being marked between the range of row numbers of its source activities in the previous column. The module interrogates a table that records the row numbers in the column that have already been assigned until it finds the empty row number closest to the center of the source activities' row range. The activity module is then marked

with that row number and the table of assigned row numbers is updated to reflect that information.

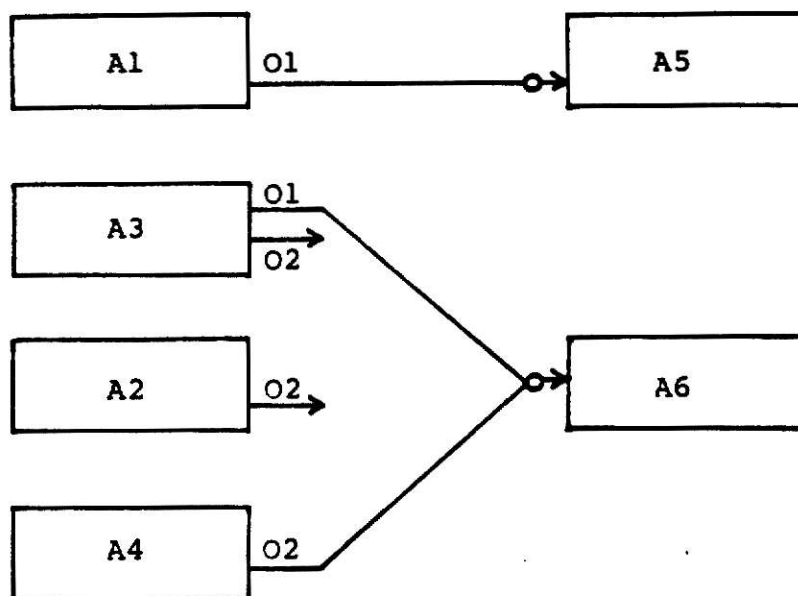


Figure 8: Row_marking - columns 2 through n

The aforementioned scheme positions activities that send and receive the same data close together in an effort to reduce the number of lines crossing between columns.

Once all activity nodes have been marked with a row number, the module establishes the lists of activities in each row. The row lists are used in the line graphing algorithm.

3.4.5 Row_Spacing Module

Now that the activity nodes have been ordered in rows and columns, the Row_Spacing module can determine how much space

will be needed between the rows for the dataflow lines. The Row_Spacing module employs the same logic as the line graphing module to identify instances where a horizontal line between rows will be needed. The module counts these instances to determine the number of lines that will be graphed between each pair of rows. If the number of lines to be graphed between a pair of rows requires more space than the default row space, the space between all of the rows is increased to accommodate the largest demand.

3.4.6 Create_Graph Module

The final module Create_Graph, calculates the coordinates of the activity boxes and dataflow lines, and creates a file of graphics commands to graph the dataflow diagram.

The module begins by identifying the center activity box of the graph from the total number of columns and rows. The coordinates (0,0) are assigned to the lower left corner of the center activity box. A box is defined by calculating the coordinates of the lower left corner and the upper right corner of the box. The coordinates of the activity boxes are calculated based on their distance from the center box. The distance from the center activity box is calculated using the activity's row and column numbers, the height and width of the boxes, and the distance between rows and

columns. The two points that define an activity box are stored in each activity node.

Once the coordinates for the activity boxes are established, the coordinates of the dataflow lines are computed. The dataflow lines are calculated in three pieces: 1) the horizontal input line going into the left of the activity box 2) the output label line coming out of the right of the activity box 3) and the lines connecting the output label line to the input line.

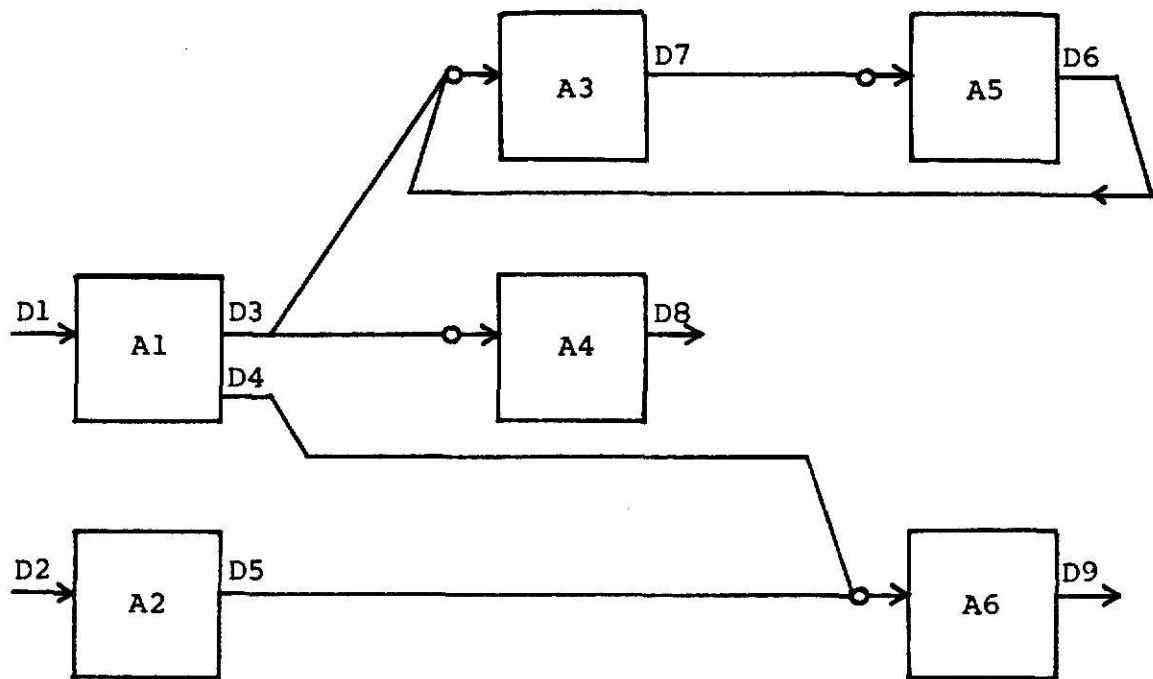


Figure 9: Dataflow diagram design

An input line is graphed as a labeled, horizontal arrow going into the left side of the box. The x coordinates of the input lines going into activity boxes in the same row will be the same, since they are calculated from the x coordinate of the left side of the activity box and the input line length. The y coordinate of the input line is calculated from a field in the activity node in which the value of the last y coordinate for an input line into that box is stored. The field is initialized to the y coordinate of the top of the box, and the y coordinate of a new input line is calculated by subtracting the distance between input lines from the y coordinate of the last input line. The resulting y coordinate is used to update the input line

field in the activity node.

For intermediate inputs, inputs generated from within the specified system, a collection node is graphed at the beginning of the input line. Dataflow lines from all source activities feed into the input collection node for that data. The location of an input line for a given input into a given activity is stored in the output list node that points the output to the source activity. This structure makes the location of an existing input line into an activity readily available for the algorithm that graphs the line connecting output label lines to the input lines. Therefore, only one input line for a given input and activity is necessary.

An exception to the input lines that go into the left side of the box is the case of an initial input that goes to an activity that is not in the first column. These inputs are graphed as horizontal lines in the space between rows going into the top of the box. The y coordinate for the horizontal line is computed from a data structure that keeps track of the next available y coordinate in that row space. The x coordinate for the line's entry point into the top of the box is calculated from the last entry point into the top of the box, which is recorded in a field in the activity node.

The output label line is graphed as a labeled, horizontal line coming out of the right side of the activity box. There is one output line for a given output coming out of a given activity. From the end of the output label line, lines may diverge to reach multiple destination activities. If the output is a final output, an arrow is added to the end of the output label line.

To graph a line from an output label line to an input collection node, the following methodology is used. For a line going to a destination activity one column to the right, or for a line going to a destination activity that is the next activity to the right in the same row, the line is graphed directly from the output label line to the input collection node. The direct line is possible because, under the aforementioned conditions, there can be no activity boxes between the source activity and the destination activity. The list of activities in each row is used to determine when a direct line is possible between two activities in the same row; the direct line is graphed when the destination activity is the next activity in the source activity's row.

For connecting dataflow lines that must span several columns, the dataflow line is graphed in the space between rows to avoid crossing through other activity markers. The

y coordinate of the line between the rows is calculated from the data structure that records the last y coordinate used in that row space (Figure 10). The reserved lines data structure is an array that contains two entries for each row space. The first entry is used when graphing a line going to a destination activity that is below the source activity. In that case, the horizontal line is graphed in the row space directly below the source activity, the y coordinates for lines are assigned proceeding from the top of the row space down towards the center of the row space.

Reserved
y coordinates

1	100
2	50
3	-50
4	-100

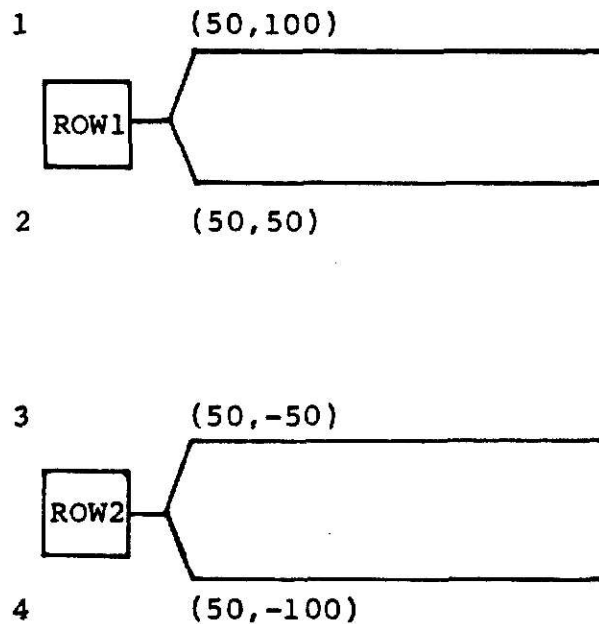


Figure 10: Reserved lines data structure

The second entry in the reserved lines data structure tracks

the y coordinates for the lines going to destination activities that are above their source activities. These lines are graphed in the row space immediately above the source activity. The y coordinates for those lines are assigned proceeding from the bottom of the row space up towards the center of the row space. The y coordinates of the two entries in the reserved lines data structure for that row space should never overlap since the Row_Spacing model has allowed enough room between rows for all of the lines.

The x coordinate for the start of the horizontal line is a fixed distance to the right of the output label line. The x coordinate for the end of the horizontal line is a fixed distance to the left of the input collection node. Once the coordinates for the horizontal line between rows are calculated, lines are graphed connecting the output label line and the input collection node to the horizontal line.

The graphics commands for all the figures in the dataflow diagram are written to an output file for display on a graphics terminal or plotter. Thus, the design provides a means to automatically produce an accurate, legible dataflow diagram.

3.5 Implementation and Testing

This system was developed on the UNIX system III Operating System at Kansas State University. The tool consists of one program written in C language. The program is approximately 2500 lines of source code. The project was designed and implemented primarily in a five-week period at Kansas State University.

The design was implemented using a modular structure which was conducive to a top-down testing approach. The control module, which orders the execution of the six main modules, was first tested with dummy modules in place of the six main modules. The six main modules were then added and tested one at a time.

As an aid in program validation, print statements were added at strategic points to display the values of internal data items. This information was used to track the control flow of each execution, to verify the correctness of the internal data structure, and to validate the coordinates written by the graphics commands.

The program was tested against requirement specifications provided by Dr. D. A. Gustafson that represented fairly complex data dataflow problems. The dimensions of the components of the dataflow diagram, for example box height

and row width, were adjusted to produce a legible dataflow diagram for a congested example.

Chapter 4 - Conclusions and Extensions

4.1 Conclusions

This tool produces a dataflow diagram that accurately and readably depicts the dataflow of the input requirement specification. The dataflow diagram may be used as a graphical aid to increase understanding of the dataflow of the system for requirements analysis. The tool is easy to use, and is completely automatic, requiring no intervention from the user. The tool is flexible in the number of activities, inputs and outputs that it can handle.

The dimensions of the components of the dataflow diagram are fairly easily adjustable. All values for the dimensions of the dataflow diagram components are implemented using the C language facility `#define`, which allows the components' dimension values to be specified once for use throughout the program. If different dimensions are desired for the boxes, lines, and spacing between boxes and lines, the dimension values could be changed quickly.

4.2 Extensions

While the automatic nature of the dataflow diagram should encourage use of the tool, it may be desirable to provide an additional tool to allow the user to modify the output of

the first tool. This facility would allow the user to modify the placement of the original dataflow diagram's activities and lines to suit the user's preference.

Another possible extension would be adding the ability to pass the dimensions of the dataflow diagram components as parameters into the program. The current implementation, while fairly easy to change, does require the program to be edited and recompiled in order to experiment with the size of the dataflow diagram components. Parameterization of those values would allow the user to "tailor make" the dataflow diagram to suit each input.

One source of potential confusion when reading a dataflow diagram produced by the current tool, is the length of the data name labels. In order to label each data line at its source activity without the text taking up an inordinate amount of space on the diagram, the data names are truncated after ten characters. In the event of two data names that are not unique within ten characters, the names will become indistinguishable. A number could be added to each data name to distinguish these cases, and a report could be provided to match the dataflow diagram data names with the input data names.

REFERENCES

- [BA79] Balzer, R. and N. Goldman, "Principles of Good Software Specification and their Implications for Specification Languages," Proceedings on the Specification of Reliable Software, April 1979, pp. 58-67.
- [BE77] Bell, T. E., D. C. Bixler and M. E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 49-60.
- [BL83] Blank, J. et al., Software Engineering: Methods and Techniques, John Wiley and Sons, New York, 1983.
- [BO85] Borgida, A. et al., "Knowledge Representation as the basis for Requirements Specification," Computer, Vol. 18, No. 4, April 1985, pp. 82-104
- [DA82] Davis, A. M., "The Design of a Family of Application-Oriented Requirements Languages," Computer, Vol. 15., No. 5, May 1982, pp. 21-28.
- [EV80] Everhart, C. R., "A Unified Approach to Software (System) Engineering," Proceedings Compsac, 1980.
- [GA77] Gane, C. and T. Sarson, Structured Systems Analysis: Tools and Techniques, IST, 1977.
- [GR82] Greenspan, S. J. et al., "Capturing More World Knowledge in the Requirements Specification", Proceedings Sixth International Conference on Software Engineering, September 1982, pp 225-234.
- [IE81] IEEE Software Requirements Guideline, July 17, 1981.

- [JE79] Jensen, R. W. and C. C. Tonies, Software Engineering, Prentice-Hall, New Jersey, 1979.
- [LE82] Levene, A. A., and G. P. Mullery, "An Investigation of Requirement Specification Languages: Theory and Practice", Computer, May 1982, pp 50-59.
- [MU79] Mullery, G. P., "CORE - A Method for Controlled Requirement Specification," Proceedings Fourth International Conference on Software Engineering, September 1979.
- [PR82] Pressman, R. A., Software Engineering: A Practitioner's Approach, McGraw-Hill, 1982.
- [R085a] Roman, G., "A Taxonomy of Current Issues in Requirements Engineering," Computer, Vol. 18, No. 4, April 1985, pp. 14-24.
- [R085b] Ross, D. T., "Applications and Extensions of SADT", Computer, Vol. 18, No. 4, April 1985, pp. 25-35.
- [R077a] Ross, D. T., "Structured Analysis (SA): A Language for Communicating Ideas", IEEE Transactions K.6 Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 16-34.
- [R077b] Ross, D. T. and K. E. Schoman, Jr., "Structured Analysis for Requirements Definition", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp .
- [RZ85] Rzepka, W. and Y. Ohno, "Requirements Engineering Environments," Computer, Vol. 18, No. 4, April 1985, pp. 9-13.
- [SC85] Sceffer, P. A. et al., "A Case Study of SREM," Computer, Vol. 18, No. 4, April 1985, pp. 47-55.

- [TE77] Teichrow, D. and E. A. Hershey, III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 41-48.
- [WI79] Wilson, M. L., "A Semantics-Based Method for Requirements Analysis and System Design", Proceedings Compsac, 1979.
- [YE82] Yeh, R. T., "Requirements Analysis - A Management Perspective", Proceedings Compsac, 1982.

Syntax Description

```

<era_spec> ::=
    <era_title> <era_body> <mode table>

<era_title> ::=
    PROCESS : <text>

<era_body> ::=
    <frame> | <frame> <era_body>

<frame> ::=
    <NL> <NL> <frame_header> <frame_body>
    | <NL> <NL> Comment : <text_lines>

<frame_header> ::=
    <i_o_data_header> : <i_o_data_name>
    | <function_header> : <CAPITAL_WORD>

<i_o_data_header> ::=
    Type | Input | Output | Input_output | Data
    | Constant | <CAPITAL_WORD>

<function_header> ::=
    Activity | Periodic_function | <CAPITOL_WORD>

<frame_body> ::=
    <relation> | <relation> <frame_body>

<relation> ::=
    <NL_B> <relation_type> : <realtion_value>

<relation_type> ::=
    keywords | input | output | required mode
    | necessary_condition | occurence | assertion
    | action | comment | media | structure | type
    | enumeration | range | units
    | subpart_is | subpart_of | uses | <WORD>

<relation_value> ::=
    <text_lines> | <structure>

<structure> ::=
    <struct> | <struct> <NL_B> : <structure>

<struct> ::=
    <name> | <text> | <name> <structure> | <text> <structure>

<name> ::=

```

```

        <mode_name> | <i_o_data_name>

<i_o_data_name> ::=
    $ <WORD> $

<mode_name> ::=
    * <WORD> *

<mode_table> ::=
    <NL> <NL> MODE_TABLE <mode_list> <initial mode>

<mode_list> ::=
    <mode> | <mode> <mode_list>

<mode> ::=
    <NL_B> Mode : <mode_name>

<initial_mode> ::=
    <NL> <NL_B> Initial_Mode : <mode_name>

<transition_body> ::=
    <NL> <NL_B> Allowed_Mode_Transitions : <transition_list>

<transition> ::=
    <NL_B> <event> : <mode_name> -> <mode_name>

<event> ::=
    <i_o_data_name>
    | <i_o_data_name> = ' <text> '
    | <function_header>

<text_lines> ::=
    <text> | <text> <text_cont>

<text> ::=
    <WORD> | <WORD> <text>

<text_cont> ::=
    <NL_B> : <text> | <NL> : <text> <text_cont>

<NL> ::=
    '0 | '0 <NL>

<NL_B> ::=
    <NL> ' '

<WORD> ::=
    <char> | <char> <WORD>

```

```
<CAPITAL_WORD> ::=
    <capital_letter> <WORD>

<char> ::=
    <lower_case_char> | <symbol>

<lower_case_char> ::=
    a | b | ... | z | 0 | 1 | ... | 9

<symbol> ::=
    # | % | & | ( | ) | ? | _

<capital_letter> ::=
    A | B | ... | Z
```

Reserved Words:

- keyboard
- crt
- internal
- secondary_storage
- NONE
- every
- mode

CONTROL MODULE

I. DESCRIPTION

This module is the main control module for the program. The control module opens and closes all files, executes the UNIX system sort to sort the `input_name_file` and the `output_name_file`, and calls the six main submodules in the following order:

- `parse_reqspec`
- `compare_inputs_outputs`
- `column_marking`
- `row_marking`
- `row_spacing`
- `create_graph`

The six main submodules are documented in subsequent module specifications.

II. INPUTS

- `era system specification`
- `input_name_file`
- `output_name_file`

III. OUTPUTS

- `input_name_file`
- `output_name_file`
- `dataflow_diagram_file`

IV. SUBMODULES

- `parse_reqspec`
- `compare_inputs_outputs`
- `column marking`
- `row marking`
- `row spacing`
- `create_graph`

PARSE_REQSPEC

I. DESCRIPTION

The `parse_reqspec` module reads the era specification file scanning for activity frames (see BNF in appendix A). The module creates a linked list of activity nodes with one node per activity in the era specification. The module parses the activity frames to identify each activity's inputs and outputs and creates a record for each input and output to be written to the `input_name_file` and the `output_name_file`, respectively.

II. INPUTS

era specification file

III. OUTPUTS

`input_name_file`
`output_name_file`

IV. SUBMODULES

<code>scan_nlnl</code>	- searches for the beginning of the next frame in the era specification
<code>parse_activity</code>	- parses an activity frame for inputs and outputs
<code>get keyword</code>	- parses the first word on a line and stores it in a key word array
<code>getname</code>	- parses the name of an activity, input or output
<code>scan_lnl</code>	- searches for the beginning of the next line
<code>crt_actnode</code>	- allocates an activity node for the current activity and links it in to the activity list
<code>crt_input_sort</code>	- creates a record for the <code>input_name_file</code>

`crt_output sort` - creates a record for the `output_name`
file

COMPARE_INPUTS_OUTPUTS

I. DESCRIPTION

The `compare_inputs_outputs` module reads the sorted `input_name` and `output_name` files, matches the input and output names, and calculates the dataflow between activities. The module creates a data structure to represent the dataflow and a list of each activity's source activities.

II. INPUTS

`input_name` file
`output_name` file

III. OUTPUTS

none

IV. SUBMODULES

<code>read_inputs_file</code>	- reads the sorted <code>input_name</code> file
<code>read_outputs_file</code>	- reads the sorted <code>output_name</code> file
<code>initial input</code>	- creates a linked list of initial input nodes
<code>match_inputs_outputs</code>	- matches input names with output names to create the dataflow data structure
<code>make_sa_list</code>	- creates a list of each activity's source activities
<code>find_sa_end</code>	- finds the final node of a source activity list
<code>final_output</code>	- creates a linked list of the output nodes
<code>link_act_to_out</code>	- creates an output list for each source activity

COLUMN_MARKING

I. DESCRIPTION

The column_marking module assigns a column number to each activity node and creates a linked list of activities in each column. Each column list is headed by a column header node, the column header nodes are linked together in order by column number.

II. INPUT

none

III. OUTPUT

none

IV. SUBMODULES

mark column	- links an activity node into a new column list
add_input_tolist	- adds a source activity to the input path list of a destination activity
search_path_list	- searches an input path list for the presence of a given activity
copy_prev path	- copies the input path list of a source activity to the input path list of its destination activity
loop_check	- searches the input path list of an activity to be column marked for a dataflow loop
unmark_prev cols	- removes an activity from its current column when it is moved to a new column
shift_col_hdrs	- reorganizes the column header node list when the last activity in a column is removed

ROW MARKING

I. DESCRIPTION

The row_marking module assigns a row number to each activity and creates linked lists of the activities in each row. Each row is headed by a row header node, the row header nodes are linked together in order by row number.

II. INPUTS

none

III. OUTPUTS

none

IV. SUBMODULES

find_output	- searches the table of column one activities for the location of a given output
add_sa	- add a source activity to the column one output table
establish_row links	- creates the linked lists of activities in each row
find_next_inrow	- finds the next activity in the row to be linked in to a row list

ROW_SPACING

I. DESCRIPTION

The row_spacing module calculates the space needed between rows to graph the dataflow lines. The module traverses the dataflow data structure to count the number of lines that will need to be graphed in the space between rows. If the line space required between any pair of rows exceeds the default line space, the space between rows is increased to accommodate the largest demand.

II. INPUTS

none

III. OUTPUTS

none

IV. SUBMODULES

none

CREATE_GRAPH

I. DESCRIPTION

The create_graph module produces a file of graphics commands that depict a dataflow diagram of the input specification. The module traverses the dataflow data structure and calculates the coordinates of the activity boxes and the dataflow lines between activity boxes. The commands to graph the dataflow diagram are written to the final output file.

II. INPUTS

none

III. OUTPUTS

dataflow diagram graphics file

IV. SUBMODULES

graph_line - writes the graphics commands to graph dataflow lines

graph_box - writes the graphics commands to graph the activity boxes

put_label - writes the graphics commands to graph the line and box labels

arrow - writes the graphics commands to graph the dataflow arrows

ccirc - writes the graphics commands to graph the input collection nodes

MAINTENANCE INFORMATION

To compile the program, use the following command:

```
cc dfd.c -lm *libs*
```

The numeric constants in the program are defined using the `#define` facility in C language so that they may be modified easily. An explanation of the numeric constants is provided below.

TEXTS - size of the text in the output dataflow diagram

INP_LABEL - length of the input line between the collection node and the activity box

RADI - radius of the input collection node circle

NEWLINE - newline character

WORD_LIMIT - maximum length allowed for a keyword in the era specification

NAME_LIMIT - maximum length allowed for an activity, input or output name in the era specification

NBR_COMMON_SAS - maximum number of common source activities allowed for an output in the row1 placement table

MAX_ROWS - maximum number of rows allowed

MAX_COLS - maximum number of columns allowed

POINT_DIST - vertical distance between the horizontal lines in the space between rows

COME_OUT - horizontal distance between the right side of an activity box and the beginning of a horizontal line in the space between rows

LABEL_LENGTH - length of the output label line coming out of an activity box

COL_WIDTH - horizontal distance between columns

BOX_HEIGHT - height of the activity boxes

BOX_WIDTH - width of the activity boxes

OUT_DIFF - horizontal distance between the left side of an activity box and the end of a horizontal line in the space between rows

CDIST - INP_LABEL length plus an arbitrary distance used to make the column width large enough for all of the data line components

UNSET - value of a y coordinate that has not been determined

ROW_WIDTH - vertical distance between rows

LABEL_DIST - vertical distance between output label lines out of an activity box

ARRHT - length of the height of an arrow head at the end of an input line

ARRBS - length of the base of an arrow head at the end of an input line

NBR_OUTPUT_COL1 - maximum number of outputs for column 1 activities

All errors, except those numbered 25 - 28, are fatal errors and will cause the program to terminate.

ERROR MESSAGES

err1 - can't open [filename]

Program was unable to open the era specification file for input.

err2 - can't open [filename]

Program was unable to open the sorted input_name file for output.

err3 - can't open [filename]

Program was unable to open the sorted output_name file for output.

err4 - can't open [filename]

Program was unable to open the sorted input_name file for input.

err5 - can't open [filename]

Program was unable to open the sorted output_name file for input.

err6 - can't open [filename]

Program was unable to open the dataflow diagram output file.

err7 - cannot allocate memory

Program could not allocate memory for an activity node.

err8 - cannot allocate memory

Program could not allocate memory for an input destination node.

err9 - cannot allocate memory

Program could not allocate memory for an initial input node.

err10 - cannot allocate memory

Program could not allocate memory for an input destination node.

err11 - cannot allocate memory

Program could not allocate memory for an output destination node.

err12 - cannot allocate memory

Program could not allocate memory for an output destination node.

err13 - cannot allocate memory

Program could not allocate memory for a source activity node.

err14 - cannot allocate memory

Program could not allocate memory for an output node.

err15 - cannot allocate memory

Program could not allocate memory for an output list node.

err16 - cannot allocate memory

Program could not allocate memory for a column header node.

err17 - no initial inputs found, cannot proceed

There were no activities having initial inputs or no inputs. The program cannot select any activities to be placed in the first column.

err18 - cannot allocate memory

Program could not allocate memory for a column header node.

err19 - cannot allocate memory

Program could not allocate memory for an input path node.

err20 - cannot allocate memory

Program could not allocate memory for an input path node.

err21 - number of columns exceeds MAX_COLS

There were more columns in the dataflow diagram than the row_table matrix could hold. The number of columns in row_table can be expanded by increasing the value of #define MAX_COLS.

err22 - number of rows exceeds MAX_ROWS

There were more rows in the dataflow diagram than the row_table matrix could hold. The number of rows in row_table can be expanded by increasing the value of #define MAX_ROWS.

err23 - number of outputs in coll_table exceeded

The number of outputs for column 1 activities exceeded the coll_table. The coll_table can be expanded by increasing the value of #define NBR_OUTPER_COL1.

err24 - number of common sa's for output exceeded

The number of common source activities for an output exceeded the coll_table limit. The coll_table limit can be expanded by increasing the value of #define NBR_COMMON_SAS.

err25 - unable to graph input line for [input name] into [activity name]

There was not enough space along the side of the activity box for [activity name] to graph all of its initial input lines.

err26 - unable to graph input line for [input name] into [activity name]

There was not enough space across the top of the

activity box for [activity name] to graph all of its initial input lines.

err27 - unable to graph output line for [output name] out of [activity name]

There was not enough space along the side of the activity box for [activity name] to graph all of its outputs.

err28 - unable to graph input line for [input line] into [activity name]

There was not enough space along the side of the activity box for [activity name] to graph all of its inputs.

To execute the dataflow diagram generator, use the following command:

```
dfdgen [file1] [file2] [file3] [file4]
```

The program name is `dfdgen`, an explanation of the files is as follows:

`file1` - input file containing the era specification to be diagramed

`file2` - work file for sorting input names

`file3` - work file for sorting output names

`file4` - output file for containing the graphics commands depicting the dataflow diagram

If errors are detected during program execution, error messages will be written to `stderr`. (See Appendix C for an explanation of error messages.)

To view the output dataflow diagram, enter UNIX graphics mode and either use command `td [file4]` to display the output on the screen or to send `[file4]` to the plotter.

```

#define TEXTS 100
#define INP_LABEL 200
#define RADII 25
#define NEWLINE '\n'
#define WORD_LIMIT 10
#define NAME_LIMIT 35
#define NBR_COMMON_SAS 10
#define MAX_ROWS 10
#define MAX_COLS 10
#define POINT_DIST 140
#define COME_OUT (LABEL_LENGTH + (LABEL_LENGTH / 3))
#define LABEL_LENGTH 800
#define COL_WIDTH (2 * LABEL_LENGTH + CDIST)
#define LINE_DIST 300
#define BOX_HEIGHT (4 * LABEL_DIST)
#define BOX_WIDTH (4 * LABEL_DIST)
#define OUT_DIFF ((COL_WIDTH - COME_OUT - CDIST) / 2 + CDIST)
#define CDIST (INP_LABEL + 300)
#define UNSET 99999
#define ROW_WIDTH (COL_WIDTH / 2)
#define LABEL_DIST 140
#define ARRHT (INP_LABEL / 2)
#define ARRBBS (LABEL_DIST / 2)
#define NBR_OUTPER_COL1 (2 * MAX_COLS)

#include <stdio.h>
#include <ctype.h>
#include "/usrb/att/specht/graf/debug.h"
#include "/usrb/att/specht/graf/errpr.h"
#include "/usrb/att/specht/graf/font.h"
#include "/usrb/att/specht/graf/gpl.h"
#include "/usrb/att/specht/graf/gsl.h"
#include "/usrb/att/specht/graf/setop.h"
#include "/usrb/att/specht/graf/util.h"
FILE *eraptr,*sortin,*sortout,*dfdout,*rtnin,*rtnout;
int eof_in=FALSE,eof_out=FALSE;
char *strcpy();
char keyword[WORD_LIMIT + 1];
char *nodename;
struct gslparm gp;
char iname[NAME_LIMIT + 1];
int c,x,y,nbr_cols=0,nbr_rows=0,nbr_acts=0,already_there,
    curr_col_no=1;
struct activity {
    char name[NAME_LIMIT + 1];
    struct outlist *olist_ptr;
    struct input *inp_ptr;
    int row;
    int acol;

```

```

    struct activity *next_inlist;
    struct activity *next_inrow;
    struct activity *next_incol;
    struct dest_source *sas;
    struct dest_source *inp_path;
    int ll_x;
    int ll_y;
    int ur_x;
    int ur_y;
    int coup_y;
    int codn_y;
    int ls_y;
    int topls_x;
    int toprs_x;
    int btls_x;
    int btrs_x;
};
struct activity *first_act = NULL;
struct activity *last_act = NULL,*aptr,*act_ptr;
struct output {
    char name[NAME_LIMIT + 1];
    struct dest_source *odest;
    struct output *next_out;
    int rowmin;
    int rowmax;
    int need_line_space;
};
struct output *last_out = NULL,*optr,*first_out=NULL;
struct input {
    char name [NAME_LIMIT + 1];
    struct dest_source *idest;
    struct input *next_input;
}
*first_inp;
struct dest_source {
    struct activity *da_ptr;
    int in_y;
    struct dest_source *next_dest;
};
struct inp_sort {
    char name [NAME_LIMIT + 1];
    char colon;
    struct activity *da;
    char nl;
}
this_inp;
struct out_sort {
    char name [NAME_LIMIT + 1];
    char colon;

```

```

        struct activity *da;
        char nl;
    }
    this_out;
    struct outlist {
        struct output *out_ptr;
        struct outlist *next_ol;
    } *olptr;
    struct column {
        int col_no;
        struct activity *fst_incol;
        struct column *next_col;
        int nbr_incol;
    } *first_col,*col_ptr;
    struct col1_table {
        struct output *op;
        struct activity *common_sa[NBR_COMMOM_SAS];
        } row_calc[NBR_OUTPER_COL1], *rptr;
    struct activity *row_table[MAX_ROWS + 1] [MAX_COLS + 1];
    int rsv_points[MAX_ROWS * 2],row_width;
    int rlines[2 * MAX_ROWS];
    main (argc,argv)
    int argc;
    char *argv[];
    {
        nodename = *argv;
        if ((eraptr = fopen (argv[1],"r")) == NULL)
        {
            fprintf (stderr,"err - can't open %s\n",argv[1]);
            exit ();
        }
        if ((sortin = fopen (argv[2],"w")) == NULL)
        {
            fprintf (stderr,"err2 - can't open %s\n",argv[2]);
            exit ();
        }
        if ((sortout = fopen (argv[3],"w")) == NULL)
        {
            fprintf (stderr,"err3 - can't open %s\n",argv[3]);
            exit ();
        }
        parse_reqspec();
        fclose(eraptr);
        fclose(sortin);
        fclose(sortout);
        system("cp inpsort newsort");
        system ("sort +0 -o inpsort inpsort");
        system ("sort +0 -o outsort outsort");
        if ((rtin = fopen(argv[2],"r")) == NULL)

```

```

    {
        fprintf (stderr,"err4 - can't open %s\n",argv[2]);
        exit ();
    }
    if ((rtnout = fopen(argv[3],"r")) == NULL)
    {
        fprintf (stderr,"err5 - can't open %s\n",argv[3]);
        exit ();
    }
    compare_inputs_outputs();
    column_marking();
    row_marking();
    row_spacing();
    if ((dfdout = fopen(argv[4],"w")) == NULL)
    {
        fprintf (stderr,"err6 - can't open %s\n",argv[4]);
        exit();
    }
    create_graph();
    fclose(rtnin);
    fclose(rtnout);
    fclose(dfdout);
}
parse_reqspec()
{
    short looping=TRUE;
    scan_nlnl();
    while (looping && (c != EOF))
    {
        get_keyword();
        if (!(strcmp(keyword,"Activity",8)))
            parse_activity();
        else if (!(strcmp(keyword,"MODE_TABLE",10)))
            looping=FALSE;
        else
            scan_nlnl();
    }
}
scan_nlnl()
{
    short scanning=TRUE;
    c = getc(eraptr);
    while (scanning && (c != EOF))
    {
        if (c == NEWLINE)
        {
            c = getc(eraptr);
            if (c == NEWLINE)
            {

```



```

                                scanning = FALSE;
                                }
                                else
                                    c=getc(eraptr);
                            }
                            else
                                c = getch(eraptr);
                        }
    }
get_keyword()
{
    int i=0;
    short scanning=TRUE;
    while (!(isalnum(c)))
    {
        c = getch(eraptr);
    }
    while (scanning && (c != EOF))
    {
        keyword[i] = c;
        ++i;
        if (i > (WORD_LIMIT - 1))
            scanning = FALSE;
        else
        {
            c = getch(eraptr);
            if (c == ' ')
                scanning = FALSE;
        }
    }
    keyword[i] = '\0';
}
parse_activity()
{
    short input_found = FALSE;
    short parsing = TRUE;
    getname();
    crt_actnode();
    while (parsing)
    {
        scan_1nl();
        c = getch(eraptr);
        if (c == ' ')
        {
            get_keyword();
            if (!(strcmp(keyword,"input",5)))
            {
                input_found = TRUE;
                getname();
            }
        }
    }
}

```

```

        crt_input_sort();
    }
    else if (!(strcmp(keyword,"output",6)))
    {
        getname();
        crt_output_sort();
    }
    else continue;
}
else
    if (c == NEWLINE)
        parsing = FALSE;
}
if (! input_found)
{
    strcpy(iname,"NONE");
    crt_input_sort();
}
last_out = NULL;
}
getname()
{
    int i = 0;
    short scanning = TRUE;
    while (!(isalnum(c)))
    {
        c = getc(eraptr);
    }
    while (scanning)
    {
        iname[i] = c;
        ++i;
        if (i > (NAME_LIMIT - 1))
            scanning = FALSE;
        else
        {
            c = getc(eraptr);
            if (!(isalnum(c)) && (c != '_'))
                scanning = FALSE;
        }
    }
    iname[i] = '\0';
}
scan_inl()
{
    short scanning = TRUE;
    c = getc(eraptr);
    while (scanning)

```

```

        {
            if (c == NEWLINE)
                scanning = FALSE;
            else
                c = getc(era_ptr);
        }
    }
crt_actnode()
{
    act_ptr = (struct activity * )
                malloc(sizeof(struct activity));
    if (act_ptr == NULL)
    {
        fprintf (stderr,"err7 - cannot allocate memory");
        exit();
    }
    ++nbr_acts;
    strcpy(act_ptr->name,iname);
    act_ptr->olist_ptr = NULL;
    act_ptr->inp_ptr = NULL;
    act_ptr->row = 0;
    act_ptr->acol = 0;
    act_ptr->next_inlist = NULL;
    act_ptr->next_inrow = NULL;
    act_ptr->next_incol = NULL;
    act_ptr->sas = NULL;
    act_ptr->inp_path = NULL;
    if (first_act == NULL)
    {
        first_act = act_ptr;
        last_act = act_ptr;
    }
    else
    {
        last_act->next_inlist = act_ptr;
        last_act = act_ptr;
    }
}
crt_input_sort()
{
    strcpy(this_inp.name,
        "
    );

    strcpy(this_inp.name,iname);
    this_inp.da = last_act;
    this_inp.nl = '\n';
    this_inp.colon = ':';
    fprintf(sortin,"%s %d\n",this_inp.name,this_inp.da);
}
crt_output_sort()

```

```

{
    strcpy(this_out.name,
           "                                     ");
    strcpy(this_out.name,iname);
    this_out.da = last_act;
    this_out.nl = 'nl';
    this_out.colon = ':';
    fprintf(sortout,"%s %d\n",this_out.name,this_out.da);
}
compare_inputs_outputs()
{
    read_inputs_file();
    read_outputs_file();
    while ((!eof_in) && (!eof_out))
    {
        if ((c = strcmp(this_inp.name,this_out.name)) > 0)
        {
            final_output();
            read_outputs_file();
        }
        else if ((c = strcmp(this_inp.name,this_out.name)) < 0)
        {
            initial_input();
            read_inputs_file();
        }
        else
            match_input_output();
    }
    if (!eof_in)
        while (!eof_in)
        {
            initial_input();
            read_inputs_file();
        }
    if (!eof_out)
        while (!eof_out)
        {
            final_output();
            read_outputs_file();
        }
}
read_inputs_file()
{
    if (fscanf(rtnin,"%s %d",this_inp.name,&this_inp.da)
        == EOF)
        eof_in = TRUE;
}
read_outputs_file()
{

```

```

    if (fscanf(rtnout,"%s %d",this_out.name,&this_out.da)
                                                == EOF)
        eof_out = TRUE;
}
initial_input()
{
    static struct input *last_inp = NULL;
    static char prev_name[NAME_LIMIT];
    static struct dest_source *last_dest = NULL;
    struct dest_source *dest_ptr = NULL;
    struct input *iptr = NULL;
    if ((c = strcmp(this_inp.name,prev_name)) == 0)
    {
        dest_ptr = (struct dest_source * )
                    malloc(sizeof(struct dest_source));
        if (dest_ptr == NULL)
            {fprintf(stderr,"err8 - cannot allocate memory\n");
             exit();
            }
        dest_ptr->da_ptr = this_inp.da;
        dest_ptr->next_dest = NULL;
        last_dest->next_dest = dest_ptr;
        last_dest = dest_ptr;
    }
    else
    {
        iptr = (struct input * ) malloc(sizeof(struct input));
        if (iptr == NULL)
            {
                fprintf(stderr,"err9 - cannot allocate memory\n");
                exit();
            }
        else
        {
            strcpy(iptr->name,this_inp.name);
            iptr->next_input = NULL;
            iptr->idest = NULL;
            dest_ptr=(struct dest_source * )
                    malloc(sizeof(struct dest_source));
            if (dest_ptr == NULL)
            {
                fprintf(stderr,
                    "err10 - cannot allocate memory\n");
                exit();
            }
            else
            {
                iptr->idest = dest_ptr;
                dest_ptr->da_ptr = this_inp.da;
            }
        }
    }
}

```

```

        dest_ptr->next_dest = NULL;
        if (first_inp == NULL)
        {
            first_inp = iptr;
            last_inp = iptr;
        }
        else
        {
            last_inp->next_input = iptr;
            last_inp = iptr;
        }
        strcpy(prev_name,this_inp.name);
        last_dest = dest_ptr;
    }
}
act_ptr=last_dest->da_ptr;
}
match_inputs_outputs()
{
    char prev_out[NAME_LIMIT];
    char prev_inp[NAME_LIMIT];
    struct dest_source *dest_ptr,*dest_list,*last_dest;
    int same_inp=TRUE,same_out=TRUE;
    final_output();
    strcpy(prev_out,this_out.name);
    strcpy(prev_inp,this_inp.name);
    dest_ptr=(struct dest_source *)
        malloc(sizeof(struct dest_source));
    if (dest_ptr == NULL)
    {
        fprintf(stderr,"err11 - cannot allocate memory\n");
        exit();
    }
    dest_list = dest_ptr;
    dest_ptr->da_ptr = this_inp.da;
    dest_ptr->in_y = UNSET;
    dest_ptr->next_dest = NULL;
    last_dest = dest_ptr;
    optr->odest = dest_ptr;
    while (same_inp)
    {
        read_inputs_file();
        if (eof_in)
            same_inp = FALSE;
        else
            if ((c = strcmp(this_inp.name,prev_inp)) != 0)
                same_inp = FALSE;
        else

```

```

    {
        dest_ptr=(struct dest_source *)
            malloc(sizeof(struct dest_source));
        if (dest_ptr == NULL)
        {
            fprintf(stderr,
                "err12 - cannot allocate memory\n");
            exit();
        }
        dest_ptr->da_ptr = this_inp.da;
        dest_ptr->next_dest = NULL;
        dest_ptr->in_y = UNSET;
        last_dest->next_dest = dest_ptr;
        last_dest = dest_ptr;
        act_ptr = last_dest->da_ptr;
    }
}

make_sa_list(dest_list,optr);
while (same_out)
{
    read_outputs_file();
    if (eof_out)
        same_out = FALSE;
    else
        if ((c = strcmp(this_out.name,prev_out)) != 0)
            same_out = FALSE;
    else
    {
        link_act_to_out();
        make_sa_list(dest_list,optr);
    }
}

}

make_sa_list(dptra,optr)
struct dest_source *dptra;
struct output *optr;
{
    struct dest_source *sa_ptr,*d,*last_dest,*find_sa_end();
    while (dptra != NULL)
    {
        sa_ptr = (struct dest_source *)
            malloc(sizeof(struct dest_source));
        if (sa_ptr == NULL)
        {
            fprintf(stderr,"err13 - cannot allocate memory\n");
            exit();
        }
        sa_ptr->da_ptr = this_out.da;
        sa_ptr->next_dest = NULL;

```

```

    aptr = dptr->da_ptr;
    if (aptr->sas == NULL)
        aptr->sas = sa_ptr;
    else
    {
        last_dest = find_sa_end(aptr->sas);
        last_dest->next_dest = sa_ptr;
    }
    d = dptr->next_dest;
    dptr = d;
}

}

struct dest_source *find_sa_end(d)
struct dest_source *d;
{
    struct dest_source *last_sa;
    while (d != NULL)
    {
        last_sa = d;
        d = last_sa->next_dest;
    }
    return(last_sa);
}

final output()
{
    static struct output *last_out;
    static char prev_name[NAME_LIMIT];
    if ((c = strcmp(this_out.name, prev_name)) != 0)
    {
        optr = (struct output *) malloc(sizeof(struct output));
        if (optr == NULL)
        {
            fprintf(stderr, "err14 - cannot allocate memory\n");
            exit();
        }
        strcpy(prev_name, this_out.name);
        strcpy(optr->name, this_out.name);
        optr->odest = NULL;
        optr->next_out = NULL;
        optr->rowmin = MAX_ROWS + 1;
        optr->rowmax = 0;
        optr->need_line_space = FALSE;
        if (first_out == NULL)
        {
            first_out = optr;
            last_out = optr;
        }
        else
        {

```



```

        last_out->next_out = optr;
        last_out=optr;
    }
    link_act_to_out();
}
link_act_to_out()
{
    struct outlist *olptr,*ol,*last_ol;
    olptr = (struct outlist *) malloc(sizeof(struct outlist));
    if (olptr == NULL)
    {
        fprintf(stderr,"err15 - cannot allocate memory\n");
        exit();
    }
    olptr->out_ptr = optr;
    olptr->next_ol = NULL;
    ol = this_out.da->olist_ptr;
    if (ol == NULL)
    {
        this_out.da->olist_ptr = olptr;
    }
    else
    {
        while (ol != NULL)
        {
            last_ol = ol;
            ol = last_ol->next_ol;
        }
        last_ol->next_ol = olptr;
    }
}
column_marking()
{
    short col1_act_found=FALSE,marking=TRUE,found_one_incol;
    struct column *cptr;
    struct activity *a,*last_incol;
    int no_loop;
    struct output *o;
    struct input *iptr,*ip;
    struct dest_source *dptr,*d;
    struct outlist *olist,*ol;
    col_ptr = (struct column *) malloc(sizeof(struct column));
    if (col_ptr == NULL)
    {
        fprintf(stderr,"err16 - cannot allocate memory\n");
        exit();
    }
    col_ptr->col_no = 1;

```

```

col_ptr->fst_incol = NULL;
col_ptr->next_col = NULL;
col_ptr->nbr_incol = 0;
first_col = col_ptr;
iptr = first_inp;
while (iptr != NULL)
{
    dptr = iptr->idest;
    while (dptr != NULL)
    {
        col1_act_found = TRUE;
        act_ptr = dptr->da_ptr;
        if (act_ptr->acol != 1)
        {
            act_ptr->acol = 1;
            if (col_ptr->fst_incol == NULL)
            {
                col_ptr->fst_incol = act_ptr;
                last_incol = act_ptr;
            }
            else
            {
                last_incol->next_incol = act_ptr;
                last_incol = act_ptr;
            }
        }
        d = dptr->next_dest;
        dptr = d;
    }
    ip = iptr->next_input;
    iptr = ip;
}
if (!col1_act_found)
{
    fprintf(stderr,
        "err17 - no initial inputs found, cannot proceed\n");
    exit();
}
cptr = first_col;
while (marking)
{
    found_one_incol = FALSE;
    col_ptr = (struct column *)
        malloc(sizeof(struct column));
    if (col_ptr == NULL)
    {
        fprintf(stderr, "err18 - cannot allocate memory\n");
        exit();
    }
}

```

```

curr_col_no++;
col_ptr->col_no = curr_col_no;
col_ptr->fst_incol = NULL;
col_ptr->next_col = NULL;
col_ptr->nbr_incol = 0;
act_ptr = cptr->fst_incol;
while (act_ptr != NULL)
{
    olptr = act_ptr->olist_ptr;
    while (olptr != NULL)
    {
        optr = olptr->out_ptr;
        dptr = optr->odest;
        while (dptr != NULL)
        {
            aptr = dptr->da_ptr;
            if (aptr->acol == 0)
            {
                found_one_incol = TRUE;
                ++(col_ptr->nbr_incol);
                mark_column(curr_col_no);
            }
            else if (aptr->acol == curr_col_no)
                mark_column(curr_col_no);
            else
            {
                no_loop = loop_check();
                if (no_loop)
                {
                    found_one_incol = TRUE;
                    unmark_prev_col();
                    ++(col_ptr->nbr_incol);
                    mark_column(curr_col_no);
                }
            }
            d = dptr->next_dest;
            dptr = d;
        }
        ol = olptr->next_ol;
        olptr = ol;
    }
    a = act_ptr->next_incol;
    act_ptr = a;
}
if (!found_one_incol)
{
    nbr_cols = cptr->col_no;
    marking = FALSE;
    free(col_ptr);
}

```

```

        }
    else
    {
        cptr->next_col = col_ptr;
        cptr = col_ptr;
    }
}
cptr = first_col;
nbr_rows = cptr->nbr_incol;
while (cptr != NULL)
{
    if (cptr->nbr_incol > nbr_rows)
        nbr_rows = cptr->nbr_incol;
    col_ptr = cptr->next_col;
    cptr = col_ptr;
}
}
mark_column(curr_col_no)
int curr_col_no;
{
    static struct activity *last_incol;
    struct dest_source *d;
    if (aptr->acol != curr_col_no)
    {
        aptr->acol = curr_col_no;
        if (col_ptr->fst_incol == NULL)
        {
            col_ptr->fst_incol = aptr;
            last_incol = aptr;
        }
        else
        {
            last_incol->next_incol = aptr;
            last_incol = aptr;
        }
    }
    if (aptr->inp_path == NULL)
    {
        d = (struct dest_source *)
            malloc(sizeof(struct dest_source));
        if (d == NULL)
        {
            fprintf(stderr, "err19 - cannot allocate memory\n");
            exit();
        }
        aptr->inp_path = d;
        d->da_ptr = act_ptr;
        d->next_dest = NULL;
        copy_prev_path();
    }
}

```

```

    }
    else
    {
        add_input_tolist(act_ptr);
        copy_prev_path();
    }
}

add_input_tolist(inp_val)
struct activity *inp_val;
{
    struct dest_source *d,*last_d,*search_path_list();
    last_d = search_path_list(a_ptr,inp_val);
    if (!already_there)
    {
        d = (struct dest_source *)
            malloc(sizeof(struct dest_source));
        if (d == NULL)
        {
            fprintf (stderr,"err20 - cannot allocate memory\n");
            exit();
        }
        d->da_ptr = inp_val;
        d->next_dest = NULL;
        last_d->next_dest = d;
    }
}

struct dest_source *search_path_list(a,inp_val)
struct activity *a,*inp_val;
{
    int searching=TRUE;
    struct dest_source *d,*last_d=NULL;
    d = a->inp_path;
    already_there = FALSE;
    if (d != NULL)
    {
        while (searching)
        {
            if (inp_val == d->da_ptr)
            {
                already_there = TRUE;
                searching = FALSE;
            }
            else
            {
                last_d = d;
                d = last_d->next_dest;
                if (d == NULL)
                    searching = FALSE;
            }
        }
    }
}

```

```

    }
    }
    return(last_d);
}
copy_prev_path()
{
    int adding=TRUE;
    struct dest_source *d,*dptr;
    if (act_ptr->inp_path == NULL)
    {
        return;
    }
    d = act_ptr->inp_path;
    while (d != NULL)
    {
        add_input_tolist(d->da_ptr);
        dptr = d->next_dest;
        d = dptr;
    }
}
loop_check()
{
    int no_loop=TRUE;
    if (act_ptr == aptr)
        no_loop = FALSE;
    else
    {
        search_path_list(act_ptr,aptr);
        if (already_there)
        {
            no_loop = FALSE;
        }
    }
    return(no_loop);
}
unmark_prev_col()
{
    struct column *tc,*c;
    int n,searching=TRUE,prev_col;
    struct activity *prev_act_incol=NULL,*curr_act_incol;
    prev_col = aptr->acol;
    tc = first_col;
    c = first_col;
    n = first_col->col_no;
    while (n != prev_col)
    {
        tc = c;
        c = tc->next_col;
        n = c->col_no;
    }
}

```

```

    }
    (c->nbr_incol)--;
    curr_act_incol = c->fst_incol;
    while (searching)
    {
        if (curr_act_incol == aptr)
            searching = FALSE;
        else
        {
            prev_act_incol = curr_act_incol;
            curr_act_incol = prev_act_incol->next_incol;
        }
    }
    if (prev_act_incol == NULL)
    {
        c->fst_incol = aptr->next_incol;
        aptr->next_incol = NULL;
    }
    else
    {
        prev_act_incol->next_incol = aptr->next_incol;
        aptr->next_incol = NULL;
    }
    if (c->fst_incol == NULL)
        shift_col_hdrs(c,tc);
}
shift_col_hdrs(c,prev_c)
struct column *c,*prev_c;
{
    struct column *curr,*next;
    int n=0;
    if (c == first_col)
        first_col = c->next_col;
    else
        prev_c->next_col = c->next_col;
    free(c);
    curr = first_col;
    while (curr != NULL)
    {
        n++;
        curr->col_no = n;
        next = curr->next_col;
        curr = next;
    }
    curr_col_no = ++n;
    col_ptr->col_no = curr_col_no;
}
row_marking()
{

```

```

struct dest_source *d,*dptr;
struct activity *sa_ptr;
struct column *cptr;
struct outlist *olptr,*ol;
int n=0,center_row,searching,prev_i,top,bot;
int op_not_there,sa_not_there,i,looping,more_acts,j;
int act_min=MAX_ROWS,act_max=1;
if (nbr_cols > MAX_COLS)
{
    fprintf(stderr,
        "err21 - number of columns exceeds MAX_COLS\n");
    exit();
}
if (nbr_rows > MAX_ROWS)
{
    fprintf(stderr,
        "err22 - number of rows exceeds MAX_ROWS\n");
    exit();
}
for (i=0;i <= MAX_ROWS;i++)
    for (j=0;j <= MAX_COLS;j++)
        row_table[i][j] = NULL;
for (i=0;i < MAX_COLS;i++)
    for (j=0;j < NBR_COMMON_SAS;j++)
        row_calc[i].common_sa[j] = NULL;
rptr = &row_calc[0];
aptr = first_col->fst_incol;
while (aptr != NULL)
{
    olptr = aptr->olist_ptr;
    if (olptr == NULL)
    {
        ++n;
        aptr->row = n;
        row_table[n][1] = aptr;
    }
    else
    {
        while (olptr != NULL)
        {
            find_output(olptr->out_ptr,
                &op_not_there);
            if (op_not_there)
            {
                rptr->common_sa[0] = aptr;
                rptr->op = (olptr->out_ptr);
                optr = olptr->out_ptr;
            }
            else

```



```

        {
            add_sa();
        }
        ol = olptr->next_ol;
        olptr = ol;
    }
    act_ptr = aptr->next_incol;
    aptr = act_ptr;
}
rptr = &row_calc[0];
looping = TRUE;
while (looping)
{
    i=0;
    more_acts = TRUE;
    while (more_acts)
    {
        aptr = rptr->common_sas[i];
        if (aptr->row == 0)
        {
            aptr->row = ++n;
            row_table[n][1] = aptr;
        }
        ++i;
        if (i > (NBR_COMMON_SAS - 1))
            more_acts = FALSE;
        else if (rptr->common_sa[i] == NULL)
            more_acts = FALSE;
    }
    ++rptr;
    if (rptr > &row_calc[MAX_COLS - 1])
        looping = FALSE;
    else if (rptr->op == NULL)
        looping = FALSE;
}
col_ptr = first_col->next_col;
while (col_ptr != NULL)
{
    aptr = col_ptr->fst_incol;
    while (aptr != NULL)
    {
        act_min = MAX_ROWS;
        act_max = 1;
        d = aptr->sas;
        while (d != NULL)
        {
            sa_ptr = d->da_ptr;
            if (sa_ptr->acol == (aptr->acol - 1))

```

```

        {
            if (sa_ptr->row < act_min)
                act_min = sa_ptr->row;
            if (sa_ptr->row > act_max)
                act_max = sa_ptr->row;
        }
        dptr = d->next_dest;
        d = dptr;
    }
    if (act_max == act_min)
        center_row = act_max;
    else
    {
        center_row = (act_max - (act_min - 1))/2;
        if ((act_max - (act_min - 1)) % 2)
            center_row++;
    }
    searching = TRUE;
    i = center_row;
    prev_i = center_row;
    top = center_row;
    bot = center_row;
    while (searching)
    {
        if (row_table[i][aptr->acol] == NULL)
        {
            searching = FALSE;
            row_table[i][aptr->acol] = aptr;
            aptr->row = i;
        }
        else
        {
            if ((prev_i > i) && (bot != (MAX_ROWS + 1)))
            {
                prev_i = i;
                i = ++bot;
            }
            else if (top != 0)
            {
                prev_i = i;
                i = --top;
            }
        }
    }
    act_ptr = aptr->next_incol;
    aptr = act_ptr;
}
cptr = col_ptr->next_col;
col_ptr = cptr;
}
establish_row_links();

```

```

}
find_output(optr,op_not_there)
    struct output *optr;
    int *op_not_there;
    {
        int searching=TRUE,i=0;
        *op_not_there = TRUE;
        rptr = &row_calc[0];
        while (searching)
            {
                if (rptr->op == optr)
                    {
                        searching = FALSE;
                        *op_not_there = FALSE;
                    }
                else
                    if (rptr->op == NULL)
                        searching = FALSE;
                    else
                        if (++i == NBR_OUTPER_COL1)
                            {
                                fprintf(stderr,"err23 - number of outputs",
                                    " in col1_table exceeded\n");
                                exit();
                            }
                        else
                            ++rptr;
            }
    }
}
add_sa()
    {
        int i=0,searching=TRUE;
        while (searching)
            {
                if (rptr->common_sa[i] == aptr)
                    searching = FALSE;
                else
                    if (rptr->common_sa[i] == NULL)
                        {
                            searching = FALSE;
                            rpтр->common_sa[i] = aptr;
                        }
                    else
                        if (++i == NBR_COMMON_SAS)
                            {
                                fprintf(stderr,"err24 - number of common ",
                                    "sa's for output exceeded\n");
                                exit();
                            }
            }
    }

```

```

    }
}
establish_row_links()
{
    int more_rows=TRUE,more_cols=TRUE,i=0,j,prev_i,prev_j;
    while (more_rows)
    {
        j = -1;
        more_cols = TRUE;
        j=find_next_inrow(i,j,&more_cols);
        prev_i = i;
        prev_j = j;
        while (more_cols)
        {
            j=find_next_inrow(i,j,&more_cols);
            if (more_cols)
            {
                aptr = row_table[i][prev_j];
                act_ptr = row_table[i][j];
                aptr->next_inrow = act_ptr;
                prev_j = j;
            }
        }
        if (++i > MAX_ROWS)
            more_rows = FALSE;
    }
}
find_next_inrow(i,j,more_cols)
int i,j,*more_cols;
{
    int searching=TRUE;
    while (searching)
    {
        ++j;
        if (j > MAX_COLS)
        {
            searching = FALSE;
            *more_cols = FALSE;
        }
        else
            if (row_table[i][j] != NULL)
            {
                searching = FALSE;
            }
    }
    return(j);
}
row_spacing()
{

```

```

int row_count[MAX_ROWS +1],i,j;
struct output *o;
struct column *cptr,*c;
struct dest_source *d,*dptr;
struct outlist *ol;
struct input *ip,*iptr;
for (i=0;i <= MAX_ROWS; i++)
    row_count[i] = 0;
for (i=0; i < (MAX_ROWS * 2); i++)
    rsv_points[i] = 0;
optr = first_out;
while (optr != NULL)
{
    dptr = optr->odest;
    while (dptr != NULL)
    {
        act_ptr = dptr->da_ptr;
        if (optr->rowmin > act_ptr->row)
            optr->rowmin = act_ptr->row;
        if (optr->rowmax < act_ptr->row)
            optr->rowmax = act_ptr->row;
        d = dptr->next_dest;
        dptr = d;
    }
    o = optr->next_out;
    optr = o;
}
cptr = first_col;
while (cptr != NULL)
{
    aptr = cptr->fst_incol;
    while (aptr != NULL)
    {
        olptr = aptr->olist_ptr;
        while (olptr != NULL)
        {
            optr = olptr->out_ptr;
            if (!optr->need_line_space)
            {
                dptr = optr->odest;
                while (dptr != NULL)
                {
                    act_ptr = dptr->da_ptr;
                    if (act_ptr->acol != aptr->acol +1)
                        if ((act_ptr->row != aptr->row) &&
                            (aptr->next_inrow != act_ptr))
                            optr->need_line_space = TRUE;
                    d = dptr->next_dest;
                    dptr = d;
                }
            }
        }
    }
}

```

```

    }
}
if (optr->need_line_space)
{
    if ((aptr->row <= optr->rowmin) &&
        (aptr->row <= optr->rowmax))
    {
        i = aptr->row;
        ++row_count[i];
    }
    else if ((aptr->row > optr->rowmin) &&
             (aptr->row > optr->rowmax))
    {
        i = aptr->row - 1;
        ++row_count[i];
    }
    else if (aptr->row <= (nbr_rows / 2))
    {
        i = aptr->row - 1;
        ++row_count[i];
    }
    else
    {
        i = aptr->row;
        ++row_count[i];
    }
}

ol = olptr->next_ol;
olptr = ol;
}
act_ptr = aptr->next_incol;
aptr = act_ptr;
}
c = cptr->next_col;
cptr = c;
}
ip = first_inp;
while (ip != NULL)
{
    dptr = ip->idest;
    while (dptr != NULL)
    {
        aptr = dptr->da_ptr;
        if (aptr->acol != 1)
        {
            i = aptr->row - 1;
            row_count[i]++;
        }
        d = dptr->next_dest;
    }
}

```

```

        dptr = d;
    }
    iptr = ip->next_input;
    ip = iptr;
}
row_width = 0;
for (i=0; i <= MAX_ROWS; i++)
    if (row_count[i] > row_width)
        row_width = row_count[i];
row_width = row_width * LABEL_DIST + 2 * LABEL_DIST;
if (row_width < ROW_WIDTH)
    row_width = ROW_WIDTH;
}
create_graph()
{
    int end_label_line_x, label_line_y, end_point_x, mid_col,
        mid_row;
    int i, j, co_assigned, consol_x, co_y, center_c;
    int coming_out_x;
    int no_errs_input;
    struct column *cptr, *c;
    struct outlist *ol;
    struct activity *a;
    struct input *iptr, *ip;
    struct dest_source *dptr, *d;
    gp.fp = dfdout;
    gp.x0 = 0.0;
    gp.y0 = 0.0;
    gp.x = 0.0;
    gp.y = 0.0;
    gp.xs = 1.0;
    gp.ys = 1.0;
    gp.ux = 0.0;
    gp.uy = 0.0;
    initgsl(&gp);
    mid_row = nbr_rows / 2;
    mid_row = nbr_rows / 2;
    if (nbr_rows % 2)
        ++mid_row;
    mid_col = nbr_cols / 2;
    if (nbr_cols % 2)
        ++mid_col;
    act_ptr = first_col->fst_incol;
    act_ptr->ll_x = 0 - ((mid_col-1) * (BOX_WIDTH+COL_WIDTH));
    act_ptr->ll_y = ((mid_row - 1) * (BOX_HEIGHT+row_width));
    act_ptr->ur_x = (act_ptr->ll_x + BOX_WIDTH);
    act_ptr->ur_y = (act_ptr->ll_y + BOX_HEIGHT);
    cptr = first_col;
    while (cptr != NULL)

```

```

{
  aptr = cptr->fst_incol;
  while (aptr != NULL)
  {
    aptr->ll_x = act_ptr->ll_x +
      ((aptr->acol-1)*(BOX_WIDTH+COL+WIDTH));
    aptr->ll_y = act_ptr->ll_y -
      ((aptr->row-1)*(BOX_HEIGHT+row_width));
    aptr->ur_x = (aptr->ll_x + BOX_WIDTH);
    aptr->ur_y = (aptr->ll_y + BOX_HEIGHT);
    graph_box();
    aptr->topls_x = aptr->ll_x;
    aptr->toprs_x = aptr->ur_x;
    aptr->btls_x = aptr->ll_x;
    aptr->btrs_x = aptr->ur_x;
    aptr->coup_y = aptr->ur_y;
    aptr->codn_y = aptr->ll_y;
    aptr->ls_y = aptr->ur_y;
    i = (aptr->row * 2) - 1;
    rlines[i] = aptr->ll_y;
    i = (aptr->row * 2) - 2;
    rlines[i] = aptr->ur_y;
    a = aptr->next_incol;
    aptr = a;
  }
  c = cptr->next_col;
  cptr = c;
}
iptr = first_inp;
while (iptr != NULL)
{
  if ((x = strcmp(iptr->name,"NONE")) != 0)
  {
    dptr = iptr->idest;
    while (dptr != NULL)
    {
      a = dptr->da_ptr;
      if (a->acol == 1)
      {
        a->ls_y = a->ls_y - LABEL_DIST;
        if (a->ls_y < a->ll_y)
        {
          fprintf(stderr,
            "err25 - unable to graph input line ");
          fprintf(stderr,"for %s into %s\n",
            iptr->name,a->name);
        }
      }
      else
      {

```



```

        end_label_line_x =
            a->ll_x - LABEL_LENGTH - ARRHT;
        graph_line(end_label_line_x,
            a->ls_y, a->ll_x, a->ls_y);
        put_label(end_label_line_x,
            a->ls_y, iptr->name);
        arrow(a->ll_x, a->ls_y, 0);
    }
}
else
{
    a->topls_x = a->topls_x + LABEL_DIST;
    if (a->topls_x > a->ur_x)
    {
        fprintf (stderr,
            "err26 - unable to graph input line ");
        fprintf (stderr, "for %s into %s\n",
            iptr->name, a->name);
    }
    else
    {
        i = (2 * a->row) - 2;
        a->coup_y = a->coup_y + LABEL_DIST;
        if (a->coup_y > rlines[i])
            rlines[i] = a->coup_y;
        end_label_line_x = a->topls_x - LABEL_LENGTH;
        graph_line(end_label_line_x, a->coup_y,
            a->topls_x, a->coup_y);
        graph_line(a->topls_x, a->coup_y,
            a->topls_x, a->ur_y);
        put_label(end_label_line_x, a->coup_y,
            iptr->name);
        arrow(a->topls_x, a->ur_y, 270);
    }
}
d = dptr->next_dest;
dptr = d;
}
}
ip = iptr->next_input;
iptr = ip;
}
cptr = first_col;
while (cptr != NULL)
{
    aptr = cptr->fst_incol;
    while (aptr != NULL)
    {
        end_label_line_x = aptr->ur_x + LABEL_LENGTH;

```

```

coming_out_x = aptr->ur_x + COME_OUT;
label_line_y = aptr->ur_y;
olptr = aptr->olist_ptr;
while (olptr != NULL)
{
    optr = olptr->out_ptr;
    label_line_y = label_line_y - LABEL_DIST;
    if (label_line_y < aptr->ll_y)
    {
        fprintf(stderr,
            "err27 - unable to graph output line");
        optr->name, aptr->name);
    }
    else
    {
        graph_line(aptr->ur_x, label_line_y,
            end_label_line_x, label_line_y);
        put_label(aptr->ur_x, label_line_y, optr->name);
        dptr = optr->odest;
        if (dptr == NULL)
            arrow((end_label_line_x + ARRHRT),
                label_line_y, 0);
        else
        {
            co_assigned = FALSE;
            while (dptr != NULL)
            {
                no_errs_input = TRUE;
                a = dptr->da_ptr;
                consol_x = a->ll_x - INP_LABEL - 2*RADI;
                if (dptr->in_y == UNSET)
                {
                    a->ls_y = a->ls_y - LABEL_DIST;
                    if (a->ls_y < a->ll_y)
                    {
                        no_errs_input = FALSE;
                        fprintf(stderr,
                            "err28 - unable to graph",
                                "input line ");
                        fprintf(stderr,
                            "for %s into %s\n",
                                optr->name, a->name);
                    }
                }
                else
                {
                    dptr->in_y = a->ls_y;
                    center_c = a->ll_x -
                        INP_LABEL - RADI;
                    ccirc(center_c, a->ls_y, RADI);
                }
            }
        }
    }
}

```

```

        graph_line((a->ll_x - INP_LABEL),
        dptr->in_y,a->ll_x,dptr->in_y);
        arrow(a->ll_x,dptr->in_y,0);
    }
}
if (no_errs_input)
{
    if (a->acol == aptr->acol + 1)
    {
        graph_line(end_label_line_x,
        label_line_y,consol_x,dptr->in_y);
    }
    else if ((a->acol > aptr->acol) &&
             (a->row == aptr->row) &&
             (aptr->next_inrow == a))
    {
        graph_line(end_label_line_x,
        label_line_y,consol_x,dptr->in_y);
    }
    else
    {
        if ((aptr->row <= optr->rowmin) &&
            (aptr->row <= optr->rowmax))
        {
            if (!co_assigned)
            {
                i = (aptr->row * 2) - 1;
                rlines[i] =
                    rlines[i] - POINT_DIST;
                co_y = rlines[i];
                co_assigned = TRUE;
                graph_line(end_label_line_x,
                label_line_y,coming_out_x,
                    co_y);
            }
        }
        else if ((aptr->row >=
                    optr->rowmin) &&
                 (aptr->row >= optr->rowmax))
        {
            if (!co_assigned)
            {
                i = (aptr->row * 2) - 2;
                rlines[i] =
                    rlines[i] + POINT_DIST;
                co_y = rlines[i];
                co_assigned = TRUE;
                graph_line(end_label_line_x,
                label_line_y,coming_out_x,

```

```

co_y);
    }
}
else
    if (aptr->row <= (nbr_rows/2))
    {
        if (!co_assigned)
        {
            i = (aptr->row * 2) - 2;
            rlines[i] =
                rlines[i] + POINT_DIST;
            co_y = rlines[i];
            co_assigned = TRUE;
            graph_line(end_label_line_x,
                label_line_y, coming_out_x,
                co_y);
        }
    }
else
{
    if (!co_assigned)
    {
        i = (aptr->row * 2) - 1;
        rlines[i] =
            rlines[i] - POINT_DIST;
        co_y = rlines[i];
        co_assigned = TRUE;
        graph_line(end_label_line_x,
            label_line_y, coming_out_x,
            co_y);
    }
}
end_point_x = a->ll_x - OUT_DIFF;
graph_line(coming_out_x, co_y,
    end_point_x, co_y);
if (coming_out_x < end_point_x)
    arrow(2 * coming_out_x -
        end_label_line_x, co_y, 0);
else
    arrow(end_label_line_x,
        co_y, 180);
ccirc(coming_out_x, co_y, 10);
ccirc(end_point_x, co_y, 10);
graph_line(end_point_x, co_y,
    consol_x, dptr->in_y);
}
}
d = dptr->next_dest;
dptr = d;

```

```

        }
        }
        ol = olptr->next_ol;
        olptr = ol;
    }
    a = aptr->next_incol;
    aptr = a;
}
c = cptr->next_col;
cptr = c;
}
}
graph_line(x1,y1,x2,y2)
    int x1,y1,x2,y2;
    {
        double larray[4];
        larray[0] = (double) x1;
        larray[1] = (double) y1;
        larray[2] = (double) x2;
        larray[3] = (double) y2;
        lines(larray,4,0,BLACK,NARROW,SOLID);
    }
graph_box()
    {
        int i = 0;
        char c;

        for (i=0; i< NAME_LIMIT;i++)
        {
            if (aptr->name[i] == '_')
            {
                aptr->name[i] = NEWLINE;
            }
        }
        textbox((double)aptr->ll_x,(double)aptr->ll_y,
            (double)aptr->ur_x,(double)aptr->ur_y,aptr->name,
            0,BLACK,NARROW,SOLID,TEXTS,BLACK,0);
    }
put_label(x,y,name)
    int x,y;
    char name[NAME_LIMIT + 1];
    {
        char short_name[10];

        x = x + 100;
        y = y + 70;
        strncpy(short_name,name,10);
        text((double)x,(double)y,short_name,0,TEXTS,BLACK,0);
    }

```

```
arrow(x,y,ang)
{
    int x,y,ang;
    {
        double arrowa[8];
        int x2,y2,x3,y3;
        arrowa[0] = x;
        arrowa[1] = y;
        if (ang == 0)
        {
            x2 = x - ARRHT;
            y2 = y - ARRBS / 2;
            x3 = x2;
            y3 = y2 + ARRBS;
        }
        else if (ang == 180)
        {
            x2 = x + ARRHT;
            y2 = y - ARRBS / 2;
            x3 = x2;
            y3 = y2 + ARRBS;
        }
        else
        {
            x2 = x - ARRBS / 2;
            y2 = y + ARRHT;
            x3 = x2 + ARRBS;
            y3 = y2;
        }
        arrowa[2] = (double) x2;
        arrowa[3] = (double) y2;
        arrowa[4] = (double) x3;
        arrowa[5] = (double) y3;
        arrowa[6] = (double) x;
        arrowa[7] = (double) y;
        lines(arrowa,8,0,BLACK,NARROW,SOLID);
    }
}

ccirc(x,y,rad)
{
    int x,y,rad;
    {
        double sx,sy,srad;
        sx = (double) x;
        sy = (double) y;
        srad = rad;
        circle(sx,sy,srad,BLACK,NARROW,SOLID);
    }
}
```

ERA Specification for Chess Program

Activity : Initialize_board
Output : chess_board

Activity : Create_special_board
Input : board_description
Output : chess_board

Activity : Retrieve_board
Input : name_of_game
Output : chess_board
Output : retrieve_move

Activity : Compute_Move
Input : chess_board
Output : chess_board
Output : computer_move
Output : status

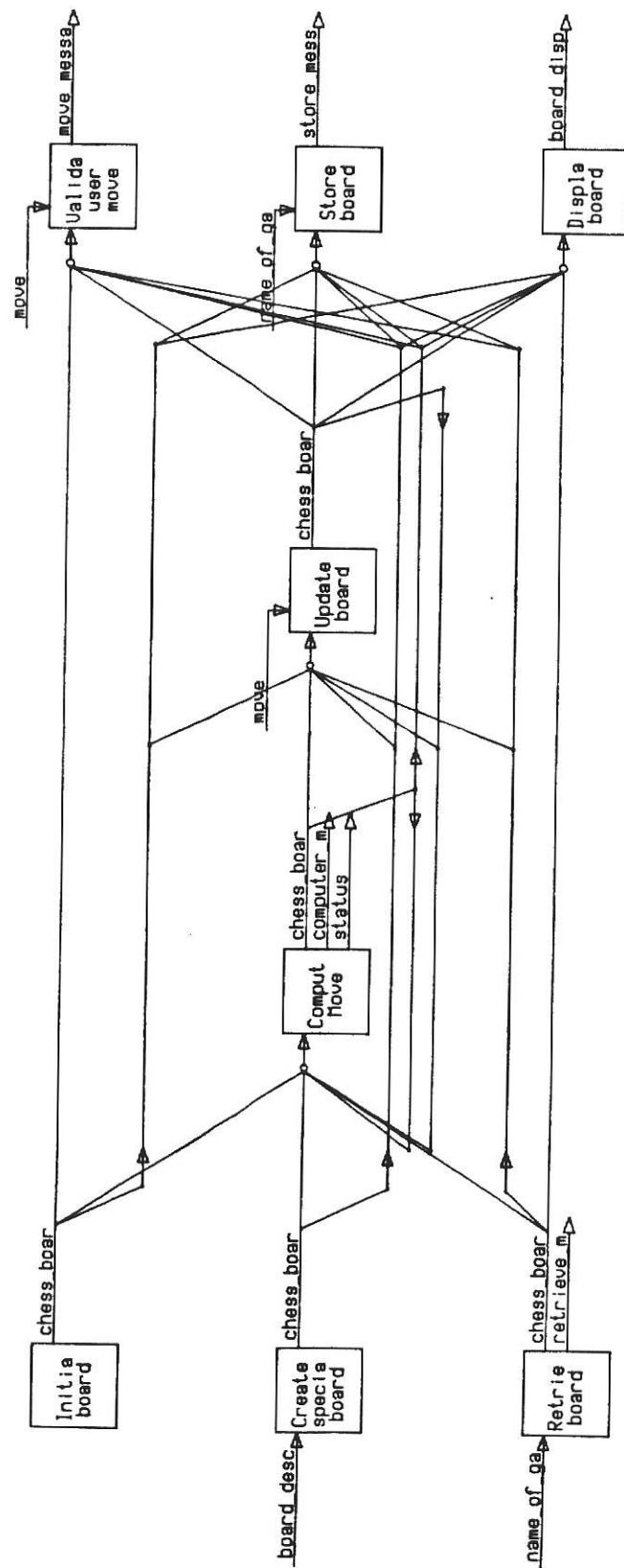
Activity : Update_board
Input : move
Input : chess_board
Output : chess_board

Activity : Validate_user_move
Input : move
Input : chess_board
Output : move_message

Activity : Store_board
Input : name_of_game
Input : chess_board
Output : store_message

Activity : Display_board
Input : chess_board
Output : board_display

Output data flow diagram for Chess Program



ERA Specification for Calculating Halstead's Metric

Activity : Print_Halstead_Heading
Output : H_table_heading

Activity : Complete_Head
Output : hk_head_msg

Activity : Store_Entity
Input : attribute
Input : entity_tab
Input : nl_n2_switch
Input : line_of_spec
Output : eof
Output : line_of_spec
Output : spec_n1_line
Output : spec_n2_line

Activity : Calc_Est_length
Input : num_spec_n1
Input : num_spec_n2
Output : est_len_msg
Output : est_length

Activity : Scan_Spec
Input : entity_tab
Input : line_of_spec
Output : eof
Output : line_of_spec

Activity : Ignore_Attribute
Input : line_of_spec

Activity : Ignore_Entity
Input : line_of_spec
Output : eof

Activity : Read_Spec_Title
Input : line_of_spec
Output : analyzer_o
Output : title

Activity : Store_Attribute
Input : fan_in_table
Input : fan_out_table
Input : nl_n2_switch
Input : verb_table
Input : line_of_spec

Output : entity_in
Output : entity_n1
Output : entity_out
Output : spec_n1_line

Activity : Print_N1N2_Heading

Input : num_spec_n1
Input : num_spec_n2
Input : spec_n1_line
Input : spec_n2_line
Output : length_n1
Output : length_n2
Output : n1n2_msg
Output : spec_length

Activity : Entity_P_Vol

Input : entity_in
Input : entity_out
Output : mod_pot_vol
Output : pot_vol_msg

Activity : Calc_Volume

Input : spec_vocab
Input : spec_length
Output : spec_vol_msg
Output : spec_volume

Activity : Calc_Est_Time

Input : num_spec_n1
Input : num_spec_n2
Output : est_time
Output : est_time_msg

Activity : Calc_HK_Metric

Input : mod_pot_vol
Input : num_ent_in
Output : hk_calc_msg
Output : hk_one
Output : hk_pot_vol
Output : num_ent_in

Activity : Comb_Pot_Vol

Input : num_mods
Input : mod_pot_vol
Output : comb_p_vol
Output : comb_pot_vol

Activity : Calc_Level

Input : comb_pot_vol

Input : spec_volume
Output : level_msg
Output : spec_prgm

Activity : Calc_Effort
Input : spec_volume
Input : spec_prgm
Output : effort
Output : effort_msg

ILLEGIBLE DOCUMENT

**THE FOLLOWING
DOCUMENT(S) IS OF
POOR LEGIBILITY IN
THE ORIGINAL**

**THIS IS THE BEST
COPY AVAILABLE**

A DATAFLOW DIAGRAM GENERATOR

by

ALICIA ELLEN SPECHT

B. A., University of Georgia, 1977

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

1986

Abstract

This project is a software tool that produces a dataflow diagram from a requirements specification. The input specification specifies the activities in the system, and each activity's inputs and outputs. The tool matches the input data names with the output data names to determine the dataflow between activities. A data structure is created to store this information. From this data structure, the tool creates a file of graphics commands that depict a dataflow diagram of the specified system.