1-1988

Graphical programming system for dataflow language

Jyun-Jier Roland Jehng

Rochester Institute of Technology
School of Computer Science and Technology

Thesis

Graphical Programming System for Dataflow Language

by
Jyun-Jier Roland Jehng

July, 1988

Rochester Institute of Technology
School of Computer Science and Technology

Graphical Programming System for Dataflow Language

by
Jyun-Jier Roland Jehng

A thesis , submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Alan R. Kaminsky 7/20/98
Prof. Alan R. Kaminsky

Donald L. Kreher July 20 1988
Dr. Donald L. Kreher

Peter G. Anderson
Dr. Peter G. Anderson

July, 1988

Title of Thesis:

Graphical Programming System for Dataflow Language

I Jyun-Jier Roland Jehng *hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.*

Date: July 22, 1988

Acknowledgements

I would like to thank my thesis chairman, Professor Alan R. Kaminsky for his guidance, time spent to ideas, returning suggestions, clearing confusions, and editing the manuscripts for my Master thesis.

I also want to thank the other member of my thesis committee, Professor Donald L. Kerher for his suggestions in testing the programs, and offering opinions.

My dear roommates Dan McDonald, Tim Iskander, and David Aikens have supported me a quiet, friendly, and confortable living and studying environment while I was devoted myself in working on the thesis. Dan, Tim, and Mr. Richard Schmidle have done proofreading on my thesis manuscripts and proposal. I feel grateful to Jimmy Ho, Joseph Chang, Sean Hwang, Jo Tsai, Min-Shin Ma, James Fan and his wife Cindy Chang, and the other friends for their encouragements and friendships.

I want to dedicate the accomplishment of my thesis to all of my families and Lillian Wang who came to my life and left. I could never have chance to complete my Master education without all the supports from my lovely father and uncle, my dearest mother, sisters and brothers, Jeff, Margie, Leslie and Martha, the Whites. You are always the most important persons in my life.

I was lucky that I was able to meet all the wonderful people at RIT. I have learned how to be hardworking, brave, and how to struggle in a very difficult situation. Now, I have accomplished one of the big jobs of my life. I am proud of it.

Abstract

Dataflow languages are languages that support the notion of data flowing from one operation to another. The flow concept gives dataflow languages the advantage of representing dataflow programs in graphical forms. This thesis presents a graphical programming system that supports the editing and simulating of dataflow programs. The system is implemented on an AT & T Unix™ PC.

A high level graphical dataflow language, GDF language, is defined in this thesis. In GDF language, all the operators are represented in graphical forms. A graphical dataflow program is formed by drawing the operators and connecting the arcs in the Graphical Editor which is provided by the system. The system also supports a simulator for simulating the execution of a dataflow program. It will allow a user to discover the power of concurrency and parallel processing. Several simulation control options are offered to facilitate the debugging of dataflow programs.

Keywords: graphical programming system, graphical dataflow language, computer graphics, concurrency, parallel processing, dataflow.

Computer Review Subject Codes:

D.1.3 Concurrent Programming
D.2.2 Tools and Techniques / Flow charts
/ User interface
D.2.6 Programming Environment
D.3.2 Language Classifications / Dataflow language
I. 3.6 Methodology and Techniques / Languages
/ Interaction techniques

Table of Contents

1. Introduction

1.1 Background

Computer technology has been developed and directed to consider how computer systems can be designed to accommodate users. One approach is human-computer interaction. It is very popular nowadays, especially in microcomputer application software design. Another approach is to use pictures to convey the ideas of both computer systems and users [Bo 1982].

The picture image -- graphical approach has many advantages over traditional textual method [Raeder 1985]. First of all, the textual method conveys the information in a sequential way, left to right and top to bottom. People might not be able to totally understand the undermeaning until they actually go over every piece of the text. Pictures provide three or more dimensions along which to lay out information, and an opportunity to use a host of properties from the physical world, such as shape, color, or direction. Second, a program is usually a complex, abstract object. When we reason about a program, we think about all these aspects in a fairly unstructured, random way; our thoughts run freely over large parts of it. Pictures are not only able to express a multitude of notions, but also provide us with direct, effortless access. All well-drawn pictures provide good metaphors, using text alone forces the reader to come up with his own mental image. Third, in conventional programming language, the only way to refer to objects is by names. This means that all objects are referenced indirectly. In the graphical approach, names are not needed because picture icons are self-explanatory. Objects are referred to by pointing to them or by connecting them, and thus references are direct and visible. The graphical approach has become a potentially powerful and useful tool in the computational environment [Glinert, Tanimoto 1984]. Interface design becomes an important role in the design of a system, and research has been done on this topic [Coutaz 1985]. People have been using pictures such as flow charts, data flows, data structures, and topology to aid their programming work. The pictures of data flow have been used mostly in connection with data flow languages and systems, and this is the focus of this thesis.

The advanced techniques in computer graphics make it possible in developing a graphical programming systems. Bitmaps are used very often. A bitmap is rectangular array of one-bit pixels, and may exist on the screen of a bitmap display or anywhere in the memory of a computer. Each pixel in a bitmap has status, ON or

OFF, which generally represent the two colors, black and white. Bitmaps are manipulated with raster operations, which are commonly optimized for speed in hardware or firmware on bitmap display devices [Thacker 1979] [Weinreb, Moon 1981].

For nearly forty years, the principles of computer design have largely remained static, based on the Von Neumann organization. Its principles include:

1. A single computing element incorporating processor, communication, and memory.
2. Linear organization of fixed-size memory cells.
3. One-level address space of cells.
4. Low-level machine language (instructions perform simple operations on elementary operands).
5. Sequential, centralized control of computation.

But the two most important characteristics of the Von Neumann model are:

1. it has a global addressable memory to hold program and data objects, with program instructions frequently updating the contents of the memory during execution, and
2. it has an instruction counter (or called program counter) which holds the address of the next instruction to be executed.

Twenty years ago, J. Rodriguer at MIT and D. Adam at Stanford began to work on research that eventually led to the development of concepts still in use today in dataflow system [Rodriguez 1969], [Adams 1968].

The difference between the Von Neumann control model and the dataflow model lies in the control of the computations [Agerwala, Arvind 1982] [Miklosko, Kotov 1984]. First for a dataflow model, only deals with values and not with names of value containers (addresses), and the language of this model has no built-in notions of gloabal updatable memory. Operators in this model produce values which are used by other operators. Second, a dataflow model has nothing like a program counter; an instruction is enabled if and only if all the required input values have been computed. An instruction in a dataflow model has no side effect, and does not introduce sequencing constraints other than the ones imposed by data dependencies in the algorithm. A programming language for a dataflow model computer must satisfy two criteria [Ackerman 1982]; it must be possible to deduce the data

dependencies of program operations; and the sequencing constraints must always be exactly the same as the data dependencies. The instruction firing rule can be based simply on the availability of data.

A control flow program is stored in memory as a sequence of instructions. Each instruction is fetched by the central processing unit according to the current address in the program counter register. An instruction can not be executed until all previous instructions have been executed. Thus the process of the computation is controlled by the sequence of instructions. An instruction (i.e. operation) in a dataflow program can be executed only when all of its operands are available. The computation in a dataflow program is controlled by the flow of data in the program. Using this method of computation, it is possible to execute as many instructions in parallel as the given computer can simultaneously handle.

The data availability firing rule of a dataflow languag is useful for modeling the concurrent execution and parallel processing. In general, dataflow languages support the notion of data flowing from one operation to another. The use of a directed graph to represent a dataflow program has been shown to be the best methodology in introducing the language [Davis, Keller 1982]. Each node in the graph stands for an instruction, an operation, or a function, and each arc between two nodes represents the medium where data flows. Before running a dataflow program, the programmer must follow a set of rules or specific techniques [Arvind, Gostelow 1982] to convert the directed graph into lines of statements. The statement representation of a program somewhat weakens the nature of proxy of the concurrency.

1.2 Scope of this Thesis

The techniques of computer graphics can be used to implement a dataflow language in a computer system with a graphics display. The work in this thesis is a combination of the techniques of computer graphics and the natural representation of dataflow languages. A Graphical Dataflow Language is defined in the thesis, and a Graphical Editor is supported for editing and simulating dataflow programs.

In the next chapter a review over the previous work on develop and research of graphical programming systems, dataflow computers, and dataflow programming languages is given. The chapter ends with a comparison between the work which has been done in this thesis and the previous work. In chapter 3, the project description

is presented. It includes the definition of the Graphical Dataflow Language, and the introduction of the Graphical Editor. In the chapter 4, several examples are given. How the system was implemented and what data structure are used in the system are explained in chapter 5. Chapter 6 contains the descriptions of the system programs. Chapter 7 gives conclusions of this thesis and a suggestion for future study. A complete bibliography and a copy of the system user's manual are attached at the end of this thesis.

2. Overview on the Previous Work

The topic of this thesis is related to the research and development of graphical programming systems and dataflow languages. In this chapter, a review of the previous work on graphical programming systems and dataflow languages is given. The projects and research presented in this section is not an exhaustive list of this subject, only the outstanding and robust projects are mentioned here.

2.1 Graphical Programming Systems

The study of graphical programming system began in the 1960's. Sutherland's "Scratchpad" was the first significant system that allowed users to construct graphs interactively. [Sutherland 1963] In the past, most interaction between human and computer has been slowed down by the need to reduce all communication to written statements that can be typed. For many types of communication, such as describing the shape of an electrical circuit, typed statements can prove cumbersome. The Scratchpad system eliminated typed statements in favor of line drawings and made man-machine conversation much more rapid. It ran on Lincoln Laboratory's TX-2 computer, equipped with a PACE plotter for printing the drawings, light pen for indicating position, knobs for rotating and magnifying picture parts, and provided several function buttons such as "draw", "move", "circle center", and "delete."

Later on, Newman's "Reaction Handler", [Newman 1968] Ellis et al.'s "GRAIL", [Ellis, Haefuer, Sibly 1969] Christensen et al.'s "Ambit/G and Ambit/L", [Christenson, Wolfberg, Fisher 1971] and Denert et al.'s "Plan2D" [Denert, Franck, Streng 1974] were built successively. The capability of developing better systems became possible because the capability of designing and manufacturing high speed processors, low cost memory, and high resolution graphics display devices became available.

David Smith's "Pygmalion" was developed in 1975, written in Smalltalk for the Xerox Alto, and designed as a demonstrational system for procedural programming in which the icon concept is central. Programming in Pygmalion is a process of designing and editing icons. With the aid of mouse, the Pygmalion programmer demonstrates the steps of a computation using sample data values; Pygmalion then replays the exact sequence of operations to perform the desired computation on other values [Smith 1975]. Gael Curry's "PAD" was implemented in XPL/G for the Xerox Sigma/5 computer equipped with an IMLAC PDS-1D vector graphics terminal and a light pen. It was also a demonstrational system for procedural programming, but it allowed user demonstrations to be carried out on potentially unbounded ranges of values, which was called abstract data, rather than actual sample data [Curry 1978].

Alan Borning's "ThingLab" was implemented in Smalltalk for the Xerox Alto. It provided programmers a set of tools to help them graphically represent experiments

in a constraint-oriented simulation laboratory, the elements in displays must conform to some relation that must be continuously maintained. Altering one value could cause any other immediately change in response according to criteria laid down by the user when the experiment is programmed. [Borning 1981] Willian Finzer and Laura Gould's "Programming by Rehearsal", announced in 1984, was implemented in Smalltalk on a Xerox Dorado computer. "Programming by Rehearsal" is a visual programming system for nonprogrammers to use to create educational software. The emphasis on the environment was on programming visually only things that can be seen can be manipulated. It provided 18 primitive performers on stage, each of which responding to about 70 cues. The process of designing and programming in this environment consists of moving 'performers' around on 'stages', and interacting with them by sending cues [Finzer, Gould 1984].

The PegaSys system developed by Mark Moriconi and Dwight F. Hare at SRI Computer Science Laboratory. The PegaSys system does not represent programs themselves in pictures, but it does use pictures for program design and documentation. Pictures are mapped into calculus formulae, and the system checks the composition of these picture elements for consistency. The system also supports a refinement methodology for the visual specifications. Pictures in PegaSys system capture a variety of concepts which are important design concepts represented in flow charts, structure charts, dataflow diagrams, and module interconnection languages. PegaSys system has been implemented on Xerox personal computer and takes advantages of a high-resolution bitmap display. [Moriconi, Hare 1986]

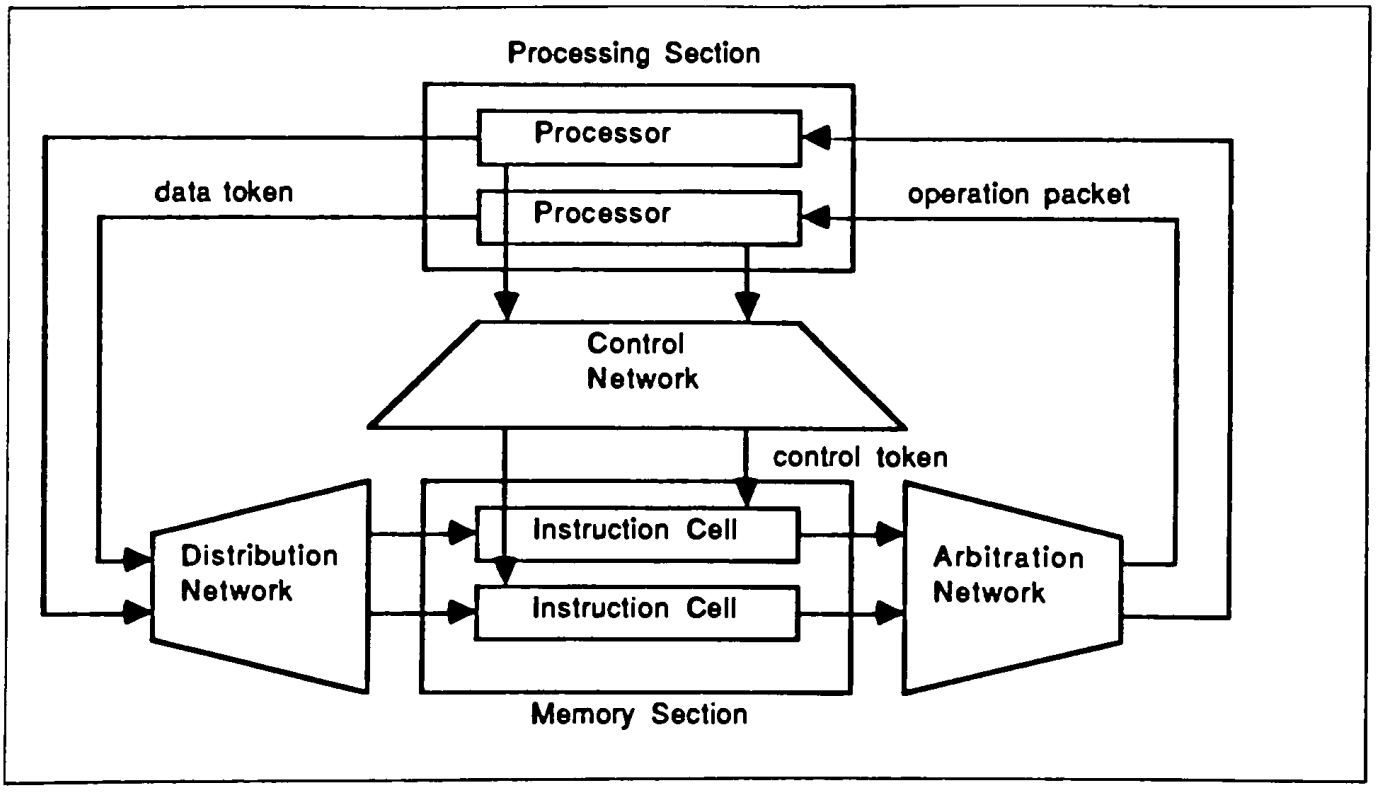
A graphical systemwhich relates to programming languages and parallel processing was created by J.C. Boarder at Oxford Polytechnic, Oxford, England in 1981. His language called LZ was developed and implemented for the ITT 2020 microcomputer and its graphics facilities. Parallel programs can be built diagrammatically and hierarchically, using the keyboard as function selector, and transformed into an interpretable code for execution in simulated parallel mode [Boarder 1981].

The invention of ICONLISP, done by G. Cattaneo, A. Guercio, S. Levialdi, and G. Tortora in 1986, presented a visual extension of an existing functional programming language LISP. Two advantages were gained: an interactive teaching of LISP properties and behavior, and the possibility of writing correct programs using icons by means of visual feedback. The system was implemented on a MacIntosh personal computer. The icons of IconLisp are made of three components, *physical component*,

logical component, and *name*. The physical component contains a graphical representation which is a natural metaphor of the semantic value of the icon. The logical component contains the LISP list. The name icon is an identifier which may be used for mnemonic and matching purpose [Cattaneo, Guercio, Levialdi, Tortora 1986].

2.2 Dataflow Computers

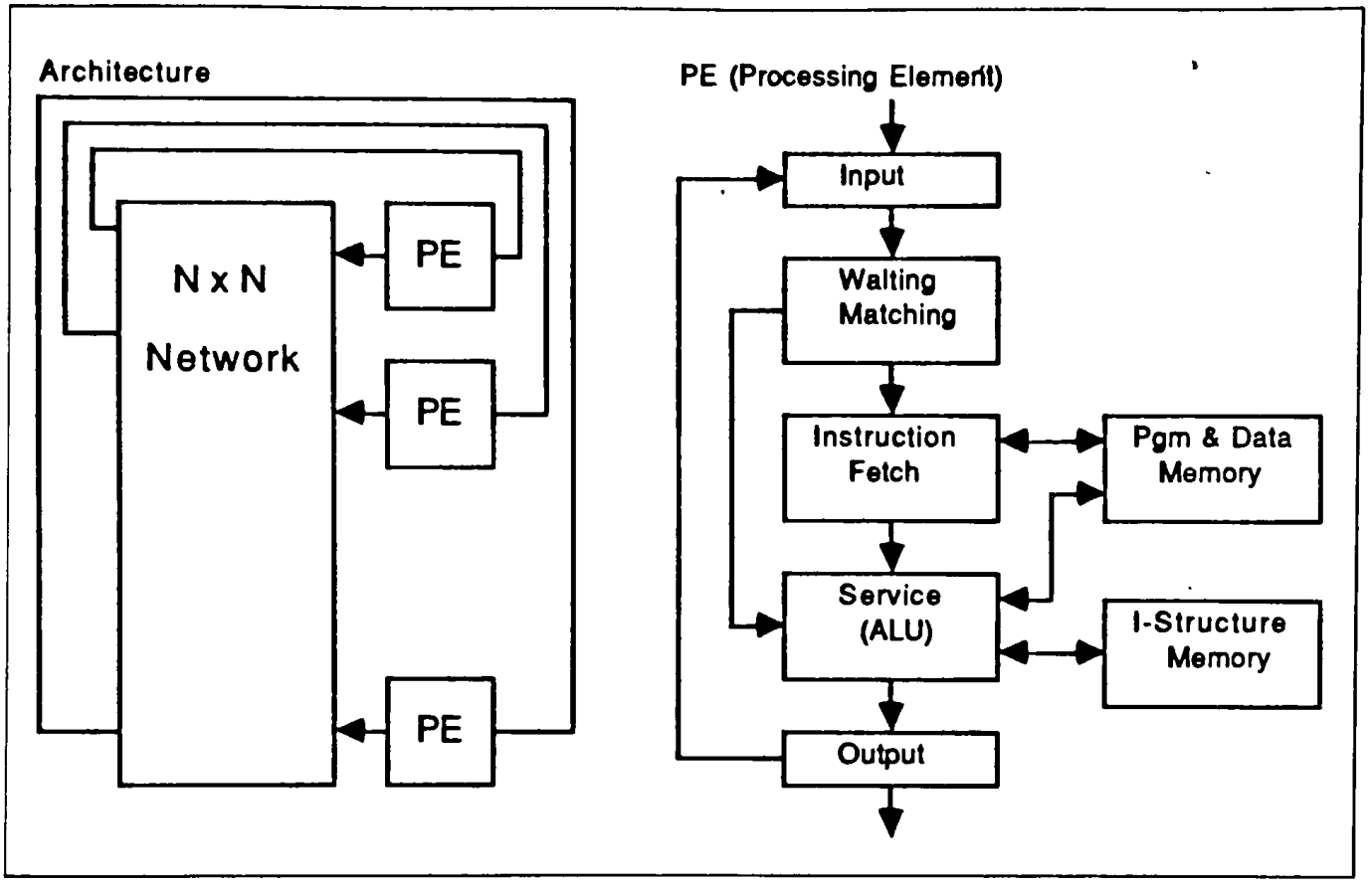
The so called Dennis Machine at MIT is one of the most well known dataflow machines. The project goals of the development [Misunas 1971] stated by Dennis were: [Misunas 1979] (1) develop a user level programming language, (2) build an engineering model, (3) address translation, oprimization, and code generation, and (4) develop specifications for a full-scale machine. The following figure shows the machine architecture [Misunas 1978] ,[Dennis, Boughton, Leung 1980]. The



machine follows a Static Execution Rule. An instruction is enabled if and only if a data token is present on all its input arcs and no token is present on its output arc.

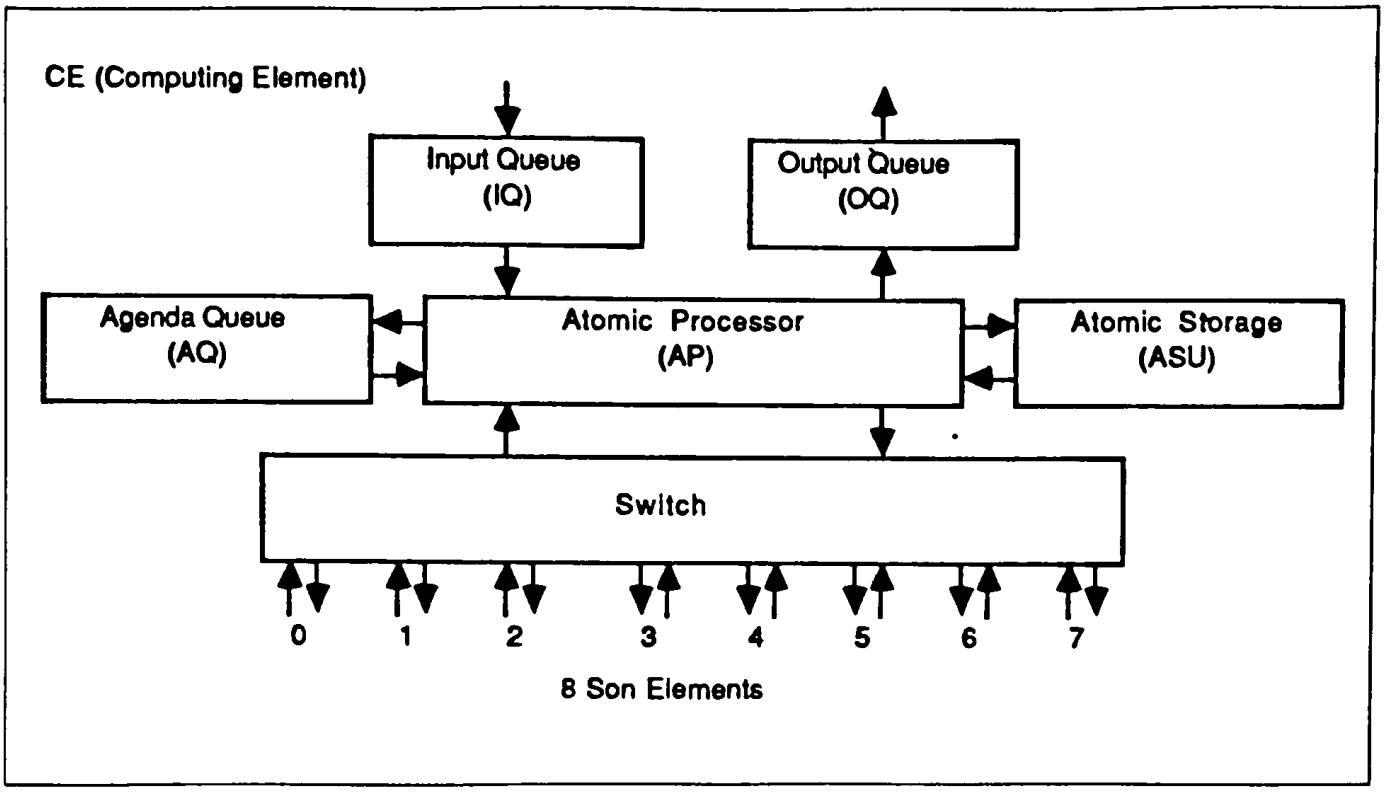
The development of the Arvind Machine at MIT was begun at the University of California-Irvine and is continuing at MIT, directed by Arvind and Gostelow. The

project goals [Misunas 1979] as stated by Gostelow were: (1) design a general-purpose computer composed of many small processors, (2) remove bottlenecks from the archithcture, (3) develop prototype based on ID (Irvine Dataflow) programming language, and (4) investigate fault tolerance. The following figure describes the architecture [Arvind, Kathail 1981] [Arvind, Gostelow 1977]. The Arvind machine



does not follow a static execution rule as does the Dennis Machine. It instead allows several tokens to be present on the input and output arcs that lead into and out of an instruction. In this way several instantiations of the same instruction (provided there are no data constraints) can execute concurrently. This kind of architecture is said to be dynamic.

The DDM1 Machine (Data Driven Machine) project at Utah University under the direction of Davis has the following goals: (1) develop a recursive machine architecture, and (2) develop a high-level data-driven graphical language [Misunas 1979]. The DDM1 architecture is shown below [Davis 1979] [Davis 1978]. DDM1 became operational in 1976 (first in the USA), the DDM1 communication with a



DEC 20/40 computer which is used for compilation, input, output, and performance measurement. The language is currently a statement description of a directed graph. An interactive graphical programming language is under development.

Some other machines are Manchester Machine at Manchester, England. Its major motivation is the exploitation of parallelism to develop a high speed machine. Secondly is the realization of cost effective and reliable design [Gurd, Kirkham, Waton 1985]. Development of Toulouse LAU Machine is taking place in Toulouse, France. The goals is to design a single assignment high level language. The language must be easy to use by non-specialists, naturally exploit parallelism in algorithms, be readable and be debuggable [Comte, Hifdi, Syre 1980].

The EDDY Machine in Japan (Experimental system for Data-Driven Processor arraY) has also recently just become operational [Hwang, Briggs 1984].

2.3 Dataflow Programming Languages

To exploit the parallelism of multiprocessor architectures, vector machines and array processor computer systems, some conventional computer languages were

adapted or extended to facilitate the use of these systems [Hansen 1975]. The Concurrent Pascal Programming Language is one of this kind. It extends the sequential programming language Pascal with concurrent programming tools called processes and monitors. A process consists of a private data structure and a sequential program that can operate on the data. One process cannot operate on the private data of another process, but concurrent processes can share certain data structures. A monitor defines a shared data structure and all the operations processes can perform on it. It also defines an initial operation that will be executed when its data structure is created. Processes cannot operate directly on the shared data. They can only call monitor procedures that have access to shared data. There are also languages which have been developed to use these parallel computation system directly [Lawrie, Layman, Baer, Randal 1975]. The Illiac IV computer consists of a control unit (CU) and 64 arithmetic processors (PEs). The CU decodes instructions and causes all the PEs simultaneously to execute arithmetic, logic, and memory fetch instructions. Primary memory consists of 128K 64-bit words divided into 64 2K word modules. Each PE is connected to one 2K module and can directly access only its own module. Since each PE has its own index registers, it can index its own module independently of the others. GLYPNIR is designed for programming the Illiac IV computer and exploits its parallel computation capability. A GLYPNIR compiler has to detect the parallel operations of the program, and the parallel execution of parallel operations. Many of these languages are unnatural in the manner in which programmers normally think about problem solving. For example, when we write a concurrent program, we must somehow represent these access rules by linear text. This limitation of written language tends to obscure the simplicity of the original structure.

Dataflow languages, were then designed for this purpose and they are now used widely in some application categories [Thacker 1979] [Weinreb, Moon 1981] [Yomaguchi, Inamoto 1986] [Matwin, Pietrzykowski 1984]. The format of dataflow languages has not been universily agreed upon. Most of the programs are usually described by graphs that have the following properties:

1. Free form side effect. This property ensures that data dependencies are the same as the sequencing constraints.

2. Locality of effect. Instructions don't have unnecessary far-reaching data dependencies.

3. Equivalence of instruction scheduling constraints with data dependencies. All the information needed to execute a program is contained in its dataflow graphs.

4. A "single-assignment" convention," A variable may appear on the left side of an assignment only once within the area of the program where it is active.

5. A somewhat unusual notation for iteration, necessitated by (1) and (4).

In this section, several dataflow languages are examined.

VAL (Value-Oriented Algorithmic Language) was developed at MIT by the Computation Structure Group, with Ackerman and Dennis the principle architects [Ackerman, Dennis 1979]. It is one of the most fully developed dataflow languages. VAL is an applicative, single assignment, side-effect free language. Consequently, all data types -- boolean, integer, character, real, arrays, and records -- are treated as mathematical values. There is a union type available where tags allow you to identify and choose from a specified set of types. All variables must be declared and strong type checking is done at compile time.

A program in VAL is a collection of separately translated modules with the following characteristics: (1) Each module contains the definition of one external function and may contain definitions of internal functions. (2) The internal function definitions are nested similar to the Pascal program format for function definition. (3) No recursion or mutual recursion is allowed. (4) Internal function can only be called from the function enclosing it. (5) external functions can be called from all modules of the program except the one defining it.

VAL includes conditional expressions, (if-then-else), iteration expressions (for-iter), and parallel iteration (for-all). Recursion is not permitted so that each function invocation would be strictly independent with no state information from one invocation to the next. VAL was designed specifically for numerical computation with the data-driven machine architecture as the primary target for translation of programs. However, the design was developed with the view of allowing the language to evolve into a general purpose language when a general purpose dataflow computer is available [Ackerman, Dennis 1979].

ID (Irvine Dataflow language) was designed and developed by the data flow project at the University of California at Irvine in the late 1970's and is the most robust of the dataflow languages at this time [Arvind, Gostelow, Plouffe 1978]. ID is a block-structured, expression oriented, single-assignment language. A program in ID is composed of a list of expressions. The language supports values, blocks, conditionals, loops, and procedure applications. In addition, it supports some new concepts such as stream, functionals, and nondeterministic programming.

In all dataflow languages, there is no concept of a memory location that is manipulated; all data types -- integer, real, boolean, string, structure, procedure definition, manager definition, manager object, programmer-defined data type and error -- are treated as mathematical values. In ID there are no explicit declarations and no type checking. Since the internal representation of the values is self-identifying. Type is associated with the value and not with the variable. Primitives are available to test the type of a value and convert it to a different type if necessary.

2.4 Compare the Thesis project with the Previous Work

The work I have done in the thesis included defining a Graphical Dataflow Language (GDL), supporting a graphical programming environment for the graphical dataflow language users to draw their dataflow programs. The environment also supplies a dataflow simulation capability for users to execute their dataflow programs in a simulated mode.

The graphical editing environment is dedicated to the editing of dataflow programs. It provides sufficient editing commands and elegant editing procedures. A user can use the mouse and the keyboard to construct dataflow program graphs. It also provides on-line help functions to help users to become familiar with the language and the system,.

The operators available in the GDL only cover the basic operations. GDL does not have loop operators, array structures, or stream manipulation capability. The data types are restricted in integer, real, character, and boolean. But GDL allows the user to have recursive functions in the programs. The essential significance between GDL and former developed dataflow languages is that all the operators in GDL are

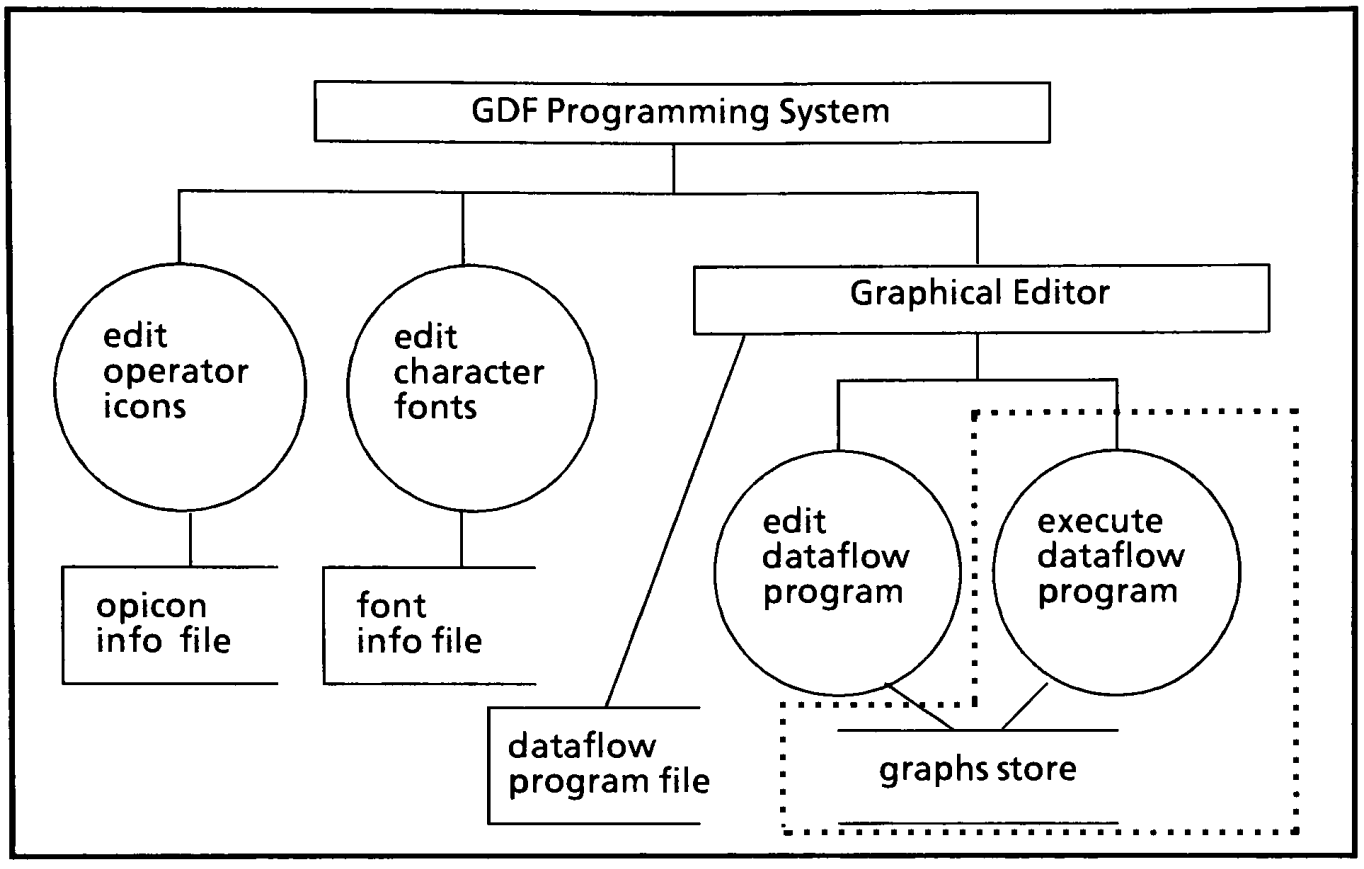
introduced in graphical forms. A dataflow program keeps its graphical representation one and for all.

The simulated execution of dataflow programs in GDL employs the Static Execution Rule. In addition, the simulator offers several control options to enable users to monitor the progresses of the execution.

3. Project Description

Several iconic programming systems and visual programming systems have been designed, which provide graphical programming environments to assist the users in programming. The GDF system developd in this thesis provides an environment for graphical programming and is dedicated to dataflow language. A graphical dataflow language is defined, and a graphical editor is designed to support the editing of the dataflow programs and the simulating of the dataflow program execution. It provides user-friendly interface between the system and the user. In this chapter, system function diagrams and the graphical dataflow language are introduced. Also, the graphical editor and the simulation of dataflow programs are described.

3.1 System Function Diagrams



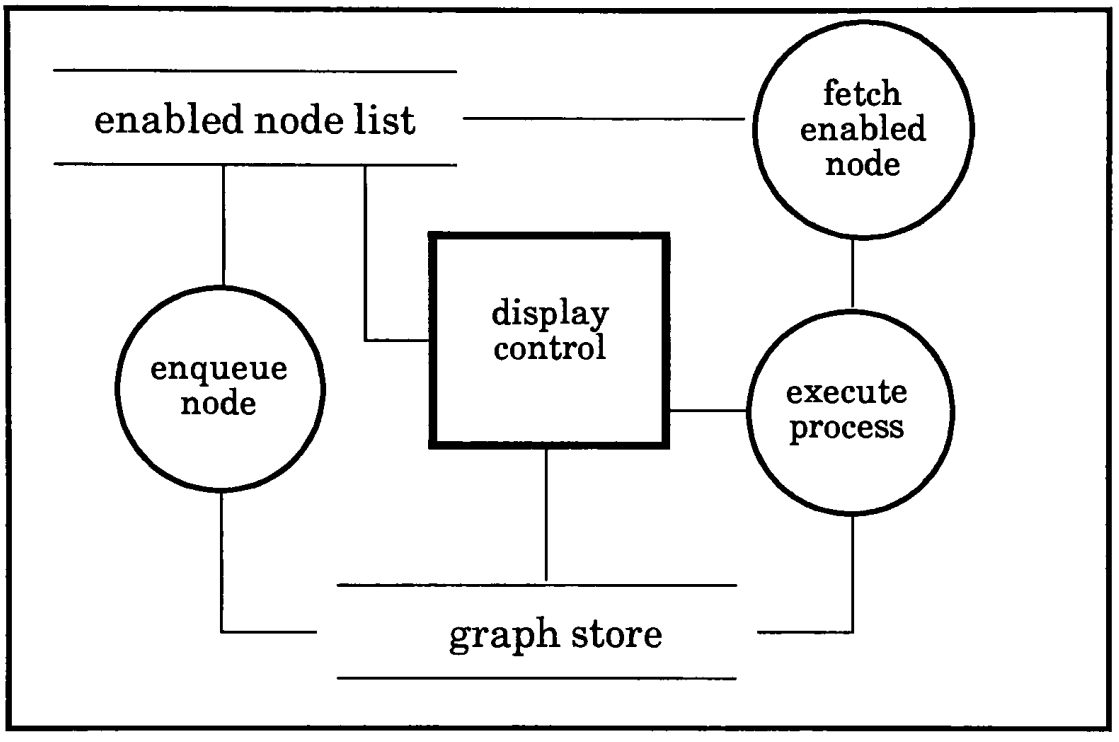
Main Function Diagram

As presented in the above diagram, the system offers users three main options:

1) edit operator icons, 2) edit character fonts, 3) edit and execute a dataflow program.

The first two options were built to facilitate the system designer in developing the system. A user is not encouraged to modify the icons or fonts until he/she has already completely understood the system architecture and data structure. All the symbols and characters used in the system (except the text displayed on the message board, the lower part of the screen which is used to show system messages) are bitmapped icons. The icon bitmaps were created and stored in two external files called *opicon info file* and *font info file.* When the GDF system is invoked, these bitmap information files are read and loaded into proper memory structure respectively. Every time the user goes through main option 1 and main option 2 to modify any of the icons, the related external bitmap info file will be updated and stored with new bitmap information.

The graphical editor is the third option. It provides commands to allow users to retrieve graph program from a file, save it to a file, interactively edit the graph on the screen, and execute the graph on the screen. The graph editor animates the graph execution simulation on the screen. Several simulation controls are also provided, they are described in more detail in section 3.3. The selected region surrounded by dash lines in the main function diagram is the simulation part of the dataflow program execution. Its functional diagram is shown below.



Funcional Diagram for Dataflow Simulation

Two data structures are shown in the diagram, the *graph store* and the *enabled node list*. The graph store keeps each graph's information in a linked list. This data structure is shared with the graphical editor. So the user can use the editor to modify the nodes or arcs in any of the graphs, then executes his program. If any error occurs, he can edit the graph and execute it again very easily. The enabled node list is also a linked list. Each element of the linked list contains a pointer which points to a node that is ready to be executed.

The three main processes are *enqueue node*, *fetch enabled node*, and *execute*. The *enqueue node* process examines each node for information on the current executed function graph in the graph store, finds out those nodes whose input tokens are ready at their input ports, and then sets up a pointer in the enabled node list to mark that the node is ready for execution. The *fetch enabled node* process fetches a node from the enabled node list, and passes the information on that node to the *execute* process. The *execute* process executes the node passed from the *fetch enabled node* process. An output token may be generated, and located at an input port of some node in the graph store. Theoretically, the *enqueue node* and the *fetch enabled node* processes are always serving the system, and many executed processes can exist at the same time. The concurrency of the system can be seen at this point. When the *fetch enabled node* process stops working, it means no more nodes are ready to be executed. The program should either terminate perfectly, or be aborted because some errors or deadlock occurred. Details of the simulation will be discussed in section 3.4.

The display control monitors the execution of a dataflow program, and provides the user with a vivid and explicit display of the execution. It dynamically highlights all the nodes which have been queued in the executed node list, and flashes the node which is being fetched and executed. The value of the tokens which flows on arcs are also displayed. The user can use the single-step function to easily find out the intermediate result from every node during program execution.

3.2 Graphical Dataflow Programming Language

Some important properties a dataflow program should have are 1) side-effect free operation, 2) locality of effect, 3) equivalence of instruction scheduling constraints with data dependencies, and 4) single assignment convention. The graphical dataflow language defined in this thesis has these properties. There are also some reasons for describing dataflow languages in graphical representations:

1) Data availability firing rule. When a node's input data are available, it is said to be fireable. The program can be represented as a directed graph where each node represents a function or an operation and each arc carries data items. The direction of the arc represents the direction of the data flow.

2) Dataflow programs are easily composable into large programs. As we know, each operation of a data flow program is driven by its input data, the operation is side-effect free. When we want to combine two pieces of the programs, the only thing we are concerned is with the output of one program and the input of the other program. This makes the composition very clear and simple.

3) Data flow programs prescribe essential data dependencies. A dependency is defined as the dependence of the data at an output arc of a node on the data at the input arc(s) of the node. So, the programs are executed concurrently. More than one node can be executed at the same time.

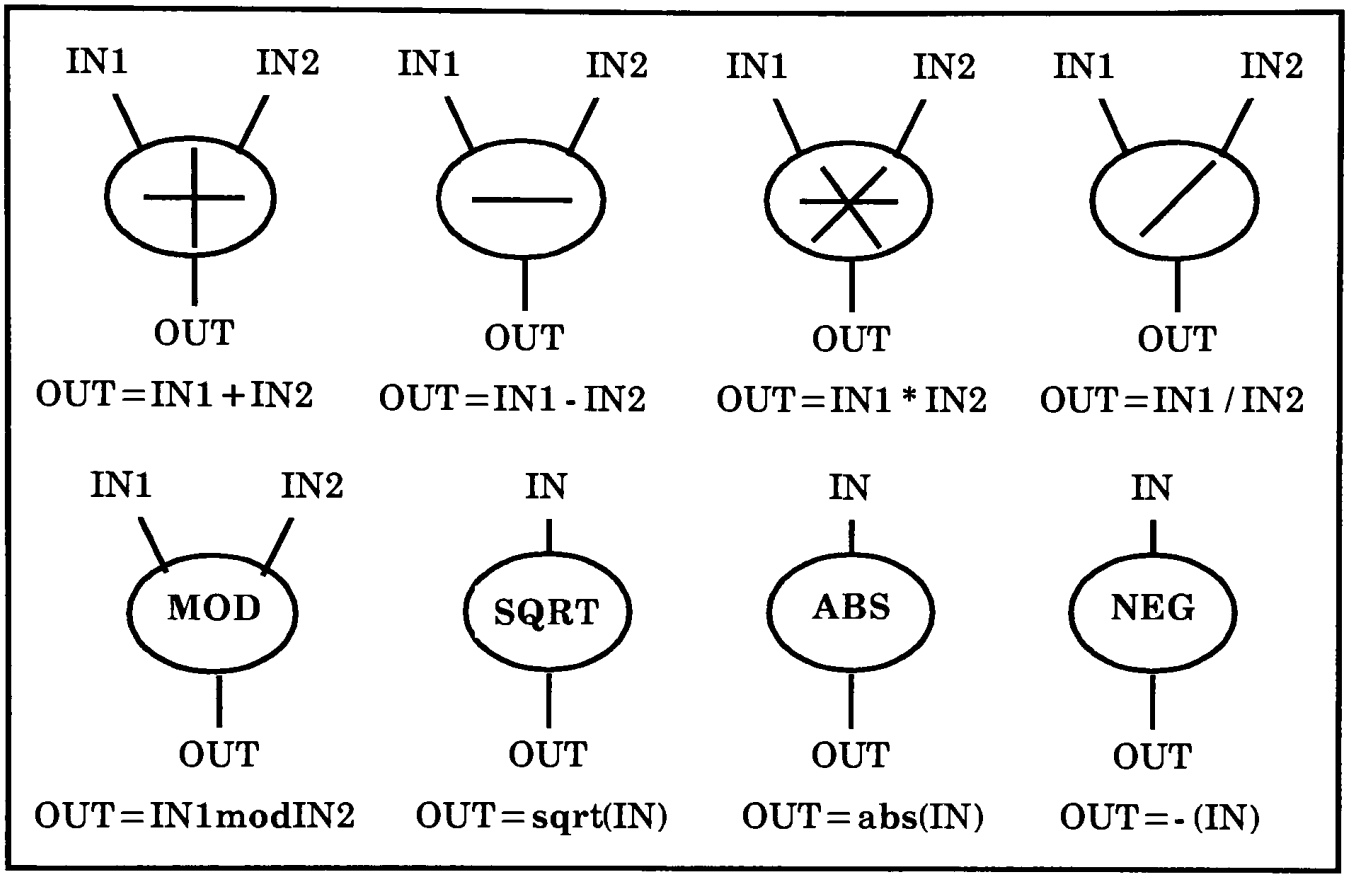
4) Graphs can be used to attribute a formal meaning to a program. The operational definition defines a permissible sequence of operations that take place when the program is executed. The functional definition describes that a single function represented by the program and is independent of any particular execution model.

These reasons are part of the motivation for establishing this system. The graphical dataflow language in this system provides 29 operators which are listed below. Their names, symbols, and functions are described in next few subsections.

ADD	SQRT	NOT	= =	DISTRIBUTE	CALL
SUB	ABS	>	/=	START	CONST
MUL	NEG	<	INPUT	STOP	ABSORBENT
DIV	AND	> =	OUTPUT	BEGIN	GATE
MOD	OR	< =	SWITCH	END	

3.2.1 Arithmetic Operators

The arithmetic operators are addition, subtraction, multiplication, division, modulo, square root, absolute value, and unary negate.

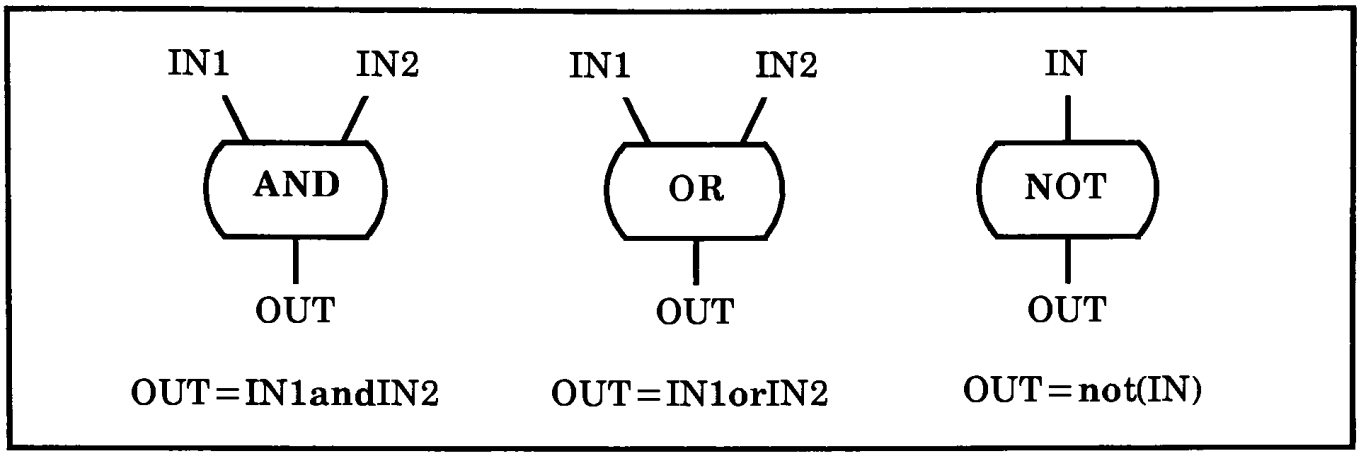


Arithmetic Operators

Proper type checking will be done on the input tokens' type. The system will display an error message when divide-by-zero occurs and terminate the execution. But some cases will not be detected, for example, if token value exceeds maximum value or goes below minimum value. The magnitude of these values is restricted by the computer architecture.

3.2.2 Logical Operators

The logical operators are logical AND, logical OR, and logical NOT.

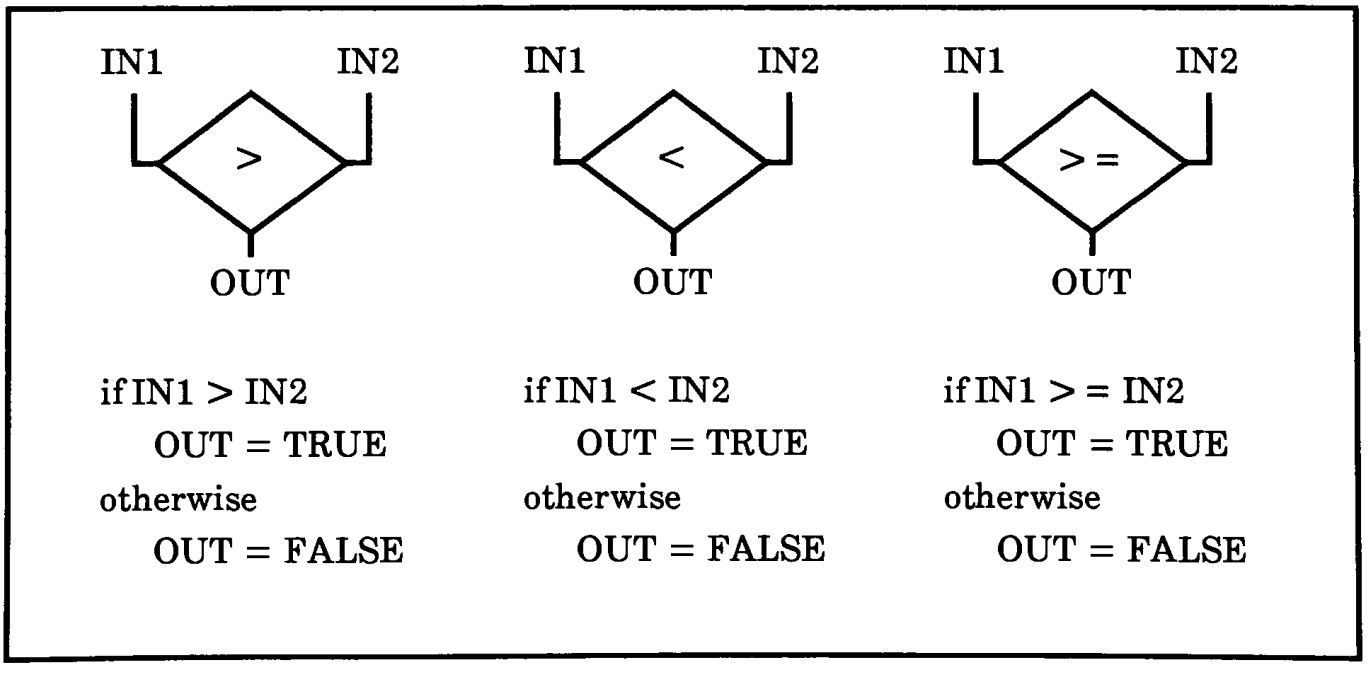


Logical Operators

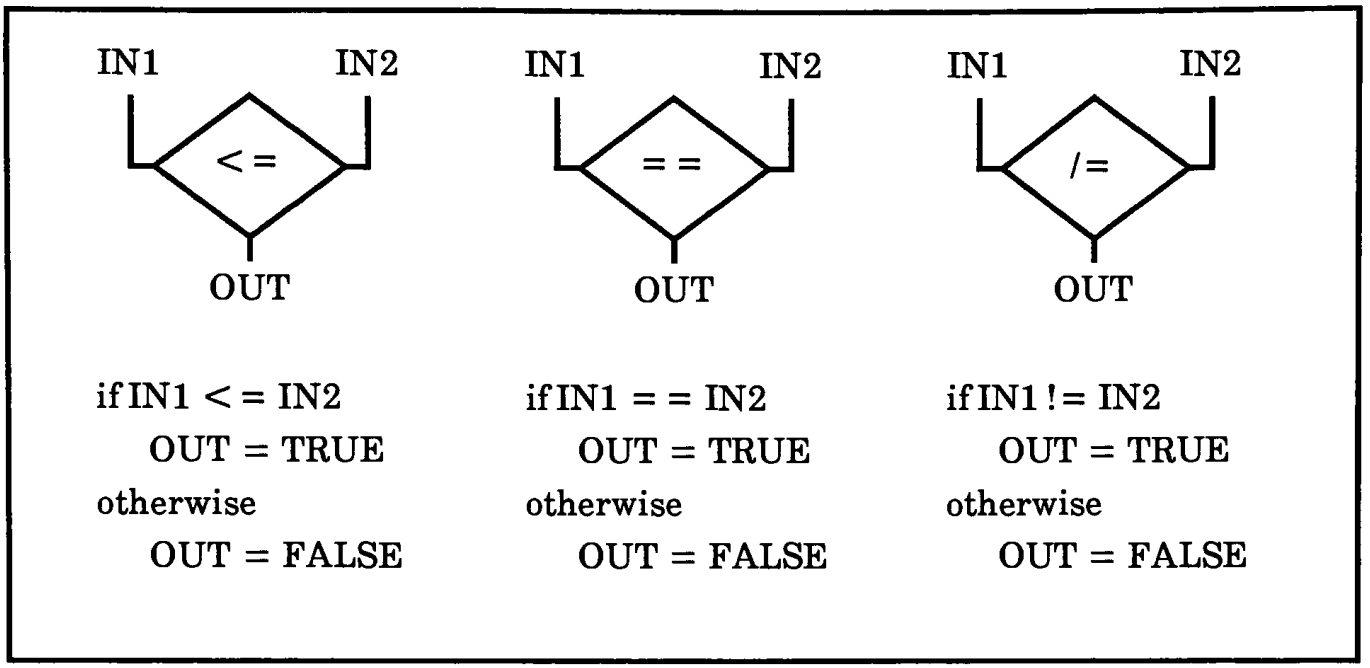
Only boolean values are allowed for the logical operations. Improper types of inputs will cause program terminated and an error message to be displayed.

3.2.3 Conditional Operators

The conditional operators are greater than, less than, greater than or equal to, less than or equal to, equal, and not equal.



To Be Continued

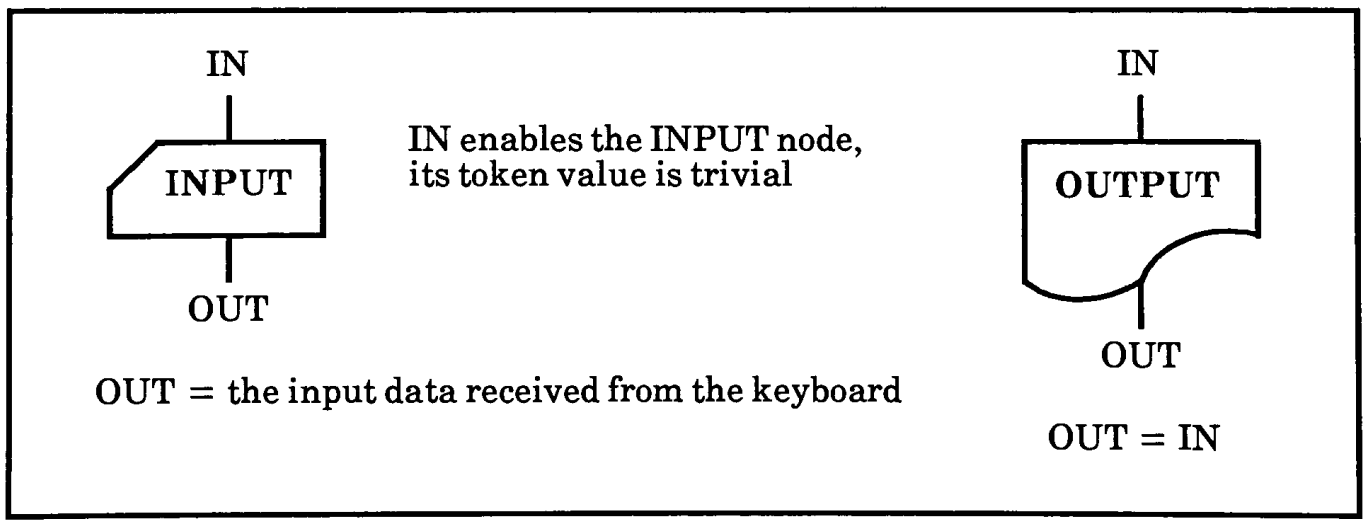


Conditional Operators

The output of a conditional operator is a boolean value, either TRUE or FALSE.

3.2.4 Input and Output Operators

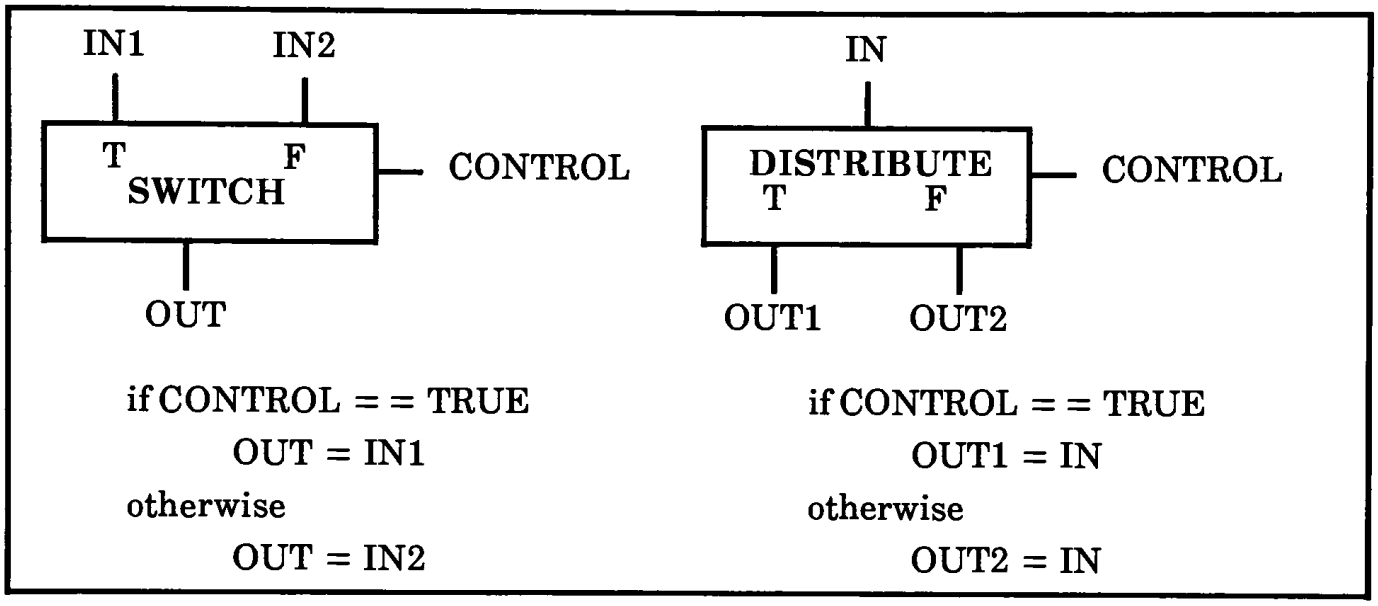
A dataflow program interacts with the user by the INPUT and OUTPUT operators.



Input/Outout Operators

Executing INPUT and OUTPUT nodes will cause the program to stop temporarily. When an INPUT node is executed, a small dialog box is created and prompt with user to key in the proper type value. The program will continue after you enter the data (then hit the <Enter> key or click [OK] box). When an OUTPUT node is executed, a small window is created and displays the output token prompt and value. The program will continue after you click the mouse or hit any key.

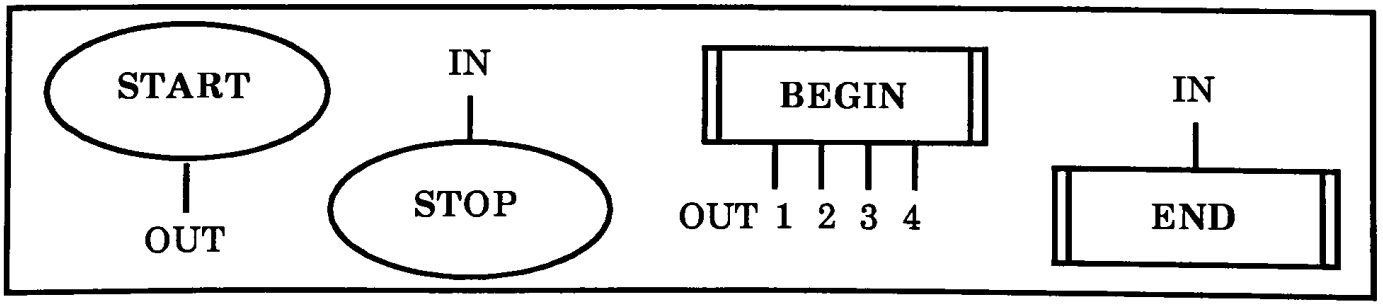
3.2.5 Switch and Distribute Operators



Switch/Distribute Operators

The SWITCH operator is a selector. One of its two input tokens is selected by the control input, and put on the output port. The DISTRIBUTE does a branching operation. The input token will flow into one of its output port which is selected by the control input.

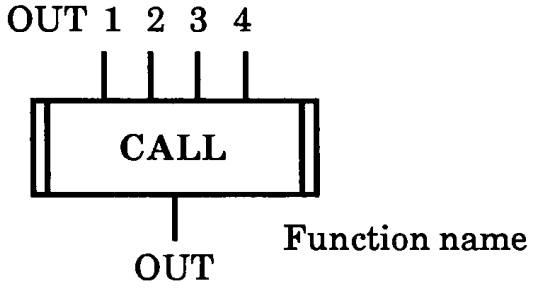
3.2.6 Start, Stop, Begin, and End Operators



Start/Stop/Begin/End Operators

The START and STOP operators are used at the beginning and the end of a main program graph respectively. The main program can only have one START node and one STOP node. The START operator generates a dummy token with NULL value, and most of the time, this token is used to enable the nodes which connected to START. The STOP operator terminates the program. BEGIN and END are used at the beginning and the end of a subroutine graph. Each port of the BEGIN node carries a token, which is from a CALL node in a calling function. The END operator returns a token to the calling function. A subroutine can only have one BEGIN operator, and as many as END operators as the user needs. After any one of the END operators is executed, the execution of subroutine is terminated and a token is returned to the calling function.

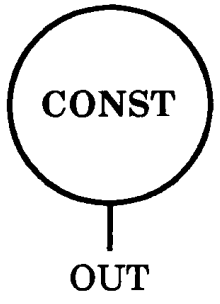
3.2.7 Call Operator



The CALL node is used when a function calls some other function. The function name is displayed at the bottom outside of the calling symbol. One function can have up to 4 input parameters, and it always returns one value.

Call Operator

3.2.8 Constant Operator

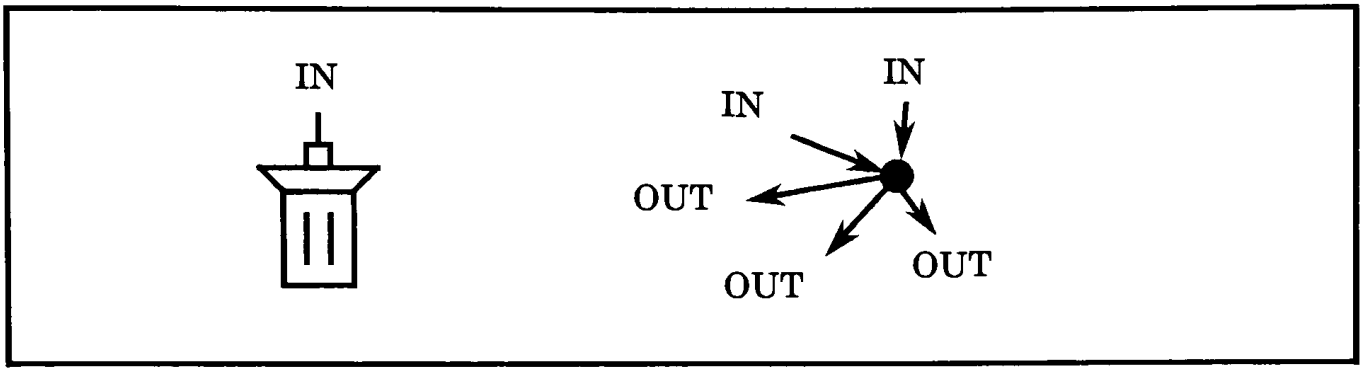


A CONST node always has a token ready on its output arc, waiting for firing. Whenever the node which is conncected to the output port of a constant node consumed its (the node's) input token, the constant node will fire.

Constant Operator

3.2.9 Absorbent and Gate Operator

An ABSORBENT node absorbs its input token which is no longer useful in the graph. A GATE node can have as many input and output arcs connected to it as you want. Whenever a token is ready on one of its input ports, the GATE is enabled to fire, and delivers the input token to all of the output ports.



Absorbent and Gate Operators

3.2.10 Types of Tokens

The GDF language has the capability to process four types of tokens. They are integer, character, boolean, and real number. The type of a token should be given when a CONST or INPUT node is selected and located on the graph scratchpad. For each of the operators except CONST and INPUT nodes, the output token type is determined by its input token. For example, if the input to an ADD operator is an integer, the output token type will be assigned as integer automatically. If its inputs are real, then the output will be a real type token. Some operators can work for mixed types. For example, the ADD, SUB, MUL, and DIV operators can do the calculation for mixed integer and real numbers. Some operators can only work for certain type of input tokens. For example, MOD operator only takes integer inputs. Some operators only generate certain types of output. For example, MOD operator generates integer type of token, and conditional operators generate boolean tokens. Some operators can take any type of input tokens, like GATE, ABSORBENT, STOP, and END.

The GDF language does some type checking. Type violation will cause a program to be terminated, and the proper error message to be displayed. The message can help the user to reassign the type (change the property) of the INPUT or CONST node, or those nodes that start with improper type of tokens. The user himself should be aware of what is the right type of token flowing on the arcs.

3.2.11 Syntax of the GDF language

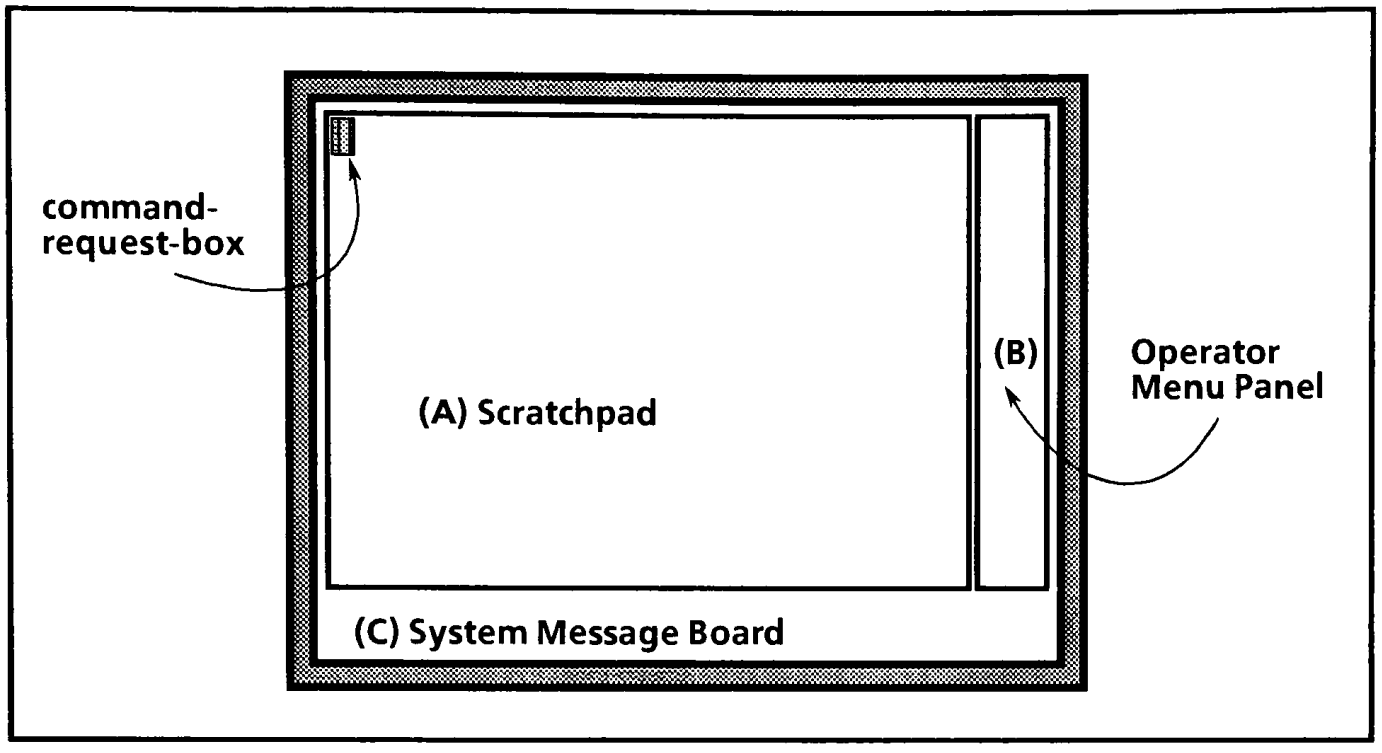
The only syntax rule that a user should obey is that all the necessary arcs should be constructed in a function or main program graph. Every input or output port of a node can only have one connection except the GATE node. The GATE node can have multiple input and output. The GDF system will check all the connections before a program is executed. The checking is only done the first time the program is executed or the first time after if is modified.

3.3 Graphical Editor

The graphical editor is a well-designed user-friendly oriented editor for editing and executing dataflow program graphs. A user can create, save, retrieve, or modify any number of function graphs. A program or function graph is composed of numbers of operators provided by the GDF language. The graphical editor supports commands for managing the graphs. The command set includes the commands for editing, file management, program execution, and on-line help. The user can see and choose a command by clicking the command-request-box which is located at the upper-left corner of the screen, and moving the mouse up and down on the pull-down command menu. Releasing the mouse button on the highlighted command will cause the command to be selected. In this section, these commands will be introduced.

3.3.1 Screen Layout

The screen is divided into three areas in the graphical editor. The figure is shown on the next page.



Screen Layout

Area (A) is called *scratchpad.* The user can construct his/her dataflow program by copying operators from the *operator menu panel*, locating them on the *scratchpad*, and connecting the arcs if necessary. Each operator on the scratchpad is a node which can be a simple operation or a call to another function which is an independent dataflow graph. The scratchpad area is also used for displaying the animation of the execution of a dataflow program. On the upper-left corner of the scratchpad is a small box called command-request-box. A pull-down menu will show up if the user clicks the mouse on the command-request-box. To keep pressing the mouse button and moving the mouse up and down inside the menu, the user can see that the command which pointed by the mouse is highlighted, and he/she can select the command by releasing the mouse button when the command is highlighted.

Area (B) is called operator menu panel. The graphical representations of the operators which provided by the system are displayed in this region for the user to choose and copy to the scratchpad to construct a program graph. There are twenty-nine operators available in the system, but only seven of them are shown on the menu panel at a time. These seven operator icons form a template. The user can click the bottom-most icon which stands for "more" operation to see more operators in another template.

Area (C) is called system message board. Various of system messages and hints which relate to the system operations, current editing status, or current execution animation status are displayed in this region. The name of the current selected command and the name of current edited graph are always displayed when a graph is under construction.

3.3.2 Graph Editing Commands

Seven system commands are provided to facilitate the editing of a dataflow program graph. They are *copy__locate*, *select*, *move*, *delete*, *connect*, *property*, and *redraw*. The function of each command is described below:

Copy- locate

This command enables a user to copy one of the operator icons which are currently shown in the operator menu panel, and locate the copied operator in the scratchpad area.

Select

This command allows a user to select one of the nodes or arcs currently displayed in the scratchpad area. The selected item will be highlighted. A user must use this command before he uses *delete*, *property*, or *move* command.

Move

This command enables a user to move one of the operators in the current edited graph around in the scratchpad area to find a pertinent location.

Delete

This command enables a user to delete any of the nodes or arcs currently displayed in the scratchpad area.

Connect

The *connect* command enables the user to construct the arcs of a dataflow graph. After the *connect* command is selected, the user can connect as many arcs as he/she wants; each arc is formed by connecting an output port of a node to an input port of a node.

Property

This command is useful in checking or changing the property for INPUT node, OUTPUT node, CONST node, and CALL node. The property of an INPUT

node is the type of the token that the INPUT node will generates and the prompt string that will be used when the INPUT node is executed. The property of an OUTPUT node is the prompt string that will be used when the OUTPUT node is executed. The property of a CONST node is the type and the value of the token that will be generated when the CONST node is executed. The property of the CALL node is the name of the function graph which is to be called when the CALL node is executed.

Redraw

This command is used to redraw the screen. The reason for providing this command is because that improper mouse operations might cause some garbage or unexpected objects left on the screen.

3.3.3 Graph File Management Commands

The GDF programming system also provides five system commands for user to manage their dataflow program graphs. These commands include new, functions, switch_pages, save, and retrieve. A dataflow program can be composed of one or more than one graph. The first edited graph of the program is called *main graph* (just like the *main* function in a C program).

New

This command can be used when the user wants to start to draw a new dataflow program. The user will be prompted to enter a name for the new program. This name is also used as the name of the main function graph name.

Functions

This command is used when a program is currently edited. It allows the user to add new function into the current edited program, so the user can draw as many new function graph as he/she wants. The *functions* command also allows the user to delete an existing function graph from the program, or rename an existing function graph of the program.

Switch-pages

If the current edited program contains more than one function graph, this command allows the user to switch the current edited graph among all the graphs.

Save

This command enables the user to save a program which may consist of any number of function graphs. The user can save the program after editing it, or before leaving the GDF programming system, or any time he/she wants to save. The user also has the options to choose either quitting the current session or resuming after the *save* operation is completed.

Retrieve

The retrieve command allows the user to retrieve a GDF language program from the current directory. The main graph of the program is then displayed on the screen. If the user wants to retrieve an non-existing program, proper error message will be given.

In the GDF programming system, only one program can be edited at a time. The *new* and *retrieve* commands can only be used when the first time the user enters the GDF programming system, or after the user quits the current editing session.

3.3.4 Execute, On-line Help, and Quit Commands

Execute

This command can be issued when the user wants to execute the current edited program. If the program is a new edited one, or an old one but has not been executed since last modification, the dataflow simulator will check the arc connections of every graph before the program is executed. Several execution simulation options are provided, and will be discussed in section 3.4.

On-Line Help

The *on-line help* command is provided to invoke the on-line help function. A brief explanation for the previous chosen event will be displayed in a newly created temporary window. The previous chosen event might be an operator or a command that the user just selected before he/she selects the *on-line help* command. No on-line help supplies to itself and the *quit* command.

Quit

Issuing the *quit* command will stop the current editing session. If there is a program in editing, the system will prompt a confirmation message to the user. The user may cancel the *quit* command, save the program then try

quitting again, or can just quit without concerning the modification or saving program.

3.3.5 Two Ways for Choosing Command

A user can select a command either by clicking the command-request-box or by pressing <B2>, the mouse command button. The former method is to select the command through a pull-down menu. The later one is especially useful when the user is trying to make changes or modifications on an existing program graph. The details about how to use both methods are described in the USER'S MANUAL.

3.4 Dataflow Simulation

The objective of the GDF Programming System is to support a dynamic simulation environment for dataflow programs. The graphs of a dataflow program are taken and executed directly by the system. The "directly" means that GDF system does not convert the graph information into an assembly-like or other programming language codes. One of the differences between the GDF simulation and other simulations which have been developed is that the GDF simulation comes along with a dynamic graphical display and GDF simulation has control options over the simulation. These control options are offered for the user to monitor the execution of dataflow programs. The concurrency can be seen very easily through the graphical display. Another difference is that a program in GDF programming system is expressed in graphs, not in conventional text statement.

3.4.1 The Simulation

At the beginning of the running a GDF program, the *main function graph* of the program is refreshed onto the screen and executed. As mentioned in section 4.1, two structures are maintained in the system, the *graph store* and the *enabled nodes list.* The graph store is a block of memory contains graphs' information. Each graph owns two linked list data structures, one for the nodes and one for the arcs. In the nodes linked list, each node element has a number of fields which are pointers and prepared for receiving and delivering tokens. When a function graph is called to be executed, the system refreshes the screen with that graph and allocates memory for duplicating the graph's information. All the duplicated graphs are stacked as run-

time activation records. The system maintains a stack pointer to keep track of the newly-called function graph. Whenever a function graph returns a token to its calling function graph, or the function graph terminates execution (if it is a *main graph*), the memory occupied by the function is released.

Three kind of nodes can fire (e.g. be executed) at the beginning of the execution of a function graph. They are START node, BEGIN node, and CONST node. The START node and BEGIN node only fire once, and the CONST node can keep firing whenever the firing is applicable. The status of each node in the node linked list is checked, that is capable of firing the node when all the necessary data tokens have arrived, and generating the output token to update the status of the destination node. After a node fires, its input tokens are released at its input ports. The output token generated by the operator only depends on the input token value and the operation of the node. Such a token generation method complies with the locality of effect, side-effect free, and single assignment rule of the general dataflow conventions.

The values of the input token and output token are displayed on the side of input and output ports when they are generated, and are erased when the tokens are consumed. The enabled node list is a run-time generated data structure. It records all the fireable nodes, and is modified every time the status of a node is changed. The whole program execution progresses by repeating all the steps described on the above.

3.4.2 Concurrency

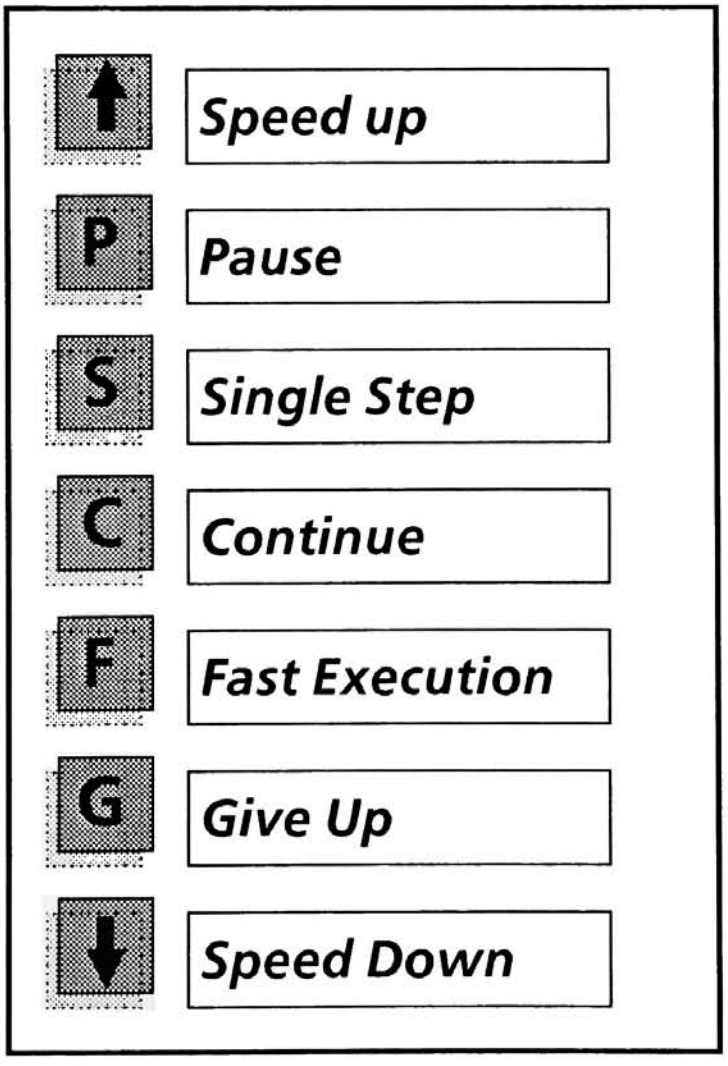
It is a very elegant presentation to show the concurrency of the execution of a dataflow program in GDF programming system. During the execution, all the operators which have their input tokens ready at the input ports are highlighted. These nodes are called *ready for executing* or *fireable nodes*, and are maintained in enabled nodes list.

In a dataflow computer, theoretically, these fireable nodes can fire or can be executed at the same time. In our simulation, only one of these highlighted nodes fires at one instance. It is because that this is a simulation on the computer with sequential architecture. The order of the firing nodes is trivial, since all these fireable nodes have equal opportunities of firing. In some on-the-shelf simulation, the order is determined by a random number generator. In the GDF simulation, the order is

determined by the order when the node was created. The node which is firing is not only highlighted, but also flashing to make difference from the other fireable nodes.

3.4.3 Control Options

The GDF programming system provides seven simulation control options for the user to control the process of a program execution. These options greatly help the user of the graphical dataflow language in writing and testing his/her dataflow programs. The options are *pause*, *continue*, *single step*, *fast execution*, *give up*, *simulation speed up*, and *simulation speed down*. They are presented in a way of seven control buttons displayed on the left hand side of the screen. The user can use the mouse to point and click the option button that he/she wants at any time when the simulation progresses. The figure and description are shown below.



Simulation Control Options

UP arrow: Speed Up

The system uses a simple "for" loop to delay the flashing of the firing node. Using the mouse to click this control button, the user can speed up the simulation speed, that means, the delay will be shorter.

P: Pause

This control option allows the user to pause the simulation temporarily. The user can have enough time to check the current execution status.

S: Single Step

The *single step* option is the most important option of the simulation controls, because that it allows the user to monitor the execution step by step to trace the program. The execution of the program will stop temporarily before a node fires and after a node fires when the user click this control buttons every single time.

C: Continue

After using *pause* or *single step* options, the user can resume the simulation to the regular speed by clicking the *continue* control button. The simulation speed remains the same as it was before *pause* or *single step* was applied.

F: Fast Execution

When the user feels that it is not necessary to look at the simulation progresses, he/she can click the fast execution control button. The system will not flash any firing nodes, and will not stop except any input or output node is reached. The user can get the result from his/her program without any system delay.

G: Give Up

This option allows the user to quit or abort the execution of a program simulation whenever it is necessary. For example, the user finds out the dataflow is generating abnormal data value or the token flow not the correct way as he/she wants, he/she can give up the current execution to modify the graph, then tries to execute the program again.

DOWN arrow: Speed Down

This option prolongs the delay time when a node is flashing. This function is opposite to the first option, speed up option.

4. Application Examples

The GDF Programming System provides a user-friendly environment for users to develop their application programs. Since the GDF language defined in this thesis is a kind of computer language, the user must go through the procedures from learning the language, practicing the use of the language, then he/she might become a skilled programmer. In this chapter, four examples will be presented. Each of the examples includes a brief explanation of the problem, the method of solution, the actual program graph, and the reason for using the example. All of these examples have been successfully tested and exectuted in the GDF Programming System. A walkthrough of the execution of a simple program can be found in the User's Manual.

4.1 Example 1: Solving a Quadratic Equation

In this section, the GDF language will be used to draw a program to solve a quadratic equation.

For a quadratic equation

$$ax^2 + bx + c = 0,$$

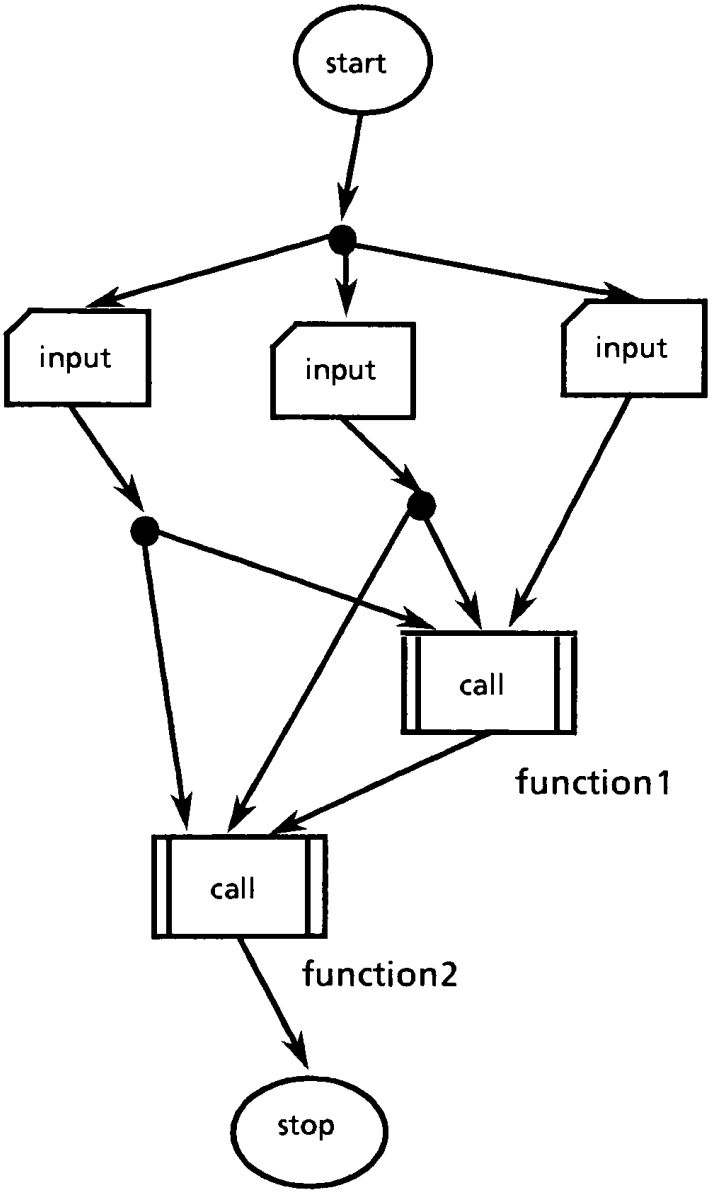
if the coefficients a, b, c are given, the roots x_1 and x_2 can be found by using the formulae

$$x_1 = (-b + \sqrt{(b^2 - 4ac)}) / 2a, \text{ and}$$

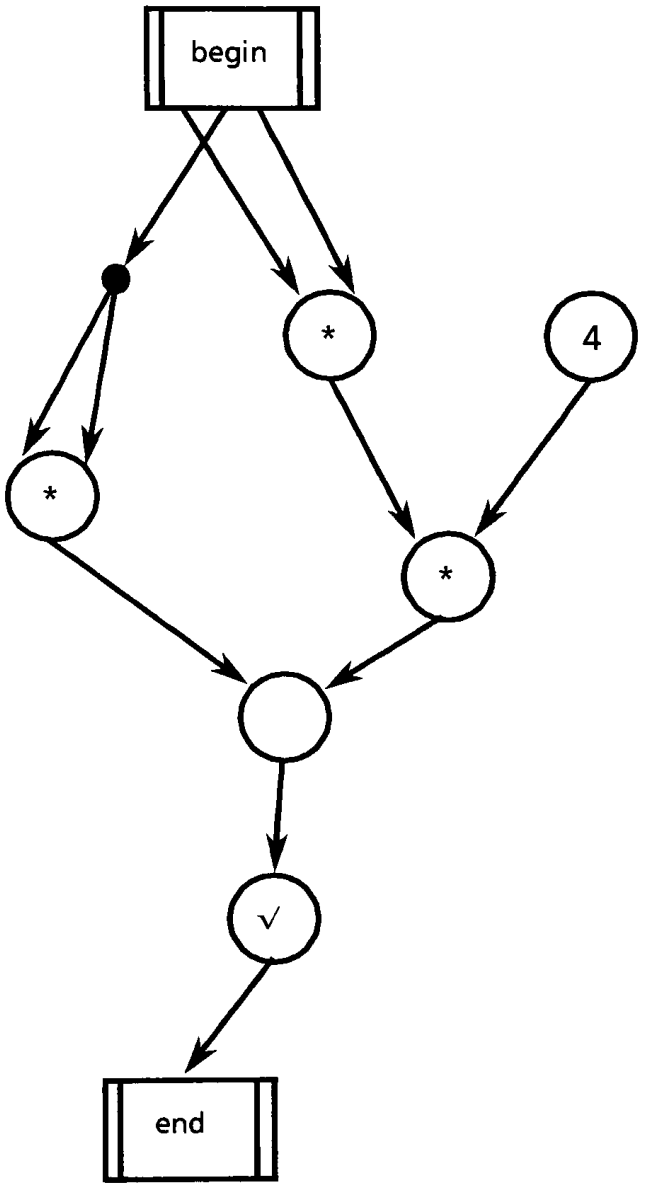
$$x_2 = (-b \quad \sqrt{(b^2 - 4ac)}) / 2a.$$

The program is composed of three graphs, *main*, *function1* and *function2*. The *main* graph is the main program of this example, and the *function1* and *function2* graphs are two subroutines. The *main* graph receives three input integers from the user, and calls *function1* and *function2* to do most of the calculations. The *function1* graph has three imported parameters a, b, and c. It returns the result of $b^2 - 4ac$ to main graph. The *function2* graph also has three imported parameters a, b, and the output of *function1*. It does the rest of the calculation to find the roots x_1 and x_2, and to print out their values. After receiving the return value from *function2*, *main* terminates. The program graphs are shown on the next two pages.

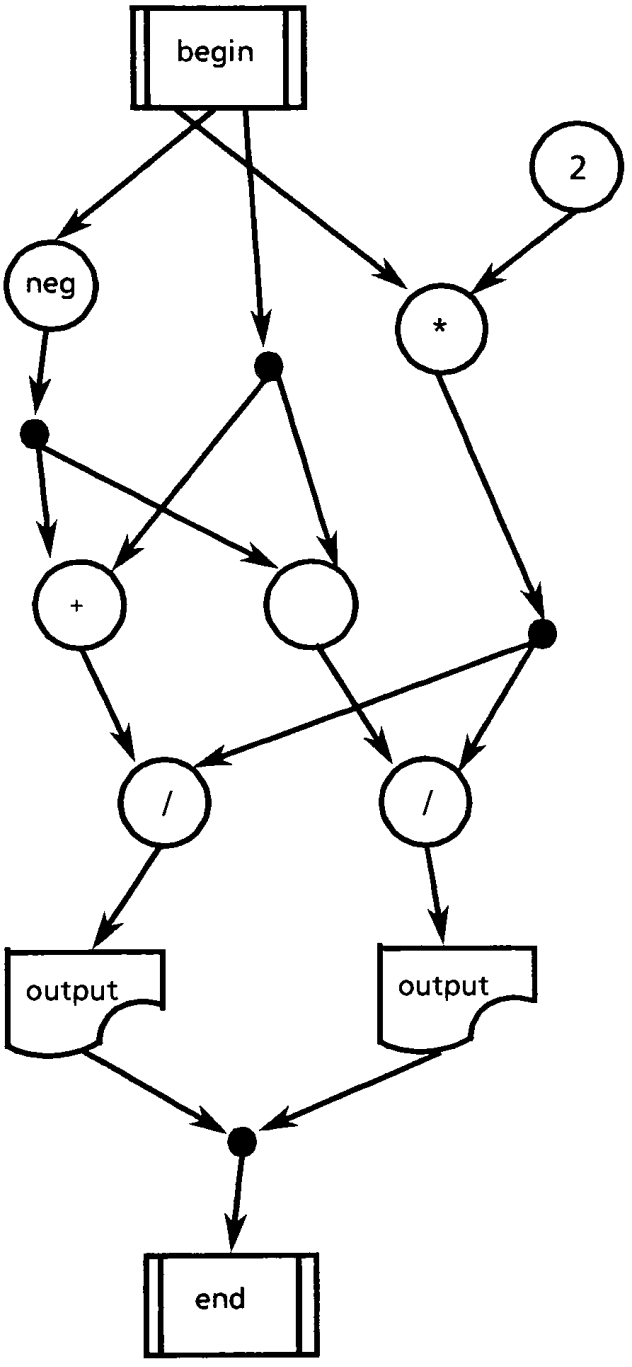
This example covers most of the arithmetic operators, negation operator, constant operator, input, and output operators. It also shows how to program in several functions in a program, which is a very common and useful scheme for structured design in the prevailing computer languages.



Main function graph of Example 1



Function1 function graph of Example 1



Function2 function graph of Example 1

4.2 Example 2: Calculation of Factorials

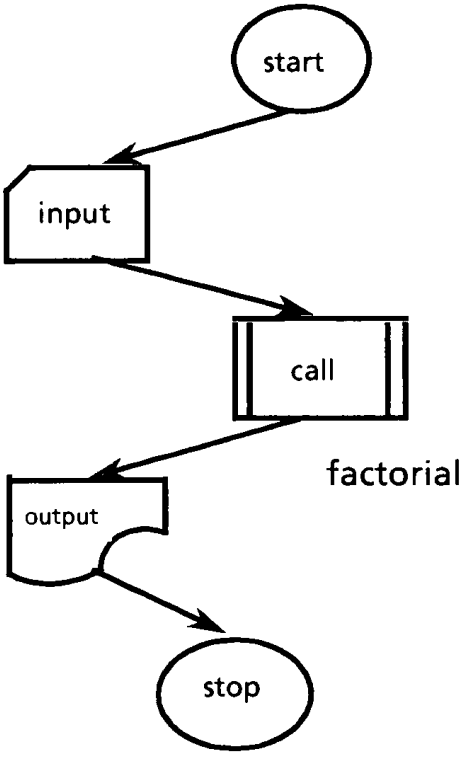
The factorial of a positive integer N is defined as

$$N! = N * (N - 1) * (N - 2) * .....* 1$$

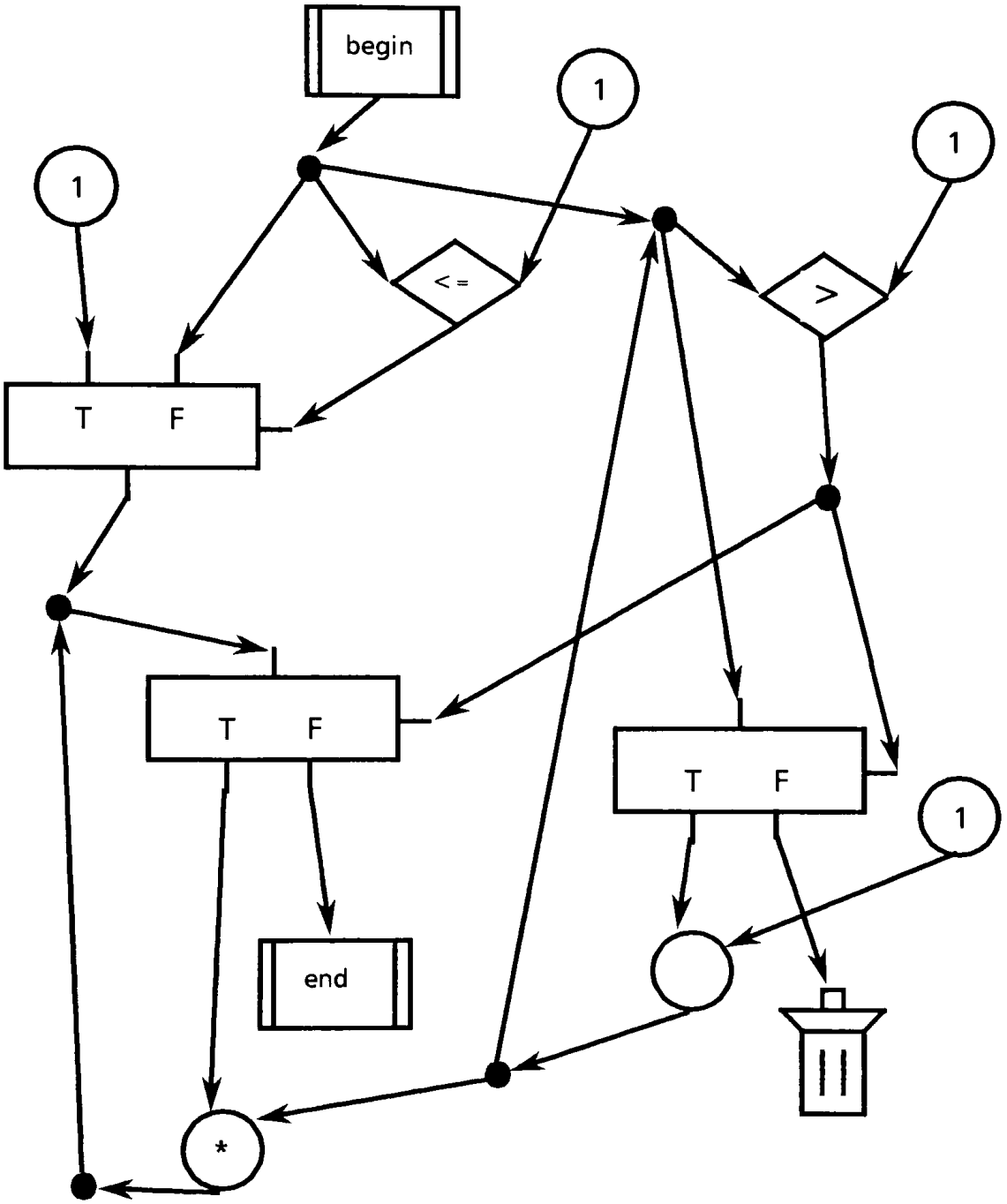
It can be programmed in computer language by using iteration to do the calculation.

This example is composed of two graphs, *main* and *factorial*. *Main* is the main program of the example. It requests the user to input an integer, calls the function factorial to compute the factorial, outputs the value returned from the factorial, and terminates the program. *Factorial* is a subroutine to calculate the factorial. It returns 1 if the imported parameter is one or zero, and returns the factorial for some other positive integer number. The program graph is shown in next page.

The purpose of this example is to show a way of doing iteration in the GDF language. The graph covers conditional testing, arithmetic, switch, and distribute operators.



Main function graph of Example 2



Factorial function graph of Example 2

4.3 Example 3: Finding the Fibonacci Number using the Power algorithm

The Fibonacci Series may be defined by the second-order linear recurrence relation:

$$F_0 = 0; \quad F_1 = 0; \quad F_n = F_{n-1} + F_{n-2} \qquad n > 1.$$

In order to solve this recurrence relation, first notice that

$F_n = c\alpha^n$ will be a solution if $c\alpha^{n-2}(\alpha^2 - \alpha - 1) = 0$.

The solutions of interest are the roots of the indicial equation $\alpha^2 - \alpha - 1 = 0$,

that is $\alpha = 1/2\,(1 \pm \sqrt{5})$.

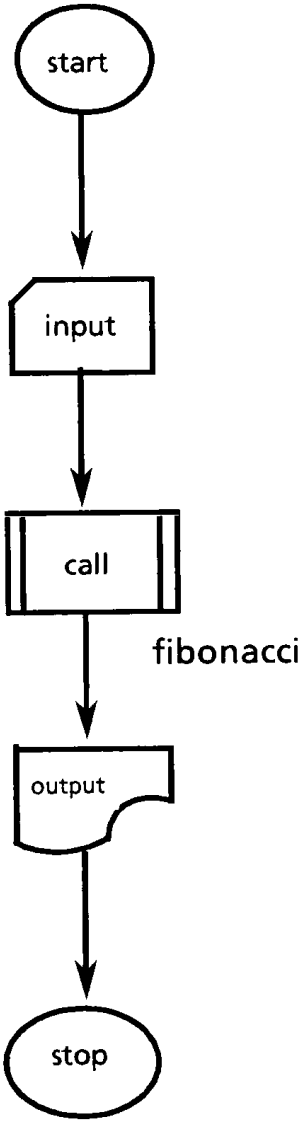
The general solution of the recurrence relation is therefore

$$F_n = c_1 * (1/2\,(1 + \sqrt{5}))^n + c_2 * (1/2\,(1 \sqrt{5}))^n,$$

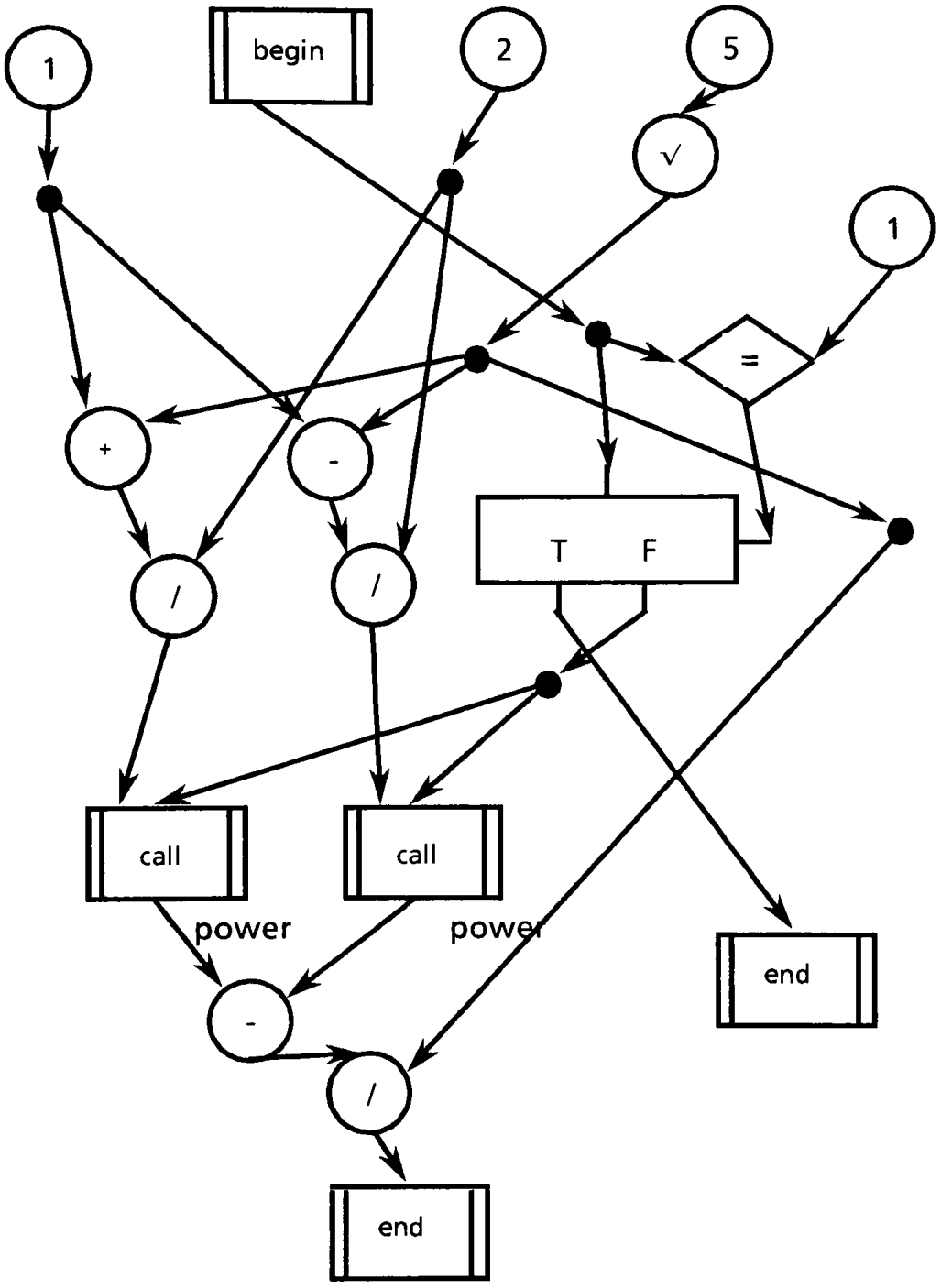
Finally, the coefficients can be determined by using the initial conditions F_0, F_1.

$$c1 = 1/\sqrt{5}, \quad c2 = -1/\sqrt{5}.$$

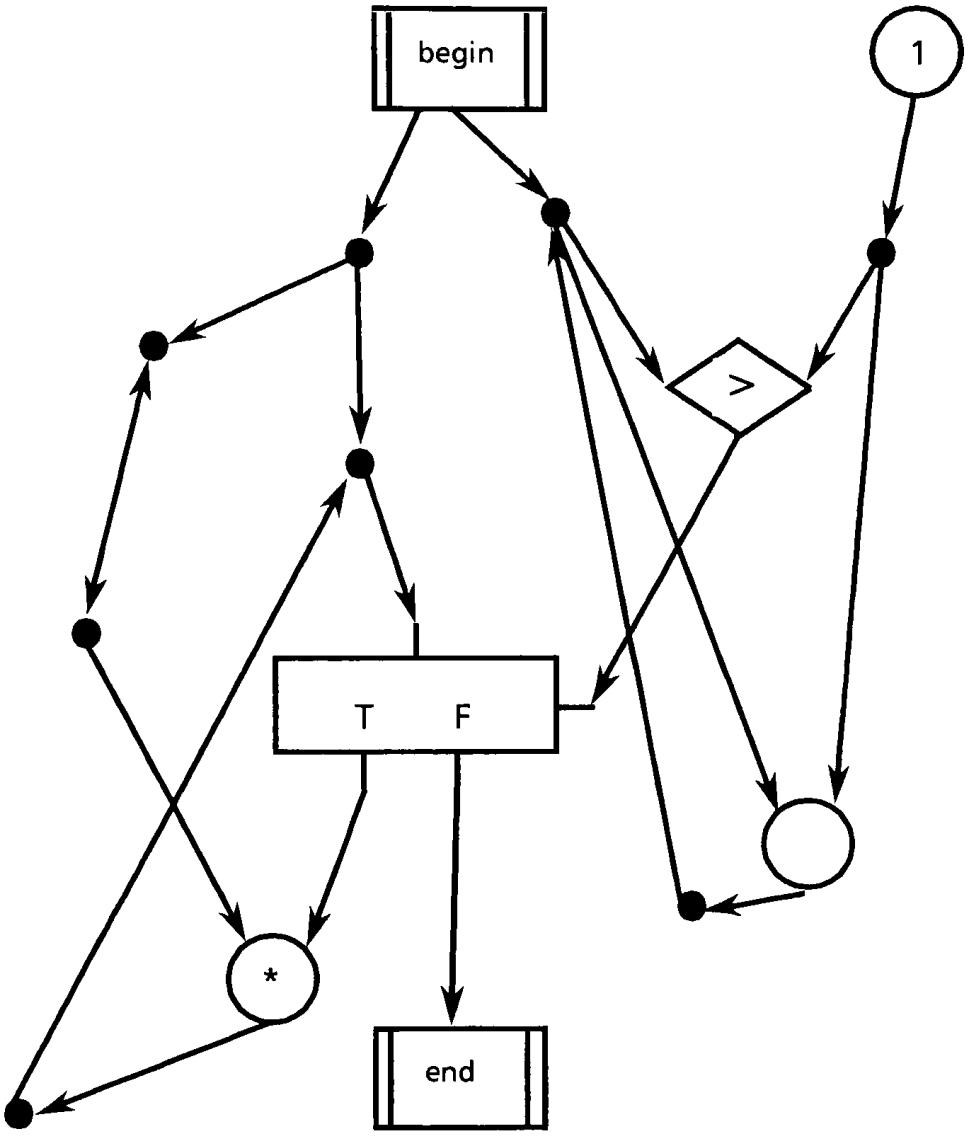
This example is composed of three graphs, *main*, *fibonacci* and *power*. *Main* graph is the main program of the example. It requests the user to input a number, calls *fibonacci*, outputs the result, and terminates the program. *Fibonacci* graph returns zero if the inported parameter is zero, returns one if one, and for the other inputs, it calculates (1/2 (1 + $\sqrt{5}$)), (1/2 (1 - $\sqrt{5}$)), and calls *power* to calculate the power of n. At the end of graph, it returns the value we want. Power graph is used to compute the nth power of a number which is greater than one.



Main function graph of Example 3



Fibonacci function graph of Example 3



Power function graph of Example 3

4.4 Example 4: Finding the Fibonacci Number using Recursion

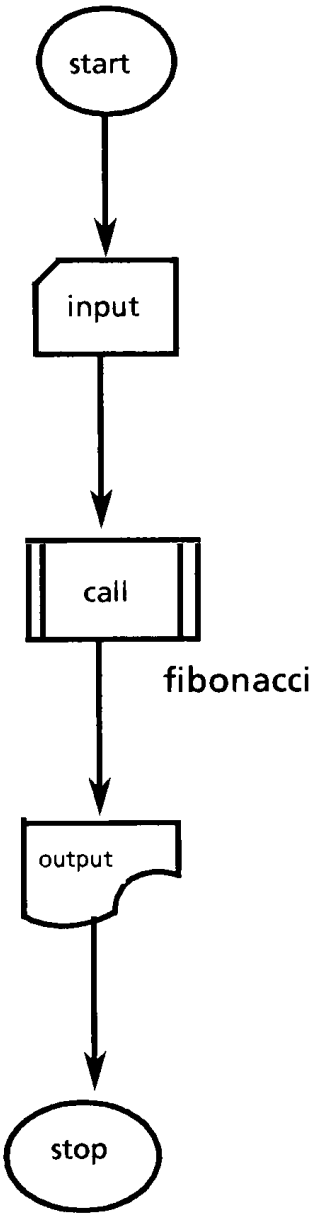
Example 4 employs recursive programming techniques to calculate the Fibonacci Series.

$$Fn = Fn\text{-}1 + Fn\text{-}2, \text{ for } n > 1.$$

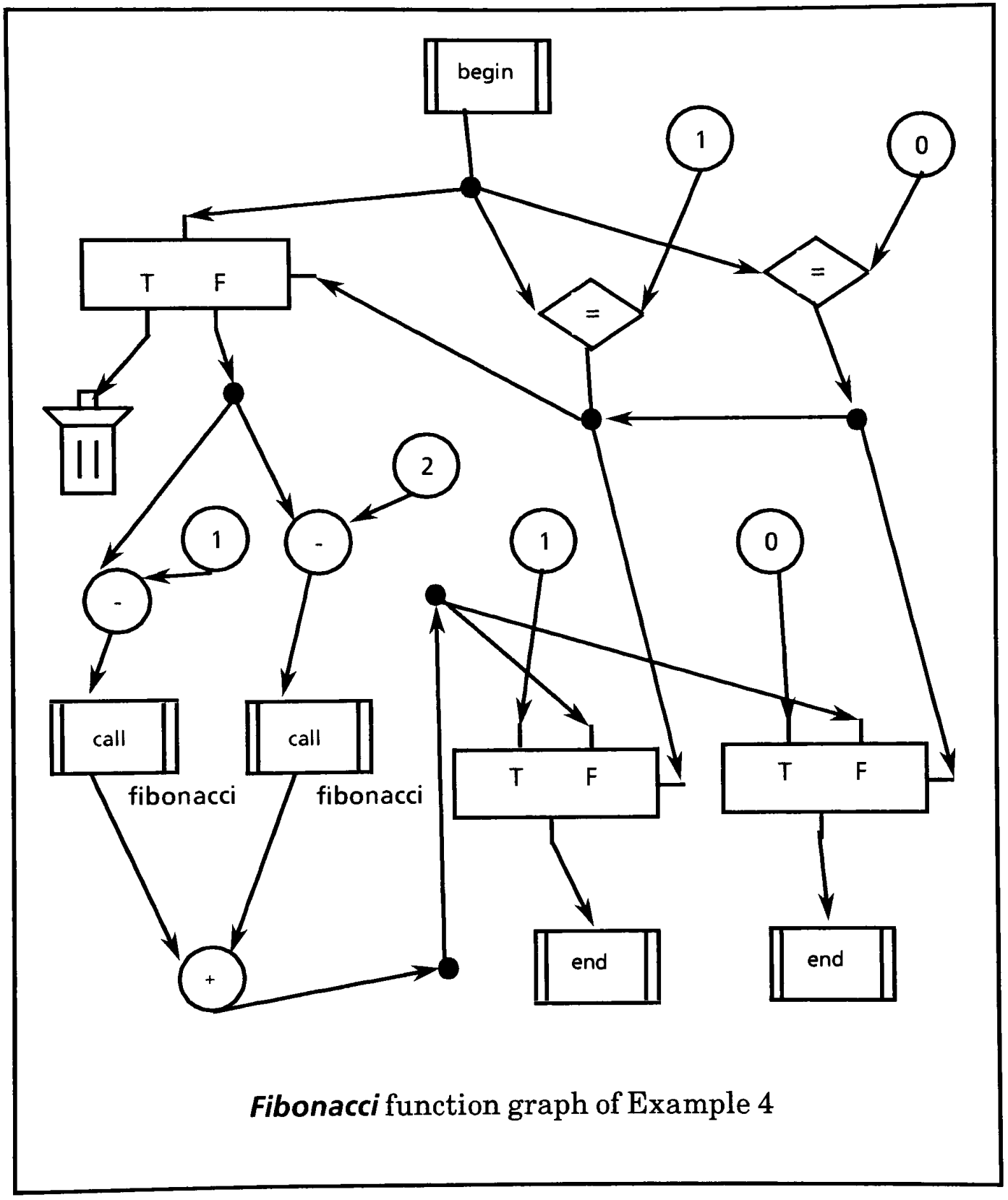
If we want to get the value of F5, the program will compute F4 and F3 first. For the F4, it will compute F3 and F2, and compute F2 and F1 for the F3. The program keeps on calling itself to approach F1 and F0 which will return value 1 and 0.

The program is composed of two graphs, *main* and *fibonacci*. *Main* is the main program of the example. It requests an integer from the user, calls *fibonacci* to compute the fibonacci number, prints the answer, and terminates the program. *Fibonacci* returns zero if the imported parameter value is zero, one if the imported parameter value is one. If the imported parameter value is N, (N > =2), the *fibonacci* graph calls itself twice to compute the fibonacci number of N-1 and N-2, then adds the returned values from these two calls, and returns the sum to *main*.

This example reveals the capability of GDF language to handle recursive programming. A function can call itself to form a recursive function call. Its own graph is displayed again without any tokens at the beginning of the execution. After the newly called function is returned, the original graph is displayed and continue the execution from the operator preceeded by the function call operator. The complete program graphs are shown on the next two pages.



Main function graph of Example 4



Fibonacci function graph of Example 4

5. Project Implementation and Data Structures

This chapter discusses the details about how the GDF programming system was implemented. It describes the implementation environment, the data structures of a graph and the data structures used in the simulation of the execution of a graph, and ends with discussion of the capability of error detection of the system. The memory needed for the application program graphs and the memory needed in the simulation of the execution of the application program are run-time dynamically allocated. There is no limitation over the number of instructions (nodes) that a program can have.

5.1 Implementation Environment

5.1.1 Language

The GDF Programming System programs were written in C programming language. The reason for choosing C language instead of those high level programming languages which support concurrent programming architecture was because that many Unix™ system function calls were used. The C language supports very convenient interfacing with the system calls, allows separation compilation, and supports bitwise operations.

5.1.2 Hardware Configuration

The GDF programming system was built on AT&T Unix™ PC, using standard input, a mouse, and standard output. The AT&T Unix™ Personal Computer is the first personal computer which runs Unix™ Operating System. In the general configuration, it has a 20 or 40 Megabyte hard disk drive, one floppy disk drive, one Megabyte RAM, a 720 by 348 resolution monitor, a standard keyboard, and a mouse. A stand-alone system, it can also use its internal built-in modem to connect to a host computer to serve as a terminal.

The AT&T Unix™ PC screen is an area of 720 pixels wide and 348 pixels high, corresponding to an 80-column by 29-line character display. Each task is viewed in a window. Default window size is set by the application with or without border. Several windows can be opened simultaneously on the display, but only one window at a time, the active window, can receive inputs from the keyboard and the mouse. The Graphical Programming System uses a full screen window without border, and many auxiliary windows for editing graphs and displaying messages.

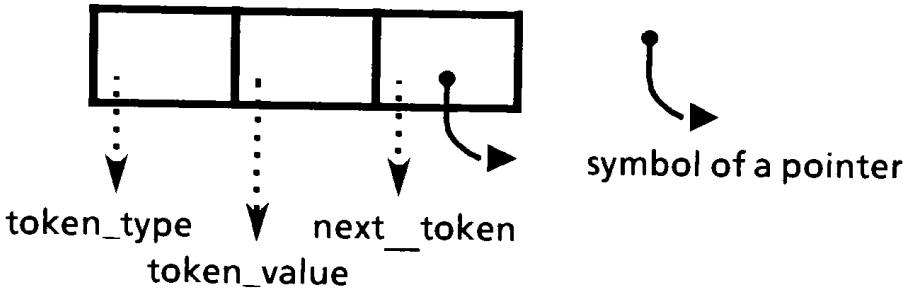
5.2 Data Structure of a Graph

A graphical dataflow language program is composed of one or many function graphs. Each of the graphs is a directed graph, and consists of a number of nodes and directed arcs in which each node represents an operator or function and each directed arc a medium over which data items flow. When a graph is executed, tokens are generated from some firing nodes, and flow on the arc as directed from the output port of a node to an input port of a node. If all the necessary input tokens have

arrived, the node is executed; its input tokens are consumed, and an output token is then generated. In this section, the data structure of tokens, node, arc, function, and program graph are discussed.

5.2.1 Token

The figure below shows the structure of a token.



Data Structure of a **Token**

token_type: The type of a token value. It could be 1, 2, 3, or 4. The numbers stand for INTEGER type, CHARACTER type, REAL NUMBER type, and BOOLEAN type respectively.

token_value: The value of a token. It is a union structure which is defined for containing integer, character, boolean value, and double floating number.

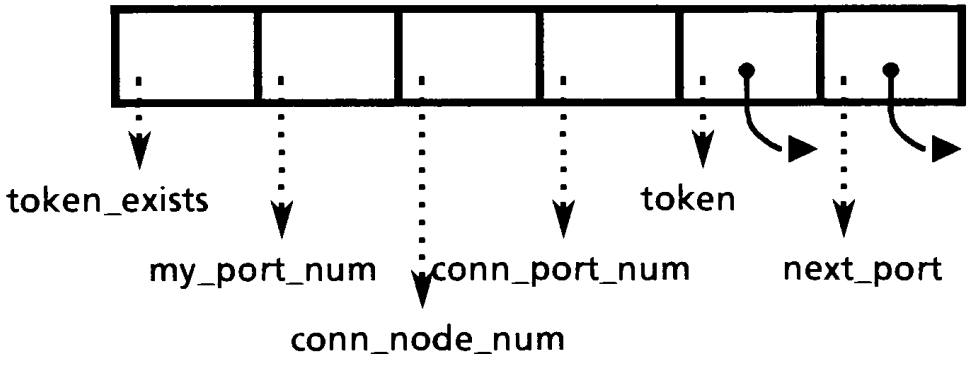
next_token: A pointer points to another *token* structure. This field is not used in current implementation. It may be useful if the dataflow computer is simulated as a dynamic dataflow simulation.

A dataflow computer can be implemented in two ways, static execution and dynamic execution. In the static execution model, a fireable node will not fire (generate an output token) until its previous generated token is consumed or absorbed by its destinated node. In this case, each arc in the graph only takes at most one token during the run-time. For the dynamic execution model, every fireable node fires when all the necessary input tokens have arrived no matter whether the previous generated token has been consumed or not. It the previous token has not been consumed, the newly generated token will be queued on the input port of the

destinated node. The dataflow computer simulator implemented in the GDF programming system follows the static execution rule. So that the next_token field in the token structure is not used in current implementation.

5.2.2 Node

Each node in the graph is one of the operators available in the system. It can be a simple operation, a comparison, or a function which accomplishes complicated operations and returns a single token. The data structure of a node contains the information which describes the node itself on the graphical display and represents its run-time status. Two other data structures are used in the node structure. They are the data structure of *port* and the data structure of *property*. The figures below show their organizations and relationships.



Data Structure of a Port

token_exists: A flag contains TRUE if a token exists on the port, or FALSE if there is no token on the port.

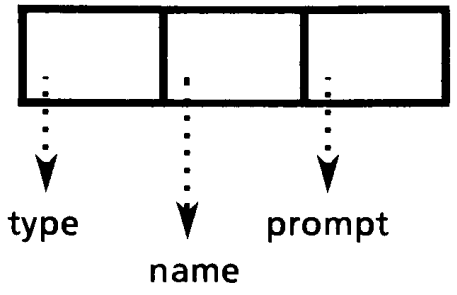
my_port_num: The port number of the port. A node can have several input ports and output ports, these ports are assigned a number by the system.

conn_node_num: The node number of a node which is connected to the current node. The current node is the node which contains this port structure.

conn_port_num: The port number of a port which is connected to the current port.

token: A pointer points to a *token* structure. The *token* structure carries a token which is generated by the preceded node.

next_port: A pointer points to another *port* structure.



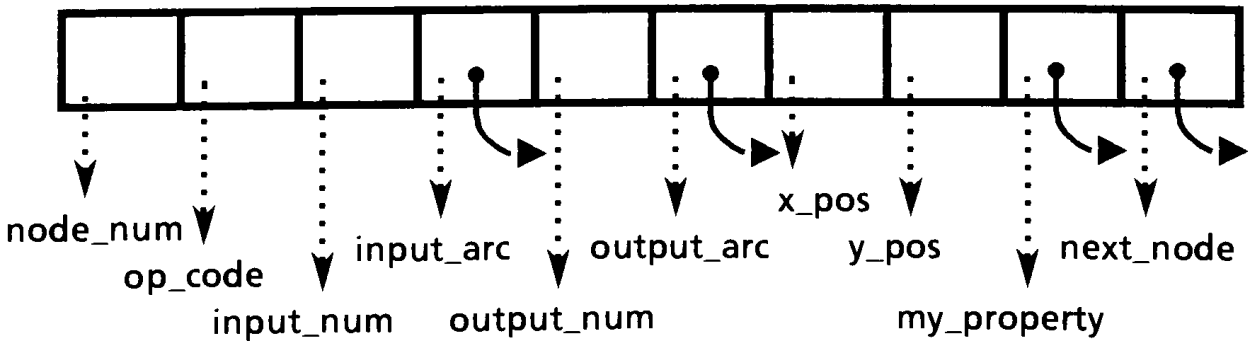
Data Structure of a **Property**

type: The type of a token which is generated by CONST node or INPUT node.

name: The name of a function which is used when a CALL node is created on a graph.

prompt: A user can create his own prompt strings to use when the INPUT and OUTPUT nodes are executed.

The property structure is only used for CONST node, INPUT node, OUTPUT node, and CALL node to hold the information of a token, the name of a function, or the prompt string.



Data Structure of a **Node**

node_num: The node number of a node. The system assigns a number to a node when the node is created on the current edited graph.

op_code: The type of the operation of a node. Each operator available in the system has a pre-defined constant name for it, such as ADD, INPUT, CALL, and etc.

input_num: The number of input ports that a node can have.

input_arc: All the input ports of a node form a linked list and the linked list is pointed by this field.

output_num: The number of output ports that a node can have.

output_arc: All the output ports of a node form a linked list and the linked list is pointed by this field.

x_pos: The X coordinate of the position where the node is located.

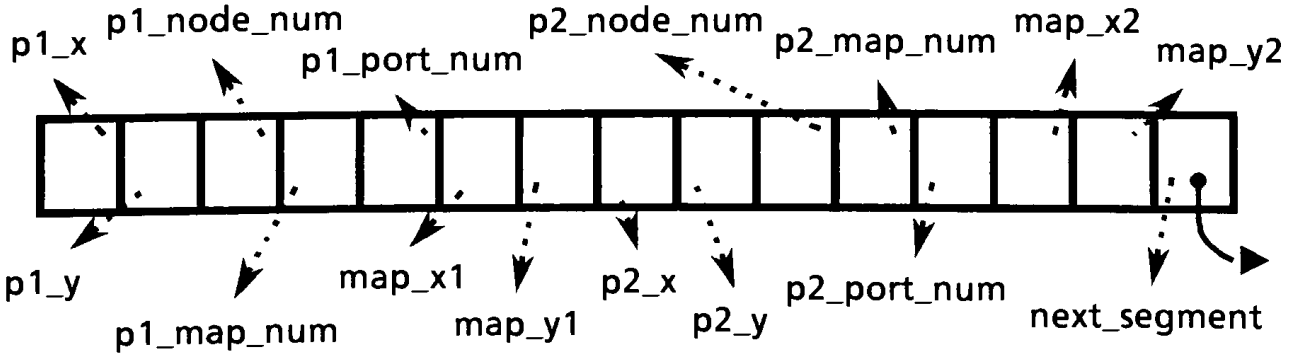
y_pos: The Y coordinate of the position where the node is located.

my_property: A pointer points to a *property* structure if the node is one of INPUT, OUTPUT, CONST, or CALL.

next_node: A pointer points to another *node* structure.

5.2.3 Arc

An arc is a line segment, connecting an input port of a node to an output port of a node. Its data structure contains the information of the two nodes it connects. The data structures of arcs are useful in the graphical display.

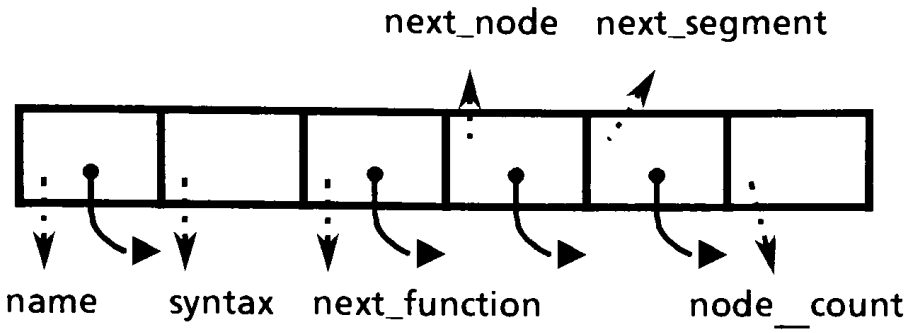


Data Structure of an Arc

p1__x:	The X coordinate of the first end point of the line segment.
p1__y:	The Y coordinate of the first end point of the line segment.
p1__node__num:	The node number of the first node that the arc connects.
p1__map__num:	The map number of the first node. The map number is the same as the op__code which is used in *node* structure.
p1__port__num:	The port number of the output port that the arc connects.
map__x1:	The X coordinate of the first node's bitmap.
map__y1:	The Y coordinate of the first node's bitmap.
p2__x:	The X coordinate of the second end point of the line segment.
p2__y:	The Y coordinate of the second end point of the line segment.
p2__node__num:	The node number of the second node that the arc connects.
p2__map__num:	The map number of the second node.
p2__port__num:	The port number of the input port that the arc connects.
map__x2:	The X coordinate of the second node's bitmap.
map__y2:	The Y coordinate of the second node's bitmap.
next__segment:	A pointer points to another *arc* structure.

5.2.4 Function

The graph of a function is composed of a number of nodes and arcs. The data structure of a graph contains two linked list, one for the nodes and one for the arcs. They are linked lists of *node* structure and *arc* structure respectively. Every function graph has a data structure to maintain the node and arc informations. It is named as *funct__head*, its data structure is shown on the next page.



Data Structure of a **funct__head**

name: The name of the function graph. (e.g. the function name)

syntax: A flag reflects the status of connections is OK or not. A function can be executed only when this flag is TRUE.

next__function: A pointer points to a function's *funct__head* structure. This field is useful when a program has more than one function.

next__node: A pointer points to a *node* structure which is used as the head of the linked list of *node* structures.

next__segment: A pointer points to an *arc* structure which is used as the head of the linked list of *arc* structures.

node__count:. This field keeps track of how many nodes that a graph has.

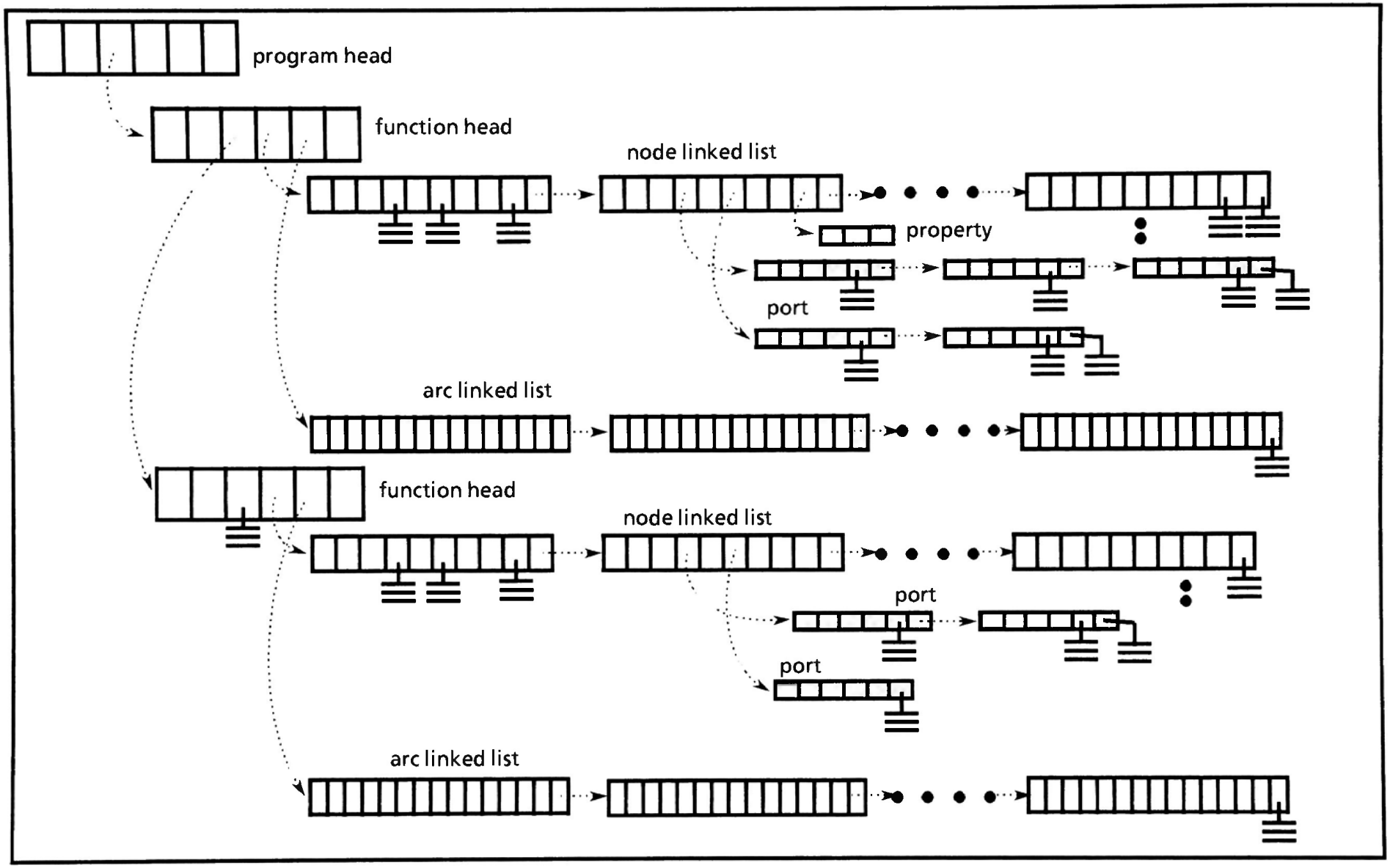
During the editing of a graph, when a node is copied from the operator menu panel and located in the scratchpad, a block of memory is allocated for the node structure and the informations of the node are recorded in pertinent fields. The pointer of the block of memory is appended to the end of the node linked list. When a node is deleted from the scratchpad, the data structure for the node is deleted from the linked list and the memory occupied by the node structure is released. Same situation happens on arcs. When two ports are connected to form an arc, a block of memory is allocated for the arc, and informations of the arc are recorded in the pertinent fields. The pointer of the block of the memory is appended to the end of the arc linked list. When the arc is deleted, the data structure for the arc is deleted from the arc linked list and the memory occupied by the arc structure is released. When

the user applies a *move* operation on a node, the related informations are updated in both node and arc data structures.

5.2.5 Program

Combine the discussion and presentation of the data structures in the previous sections, a complete program data structure is shown on next page. An extra funct_head structure is used as the head of the function linked list.

The GDF programming system efficiently exploits the usage of memory, because all the memory for data structures is run-time allocated, consistently, the memory is released immediately whenever the structure is not used any more.



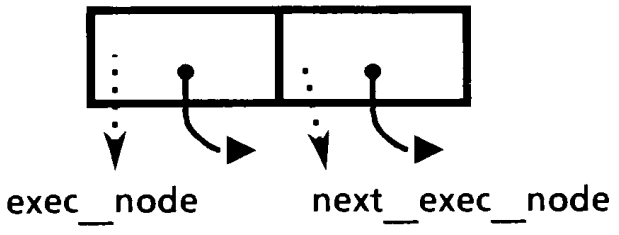
general data structure of a program

5.3 Dataflow Simulation

The dataflow simulation in GDF programming system follows the static execution rule. A node is fireable when all the necessary input tokens have arrived and the previous generated token has been consumed by the successive operator. All the fireable nodes are maintained in a linked list which is call enabled node list. Every function in a program except the main function is invoked by a CALL operator. The called function will be duplicated, and the duplicated function is stacked. A function can be called as many times as necessary, and is allowed to call itself recursively. A block of memory is allocated for a token structure when a token is generated. The pointer of the token structure is placed in a port of the destinated node. Tokens are generated by firing and consumed by executing a node.

5.3.1 Basic Data Structures

Two data structures are used for dataflow simulation. They are node_exec and funct_exec. The figures below show their structures.

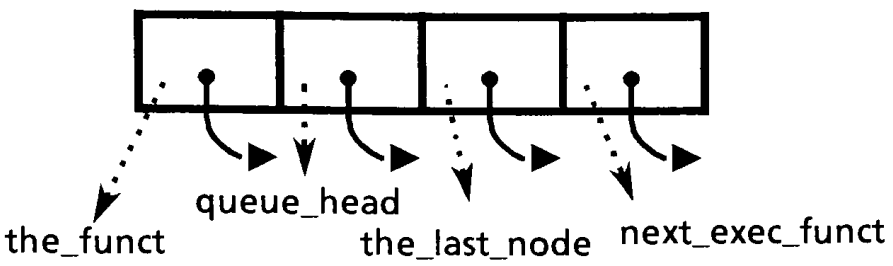


Data Structure of a **node_exec**

exec_node: A pointer points to a *node* structure which is a fireable node in the node linked list.

next_exec_node: A pointer points to a *node_exec* structure where records the next fireable node.

A linked list of node_exec structure constitutes the enabled node list, a structure we have discussed in chapter 4.



Data Structure of a **funct__exec**

the__funct: A pointer points to a *funct__head* structure. When a function is duplicated, the address of its *funct__head* structure is placed in this field.

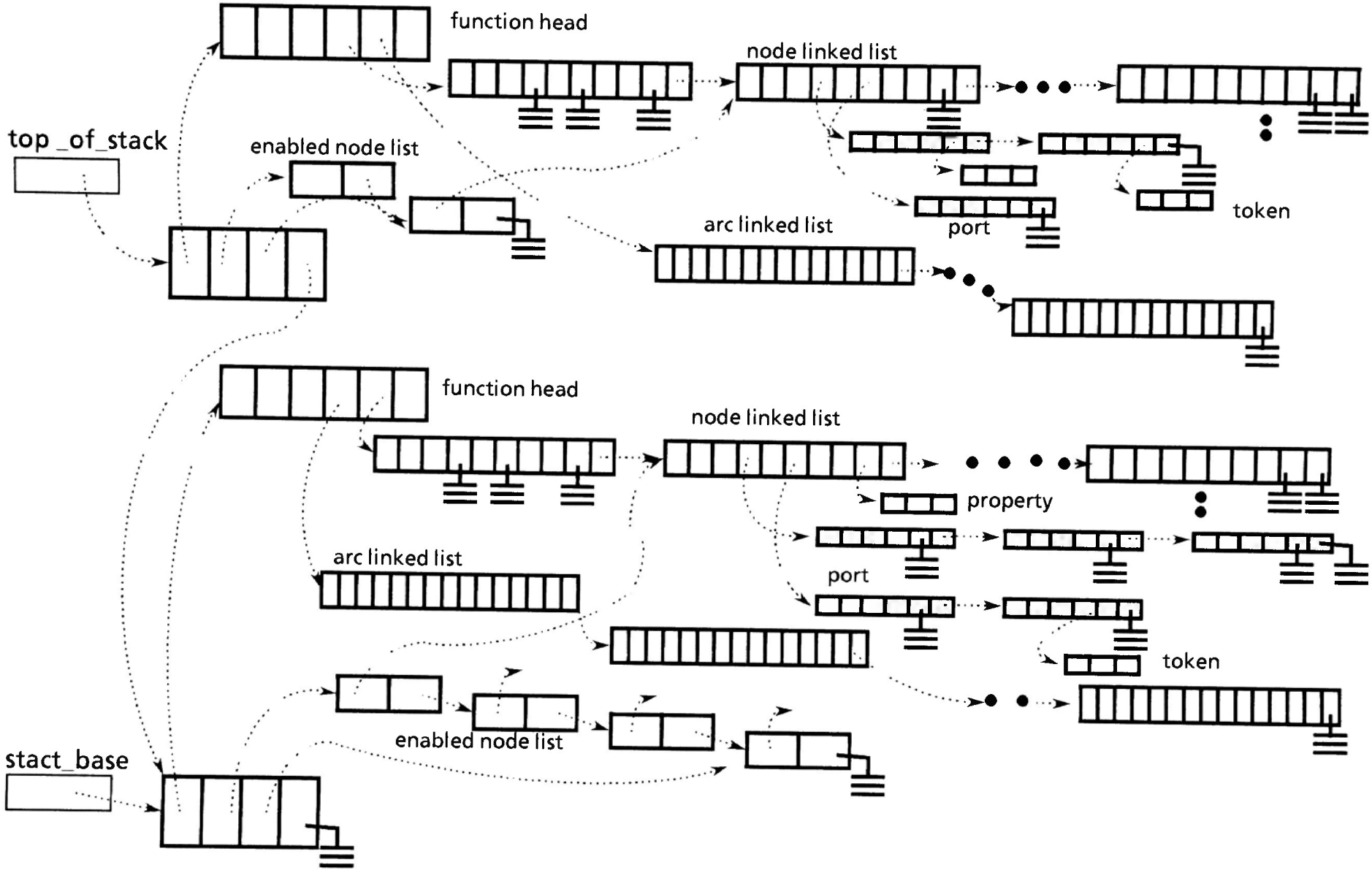
queue__head: A pointer points to a *node__exec* structure. This field contains the head of the linked list where all the executable nodes are queued; the linked list is the enabled node list.

the__last__node: A pointer points to a *node__exec* structure which is last object in the enabled linked list. If the content of the *the__last__node* is the same as *queue__head*, then, it means that no more fireable node exists. The function should either returning a token and terminates itself, or aborted because of deadlock or starvation.

next__exec__funct: A pointer points to a *funct__exec* structure. The function pointed by this field is the previous executed function which will resume its execution after the current function execution returns token.

5.3.2 Run-time Data Structure

Two variables are used during the run-time simulation, one is *stack__base*, and the other is *top__of__stack*. Both of these two variables are pointers which point to *funct__exec* structures. The figure on the next page shows the run-time data structure.



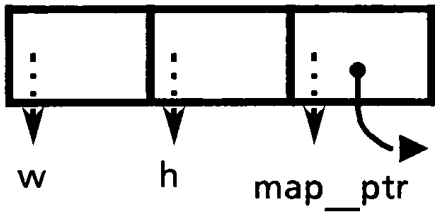
Run-time Data Structure

5.4 Bitmap and Display Control

The AT&T Unix™ PC screen is an area of 720 pixels wide and 348 pixels high. The graph editor uses the top 720 by 300 for scratchpad and operator menu panel, and the bottom 48 lines (4 text line, line height 12 pixels) for displaying messages.

5.4.1 Bitmap

All the operators, digits, characters displayed on the top 720 by 300 area are formed in bitmaps which are two dimensional arrays. An operator icon is contained in a 48 pixels wide and 37 pixels high box, and in memory, it is expressed in a structure with 3 short integers in X dimension and 37 rows in the Y dimension. A digit or standard size character is contained in an 8 pixels wide and 8 pixels high box, expressed in a structure with 1 byte by 12 rows in memory. In additional to the operators, digits, characters bitmaps, the system needs a bitmap for the entire screen, which is 45 x 300 short integers memory block. These information about a bitmap is contained in a bitmap data structure. The figure below shows the structure.



Data Structure of a **Bitmap**

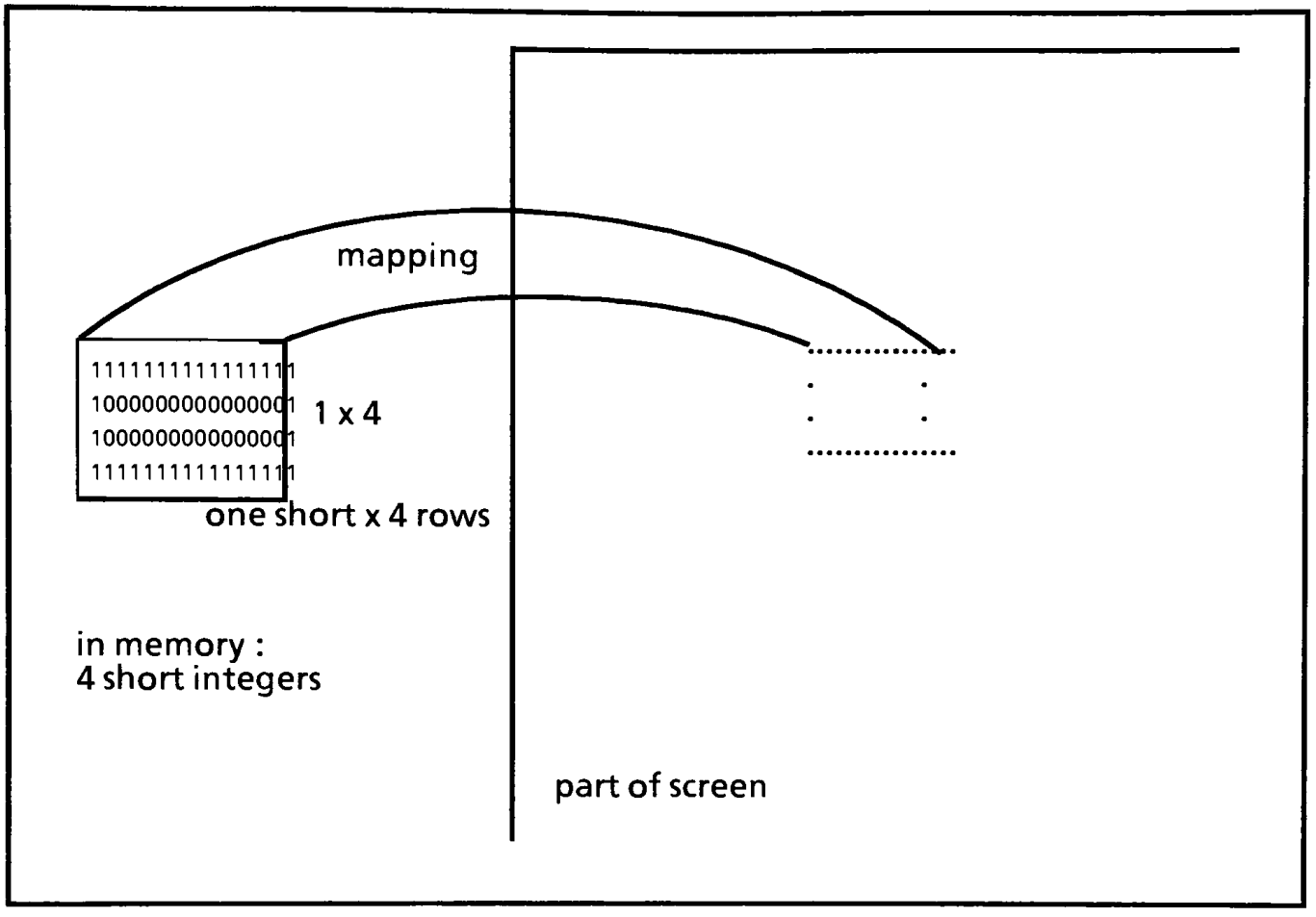
w: The width of a bitmap. It is expressed in the number of short integers. One short integer consists of 2 bytes which is 16 bits, and can hold the values of 16 pixels.

h: The height of a bitmap. It is expressed in the number of pixels.

map__ptr: A pointer points to an area of memory used by the bitmap.

One bit of memory holds a 0 or 1, which reflects the status of one pixel on the screen. If the bit value is 0, after mapping the memory onto the screen, the related position pixel will be turned off, which will display a dark spot. If the bit value is 1, after mapping the memory onto the screen, the related position pixel will be turned on,

which will display a bright spot. The mapping between a memory bitmap and screen display is shown below.



5.4.2 Display Control

The system employs an *ioctl* system call. The *ioctl* call performs a variety of functions on character special files (devices). One of the functions, WIOCRASTOP is used in system programs to access window's pixel data. The WIOCRASTOP function performs raster operations from source to destination, which could be memory to memory, memory to screen, screen to memory, and screen to screen. The display screen can be refreshed very quickly for the benefits of the fast speed of the raster operations.

Both the memory of the source and destination planes are rectangular areas. The basic behavior of raster operation conforms to a vector description

destination = destination operation (source operation (source, pattern))

where the pattern is an array of 16 x 16 pixels arranged as 16 consecutive shorts. There are five source operations, SRCSRC, SRCPAT, SRCAND, SRCOR, SRCXOR; and five destination operations, DSTSRC, DSTAND, DSTOR, DSTXOR, and DSTCAM. SRCSRC is the identity function whose value is the unmodified source rectangle itself. SRCPAT's value is that of the pattern and bears no relationship to the source. SRCAND is the AND of the source and the pattern; SRCOR, the inclusive OR; SRCXOR, the exclusive OR. DSTSRC is the identity function, returning the result of the source operation unchanged. DSTAND is the AND of the destination with the result of the source, DSTOR is the inclusive OR, and DSTXOR the exclusive OR. DSTCAM AND's the one's complement of the source operation into the destination.

The system uses SRCSRC to combine with one of DSTSRC, DSTOR, DSTAND, and DSTXOR operations to refresh the screen, to display and refresh single object, to make flashing on the firing node, and to move bitmaps data around memory.

The system also utilizes Bresenhman's algorithm in drawing the arcs on the screen bitmap. Some transformation and rotation matrix computations are needed in drawing the arrows of arcs.

5.5 Error Detection

The GDF programming system is capable of detecting errors for the user in editing a graph or executing a program. The complete set of warning and error messages are listed and explained in the User's Manual.

5.5.1 Graph Program Syntax

To edit a dataflow program in the GDF programming system, unlike writing a high level programming language, the user does not have to worry about the order of locating operations, spaces, precedence, parentheses, or semicolons. But a set of simple rules should be followed in the editing of the graphs of a program.

1. Before executing a program, all the necessary arcs should be connected.
2. A program can only have one START node.
3. A function can only have one BEGIN node, but can have multiple END nodes.
4. In connecting an arc, the node with the output port should be selected first, then the node with the input port can be selected.
5. The copied operator can only be located inside of the scratchpad area.

5.5.2 Run-time Error messages

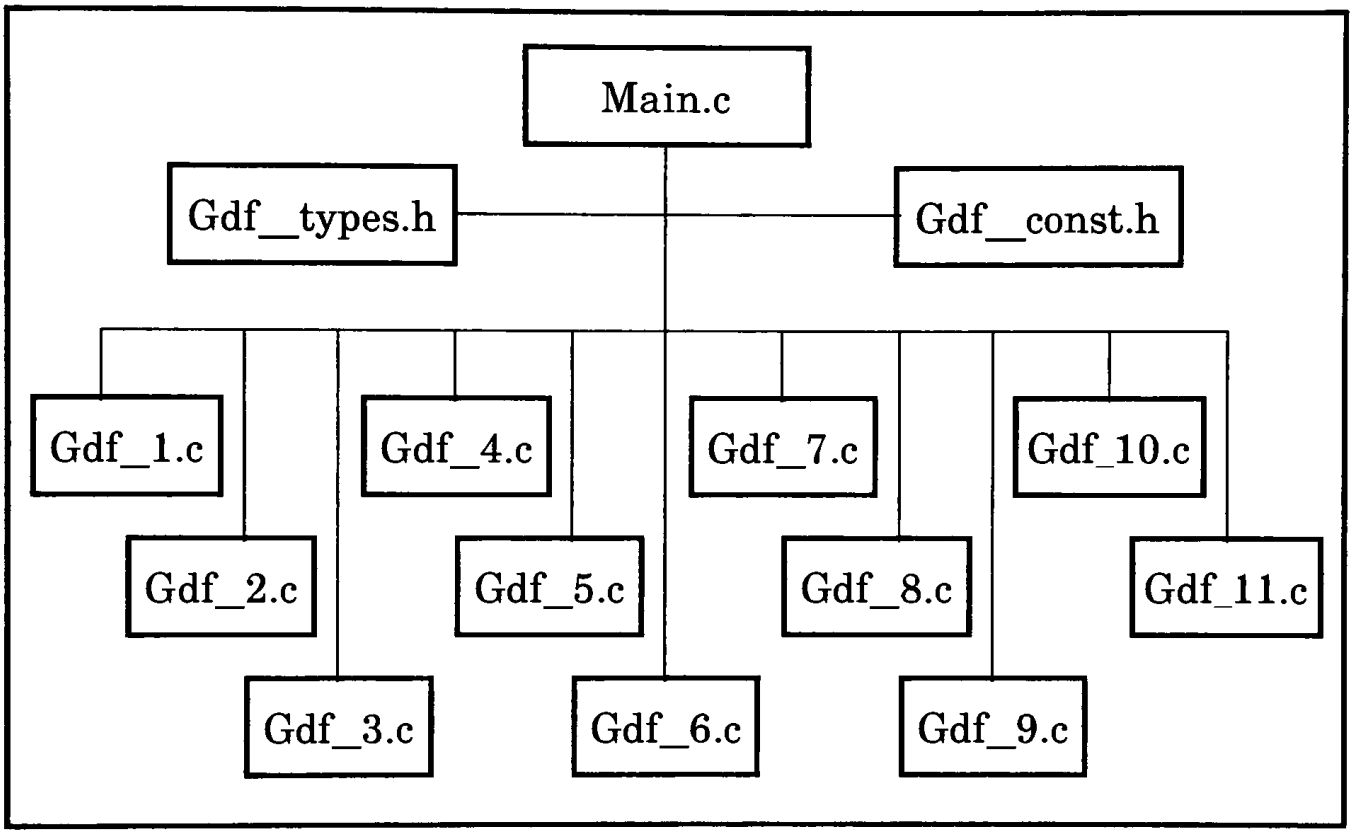
Various of messages will be displayed in an error message window when the program execution encounters an error which may cause the execution to be aborted. The errors include deadlock, starvation, insufficient memory , type inconsistent of an operator, or improper termination of a function or program.

6. Program Description

The system programs of the GDF Programming System were written in C Programming Language, and utilize structured module design. The system programs structure and description of each function are described in this chapter.

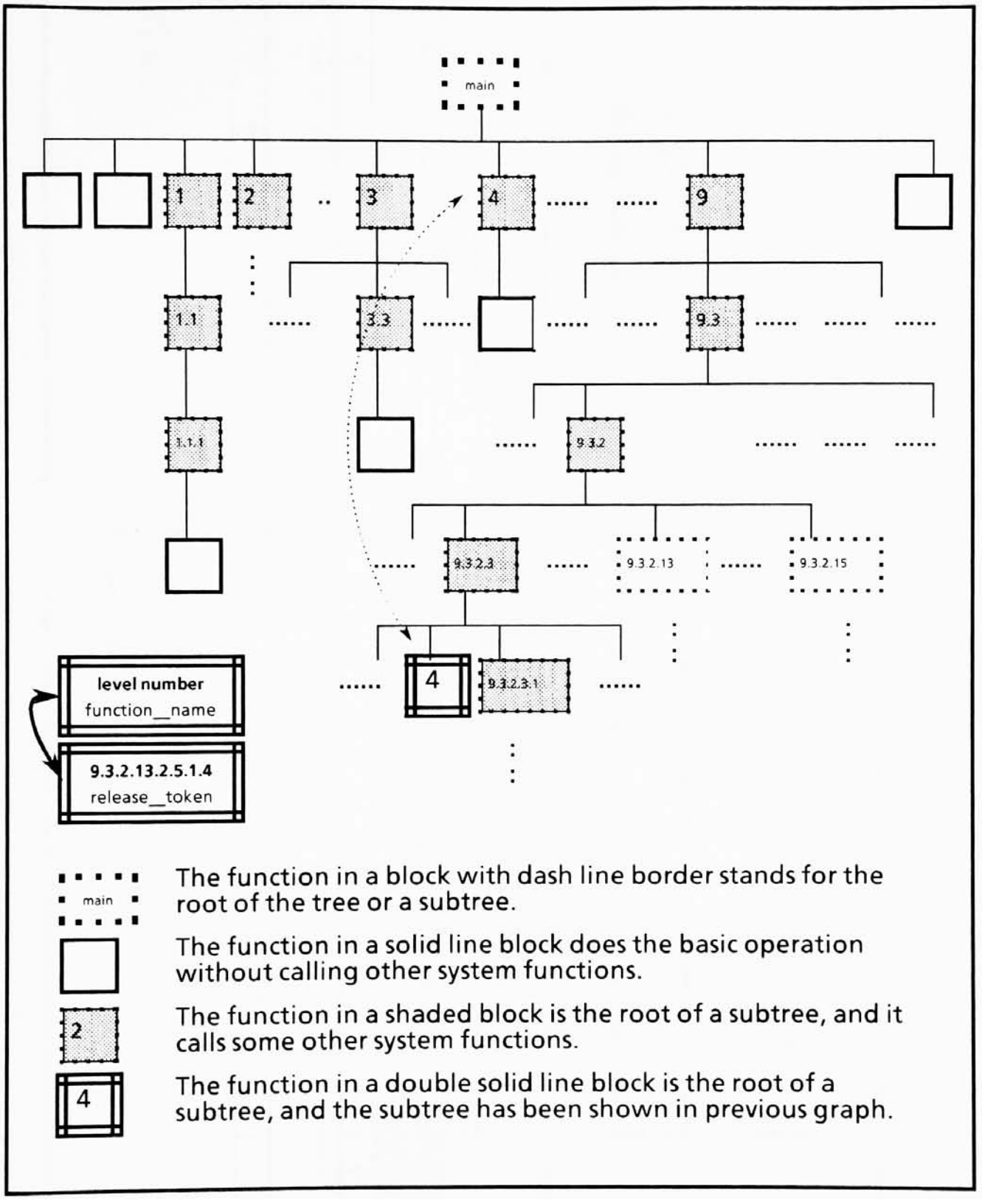
6.1 Program Organization

The complete system programs are composed of two .h files, main.c, and eleven gdf__n.c files (n is in [1..11]). For these gdf__n.c files, functions which perform related operations are grouped in one file. The details of the file contents are explained in next the three sections. The diagram below shows the organization.

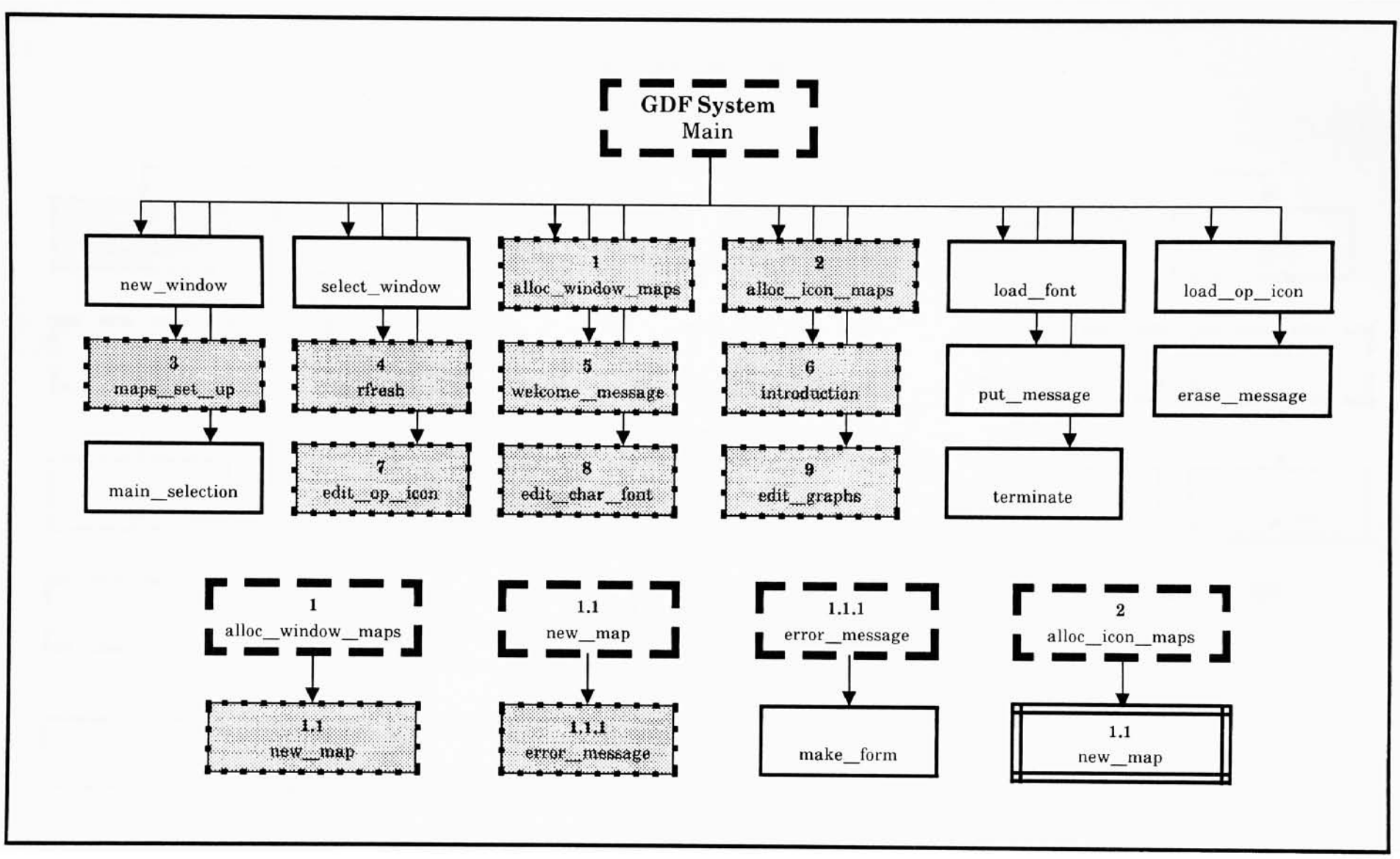


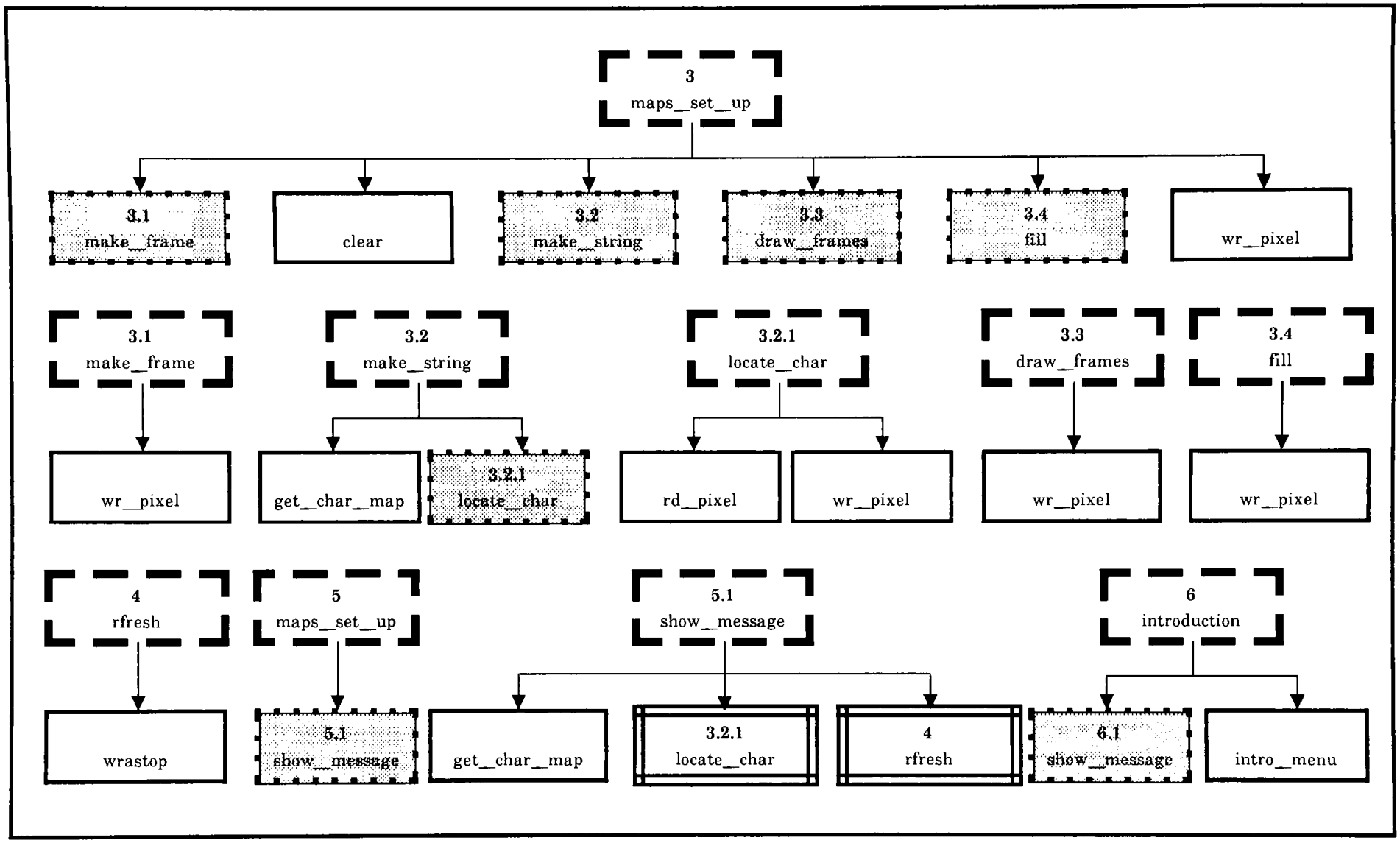
6.2 System Process Diagrams

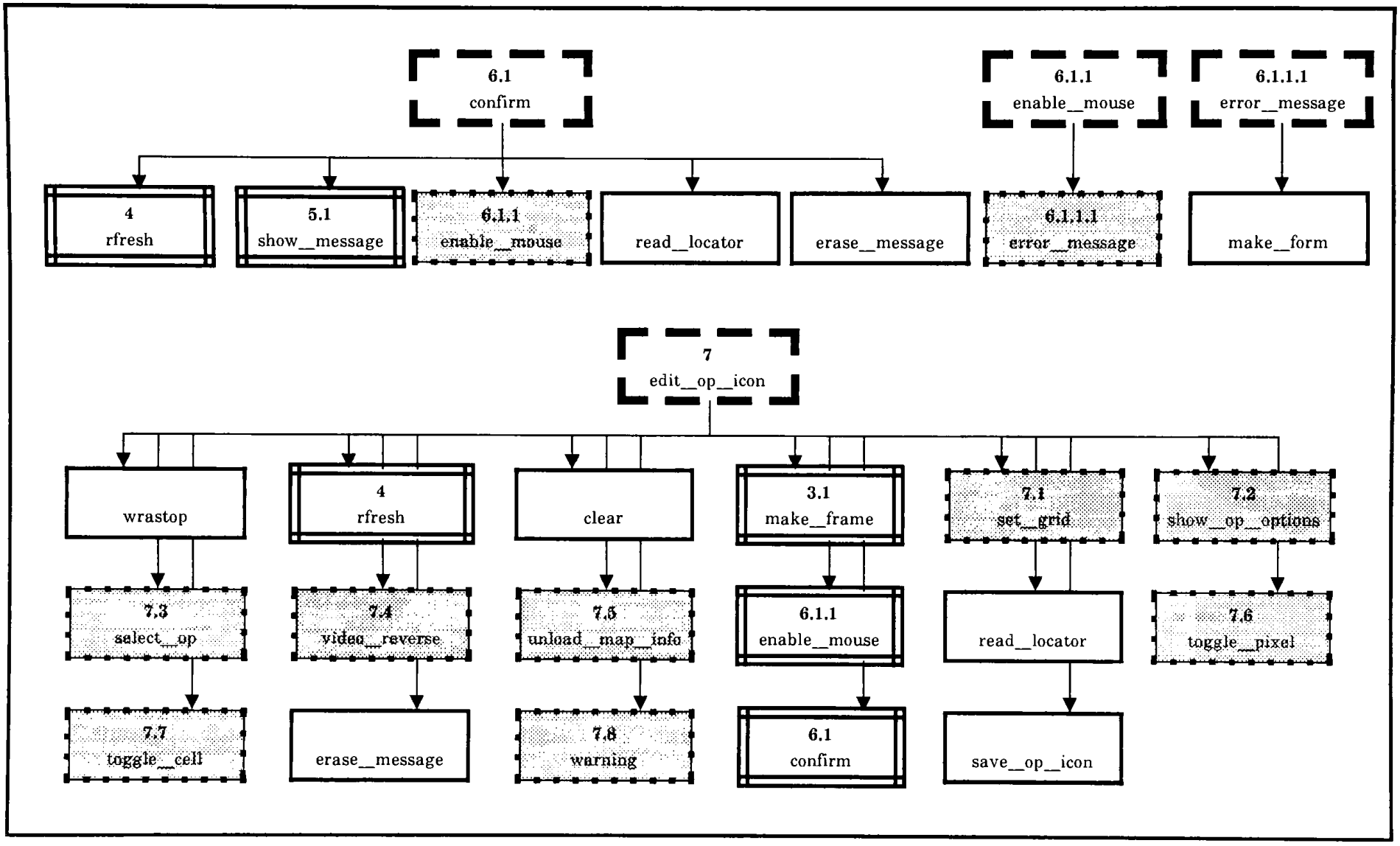
The process diagrams of the system are shown in this section. Each diagram shows one level of the relationship between the calling function and the called functions. Each block contains the name of a function (or process) and a level label which shows the position of that function in the hierarchical structure. The functions that introduced in solid line rectangles in the called function level perform basic operations and do not call any other system functions. The functions that introduced in dash line rectangles do call some other functions, and their hierarchy structure will be shown in another diagrams. The relationship can be traced by following the level number. These functions introduced in double solid lines rectangles have their own structure which has already been illustrated in previous diagrams.

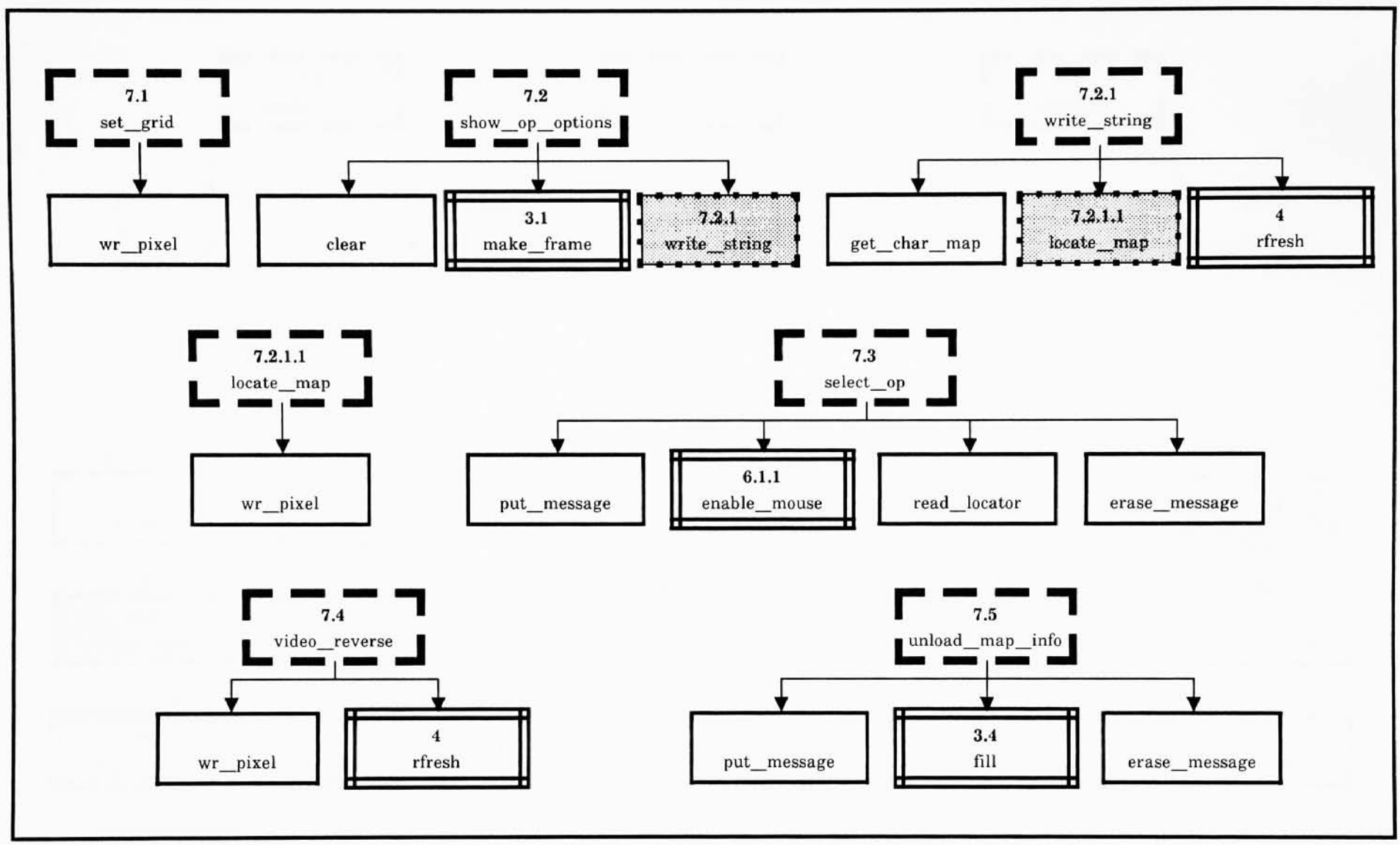


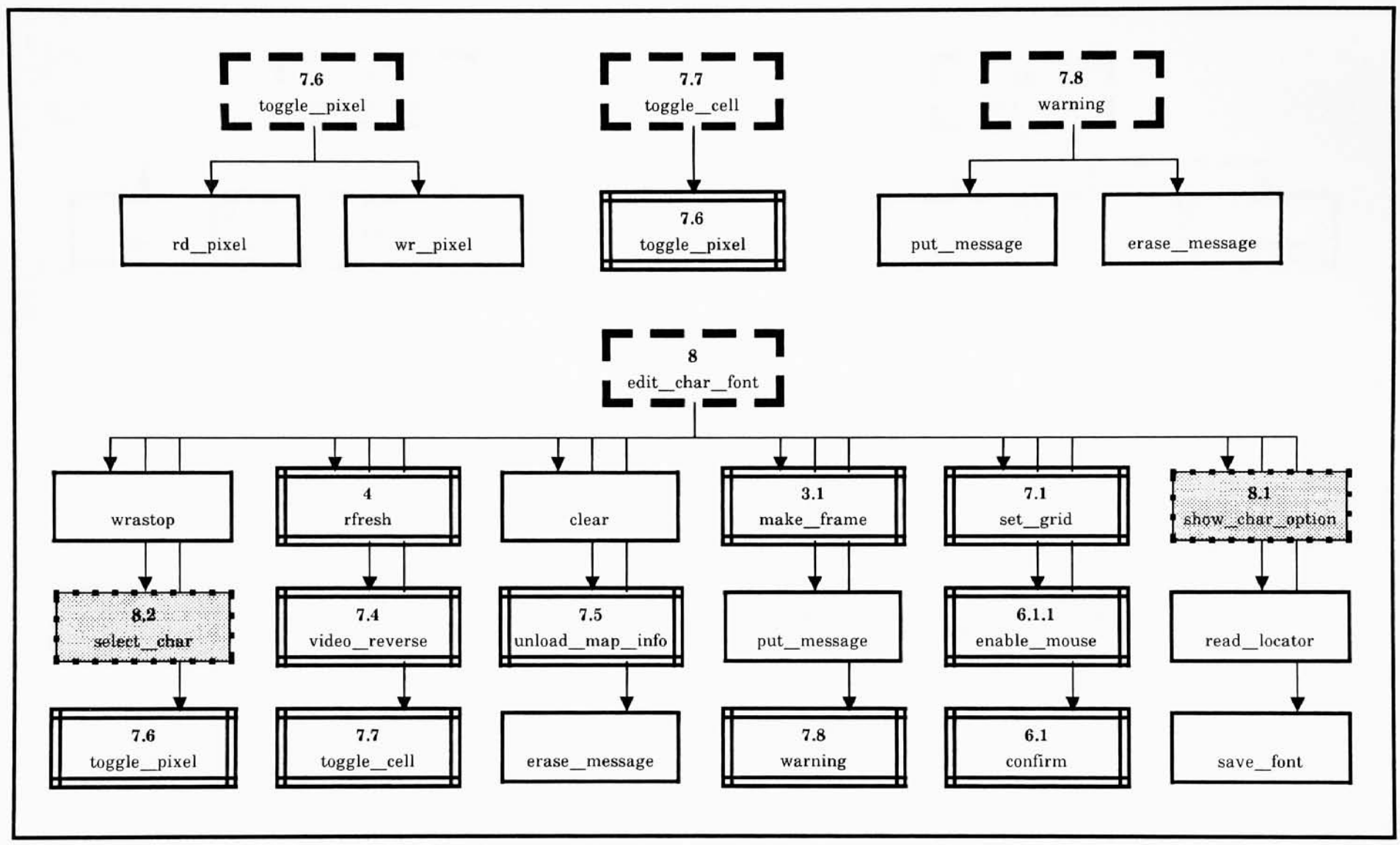
System Processes Diagram Overview

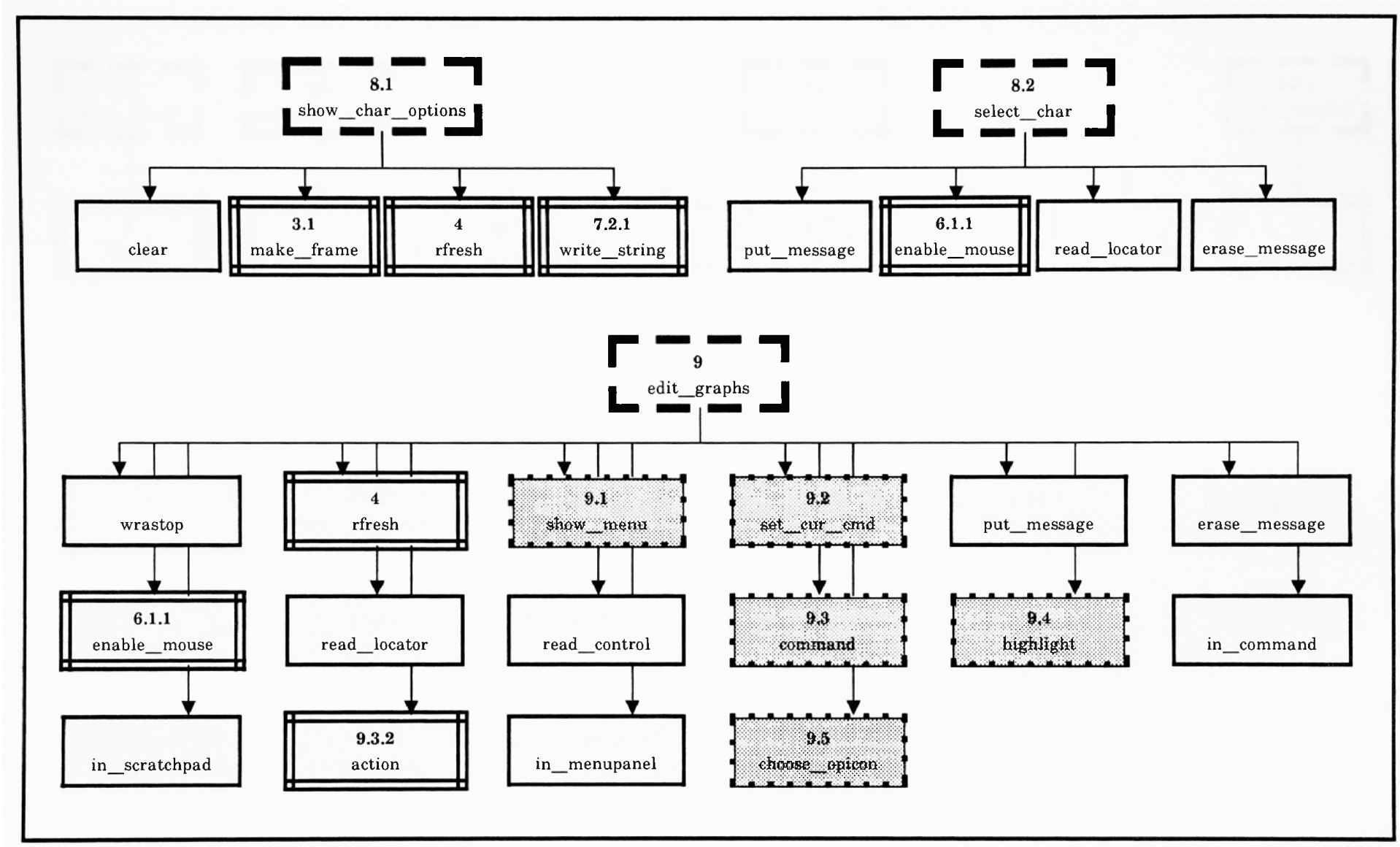

GDF System
Main
new_window
select_window
1
alloc_window_maps
2
alloc_icon_maps
load_font
load_op_icon
3
maps_set_up
4
rfresh
5
welcome_message
6
introduction
put_message
erase_message
main_selection
7
edit_op_icon
8
edit_char_font
9
edit_graphs
terminate
1
alloc_window_maps
1.1
new_map
1.1.1
error_message
2
alloc_icon_maps
1.1
new_map
1.1.1
error_message
make_form
1.1
new_map

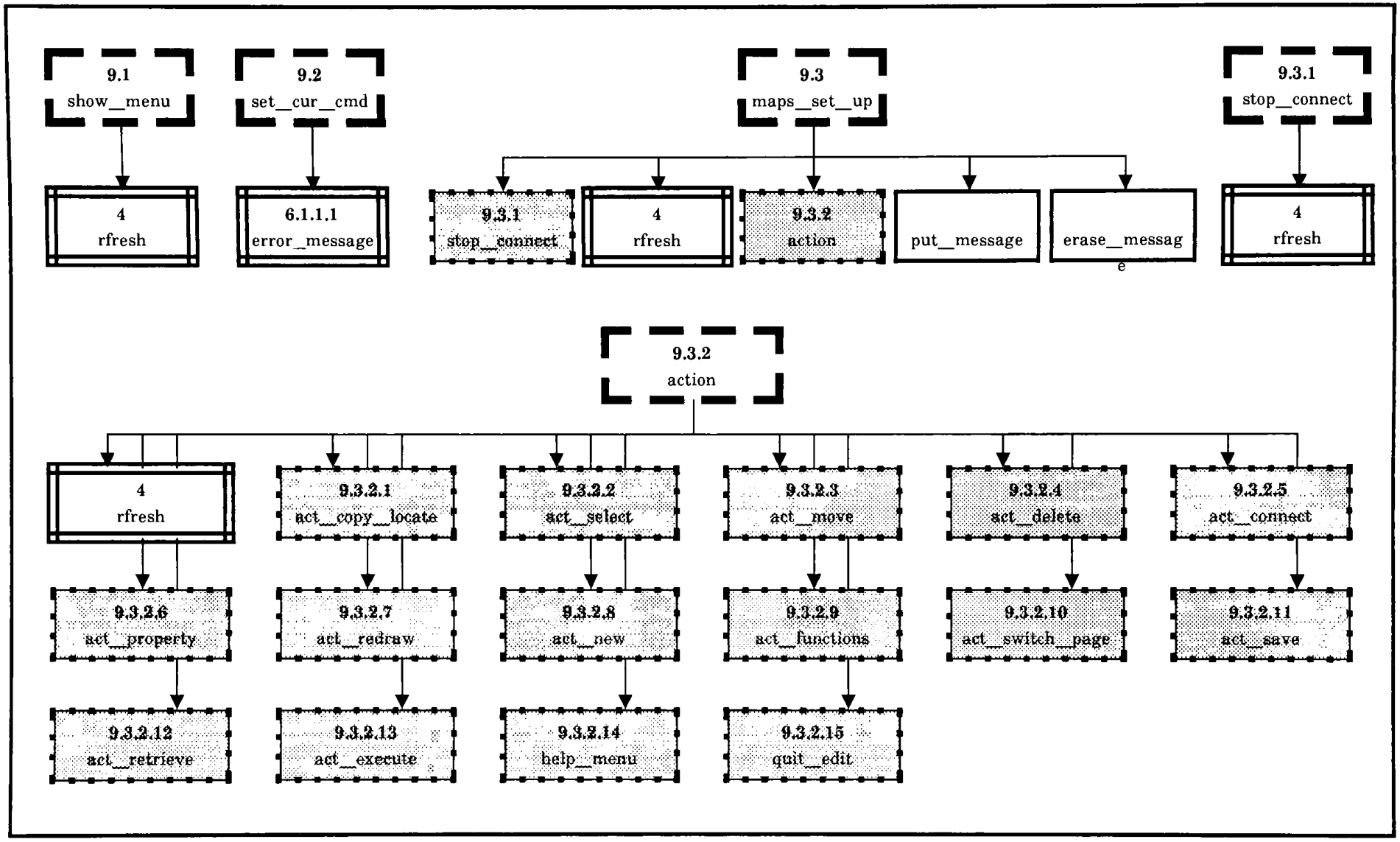

3
maps__set__up
3.1
make__frame
clear
3.2
make__string
3.3
draw__frames
3.4
fill
wr__pixel
3.1
make__frame
wr__pixel
3.2
make__string
get__char__map
3.2.1
locate__char
3.2.1
locate__char
rd__pixel
wr__pixel
3.3
draw__frames
wr__pixel
3.4
fill
wr__pixel
4
rfresh
wrastop
5
maps__set__up
5.1
show__message
5.1
show__message
get__char__map
3.2.1
locate__char
4
rfresh
6
introduction
6.1
show__message
intro__menu

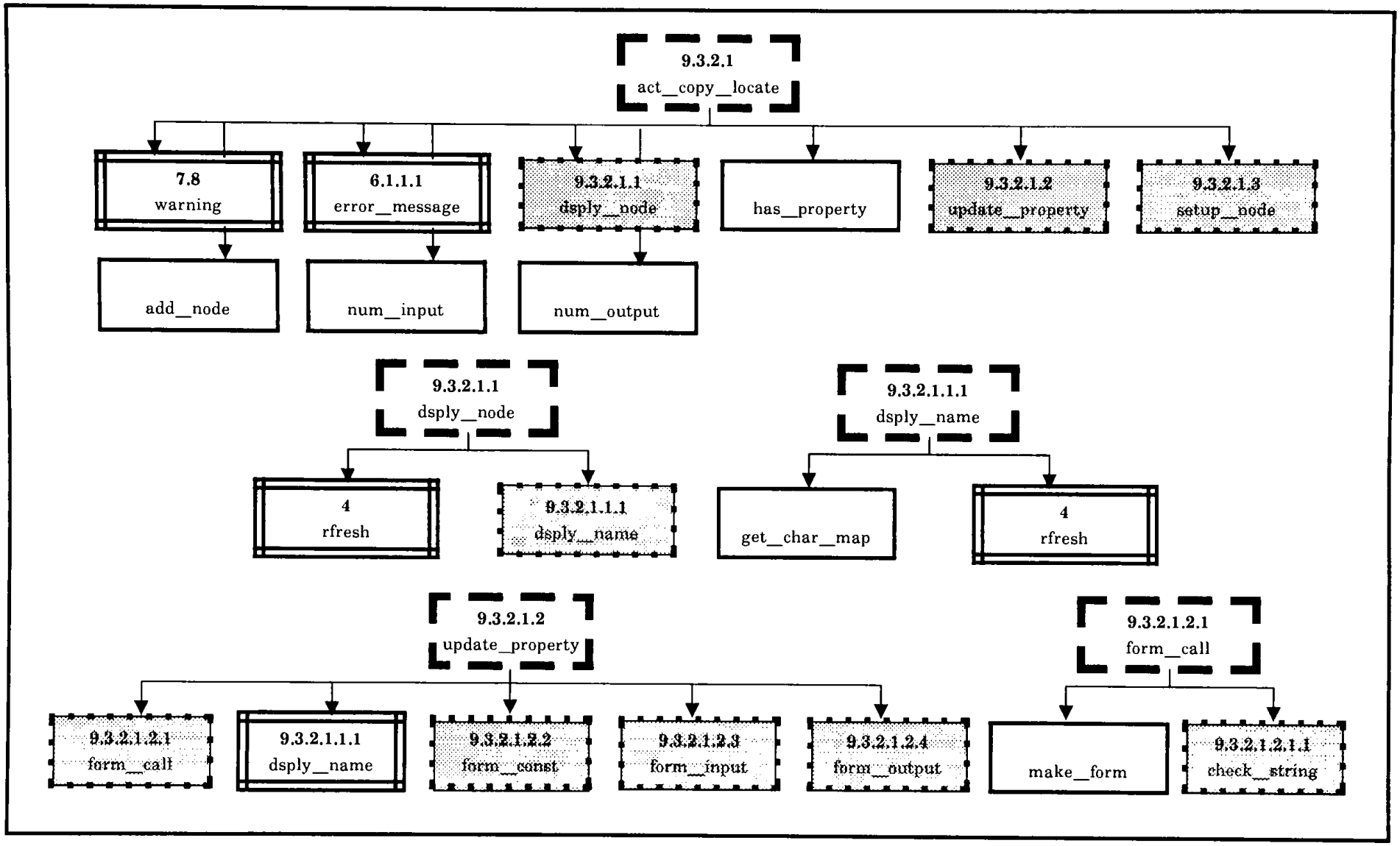

6.1
confirm
6.1.1
enable__mouse
6.1.1.1
error__message
4
rfresh
5.1
show__message
6.1.1
enable__mouse
read__locator
erase__message
6.1.1.1
error__message
make__form
7
edit__op__icon
wrastop
4
rfresh
clear
3.1
make__frame
7.1
set__grid
7.2
show__op__options
7.3
select__op
7.4
video__reverse
7.5
unload__map__info
6.1.1
enable__mouse
read__locator
7.6
toggle__pixel
7.7
toggle__cell
erase__message
7.8
warning
6.1
confirm
save__op__icon

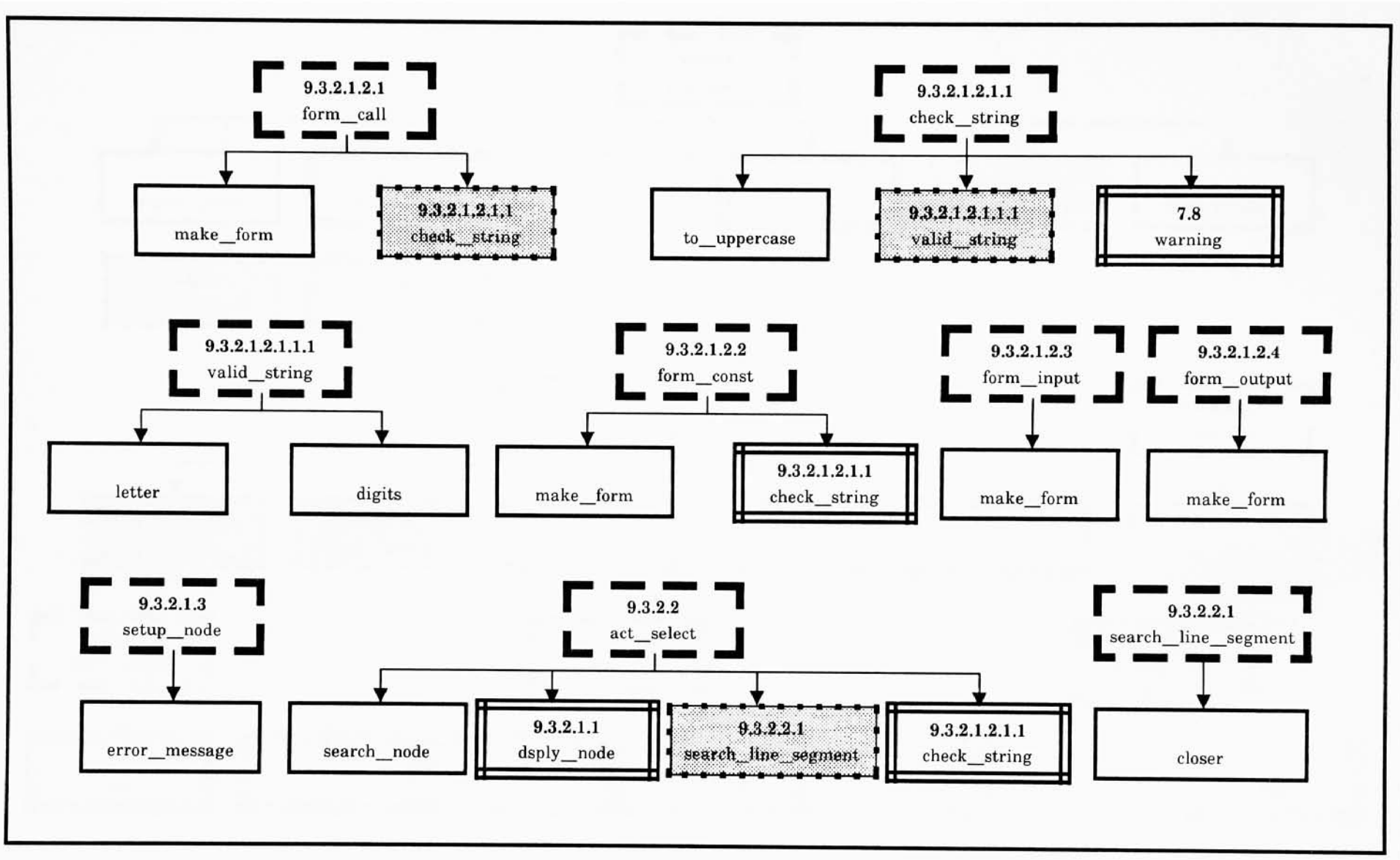

7.1
set__grid
wr__pixel
7.2
show__op__options
clear
3.1
make__frame
7.2.1
write__string
7.2.1
write__string
get__char__map
7.2.1.1
locate__map
4
rfresh
7.2.1.1
locate__map
wr__pixel
7.3
select__op
put__message
6.1.1
enable__mouse
read__locator
erase__message
7.4
video__reverse
wr__pixel
4
rfresh
7.5
unload__map__info
put__message
3.4
fill
erase__message

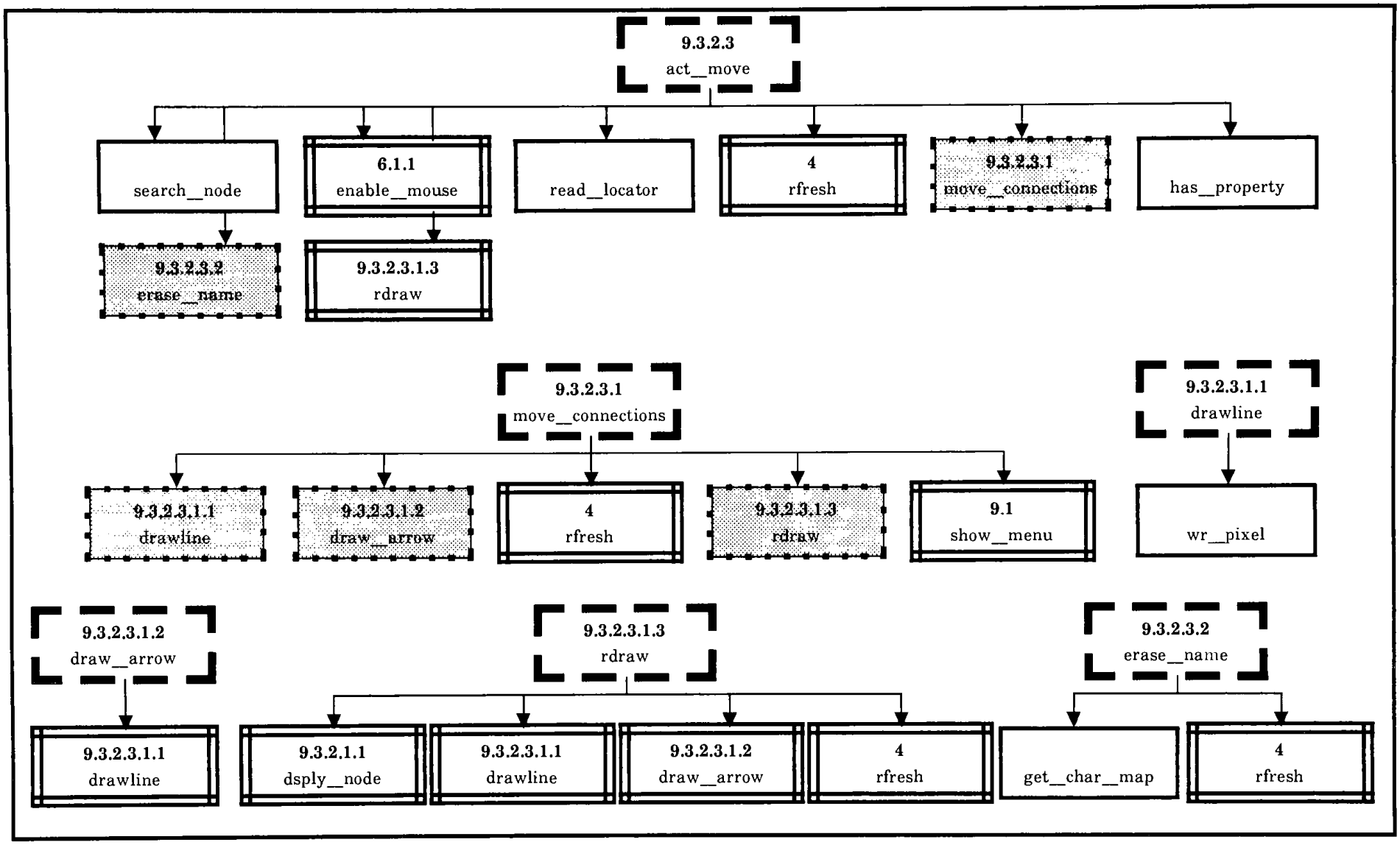

7.6
toggle_pixel
rd_pixel
wr_pixel
7.7
toggle_cell
7.6
toggle_pixel
7.8
warning
put_message
erase_message
8
edit_char_font
wrastop
8.2
select_char
7.6
toggle_pixel
4
rfresh
7.4
video_reverse
7.7
toggle_cell
clear
7.5
unload_map_info
erase_message
3.1
make_frame
put_message
7.8
warning
7.1
set_grid
6.1.1
enable_mouse
6.1
confirm
8.1
show_char_option
read_locator
save_font

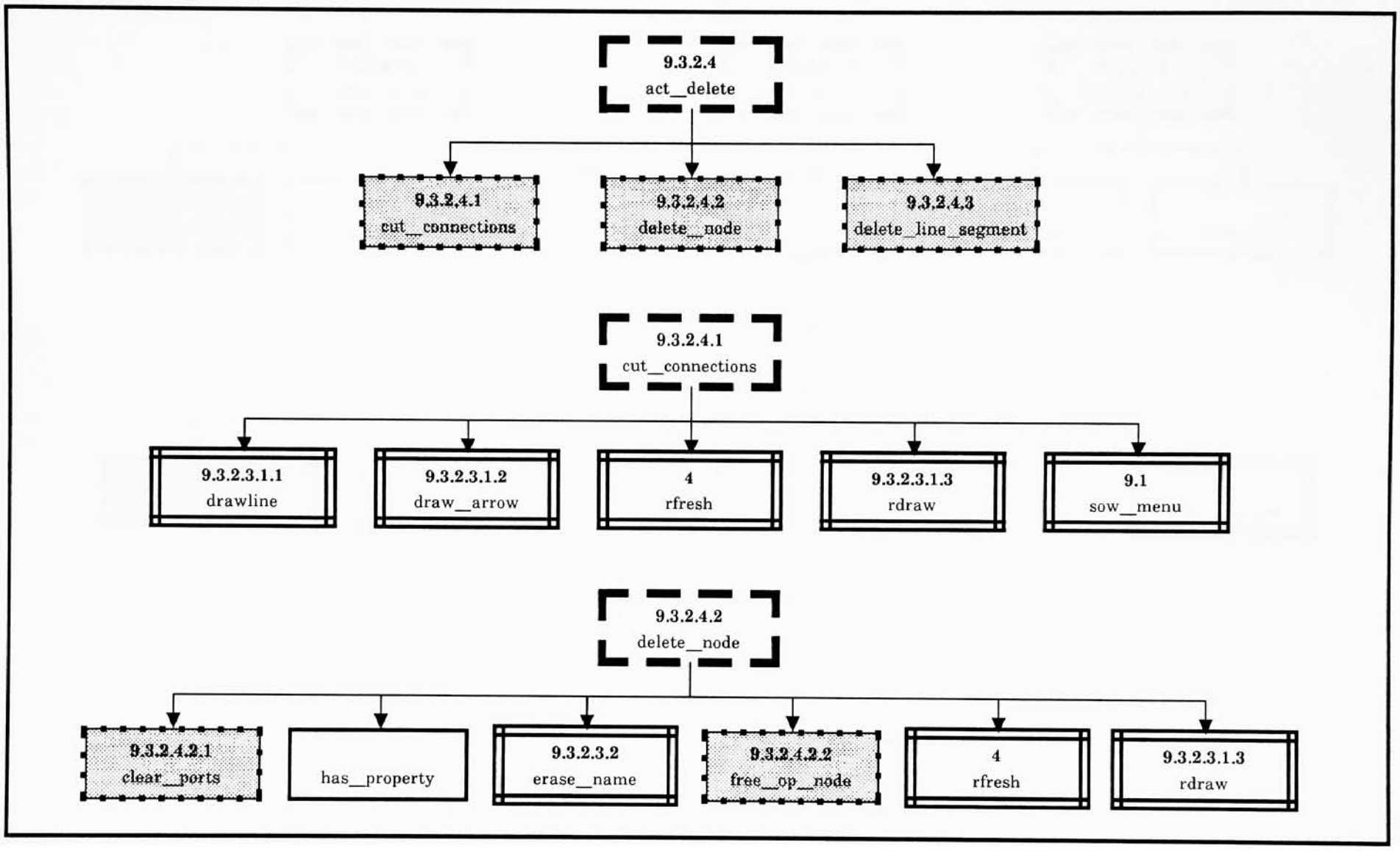

8.1
show__char__options
clear
3.1
make__frame
4
rfresh
7.2.1
write__string
8.2
select__char
put__message
6.1.1
enable__mouse
read__locator
erase__message
9
edit__graphs
wrastop
6.1.1
enable__mouse
in__scratchpad
4
rfresh
read__locator
9.3.2
action
9.1
show__menu
read__control
in__menupanel
9.2
set__cur__cmd
9.3
command
9.5
choose__opicon
put__message
9.4
highlight
erase__message
in__command

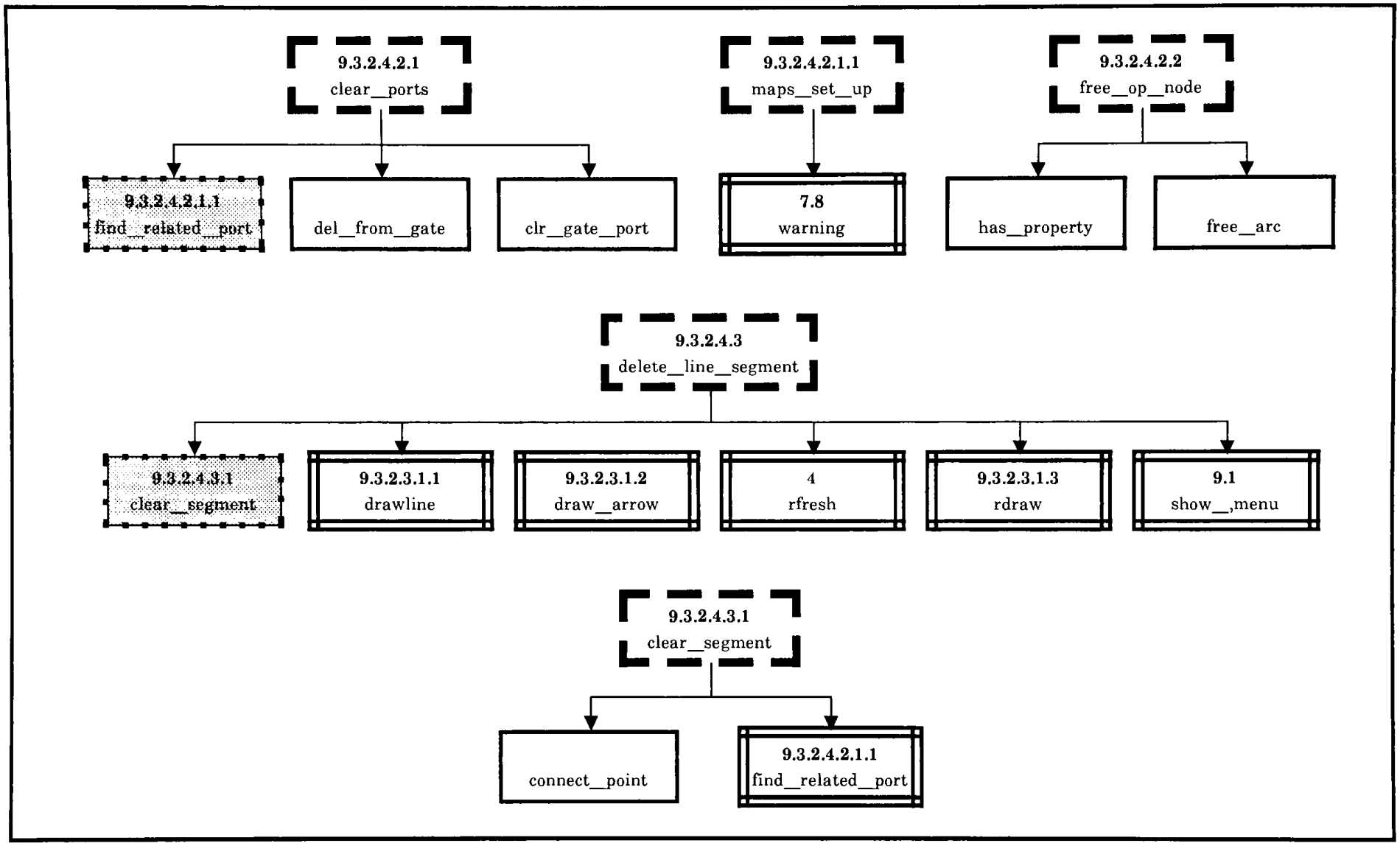

9.1
show__menu
9.2
set__cur__cmd
9.3
maps__set__up
9.3.1
stop__connect
4
rfresh
6.1.1.1
error_message
9.3.1
stop__connect
4
rfresh
9.3.2
action
put__message
erase__messag
e
4
rfresh
9.3.2
action
4
rfresh
9.3.2.1
act__copy__locate
9.3.2.2
act__select
9.3.2.3
act__move
9.3.2.4
act__delete
9.3.2.5
act__connect
9.3.2.6
act__property
9.3.2.7
act__redraw
9.3.2.8
act__new
9.3.2.9
act__functions
9.3.2.10
act__switch__page
9.3.2.11
act__save
9.3.2.12
act__retrieve
9.3.2.13
act__execute
9.3.2.14
help__menu
9.3.2.15
quit__edit

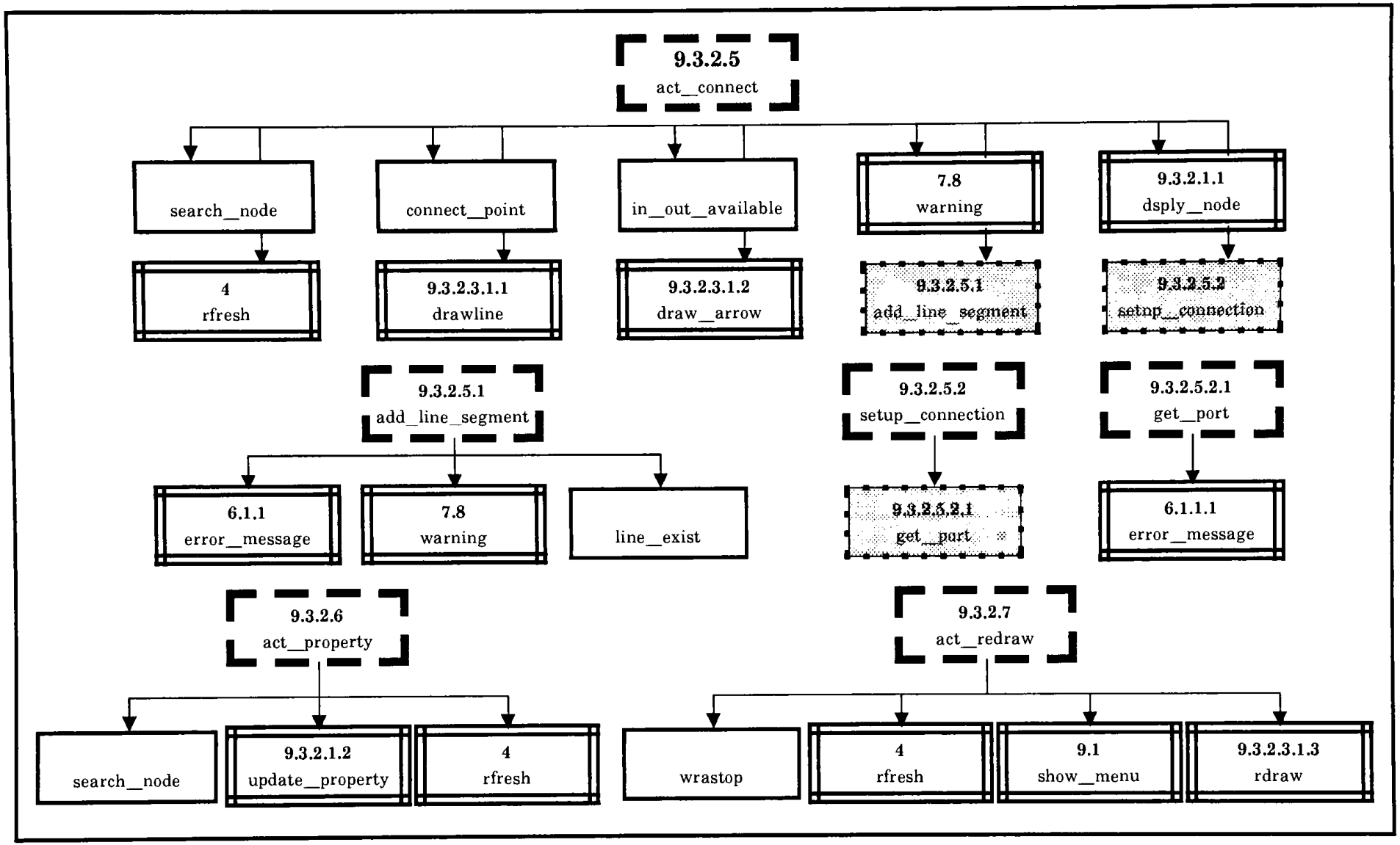

9.3.2.1
act__copy__locate
7.8
warning
6.1.1.1
error__message
9.3.2.1.1
dsply__node
has__property
9.3.2.1.2
update__property
9.3.2.1.3
setup__node
add__node
num__input
num__output
9.3.2.1.1
dsply__node
4
rfresh
9.3.2.1.1.1
dsply__name
9.3.2.1.1.1
dsply__name
get__char__map
4
rfresh
9.3.2.1.2
update_property
9.3.2.1.2.1
form__call
9.3.2.1.1.1
dsply__name
9.3.2.1.2.2
form__const
9.3.2.1.2.3
form__input
9.3.2.1.2.4
form__output
9.3.2.1.2.1
form__call
make__form
9.3.2.1.2.1.1
check__string

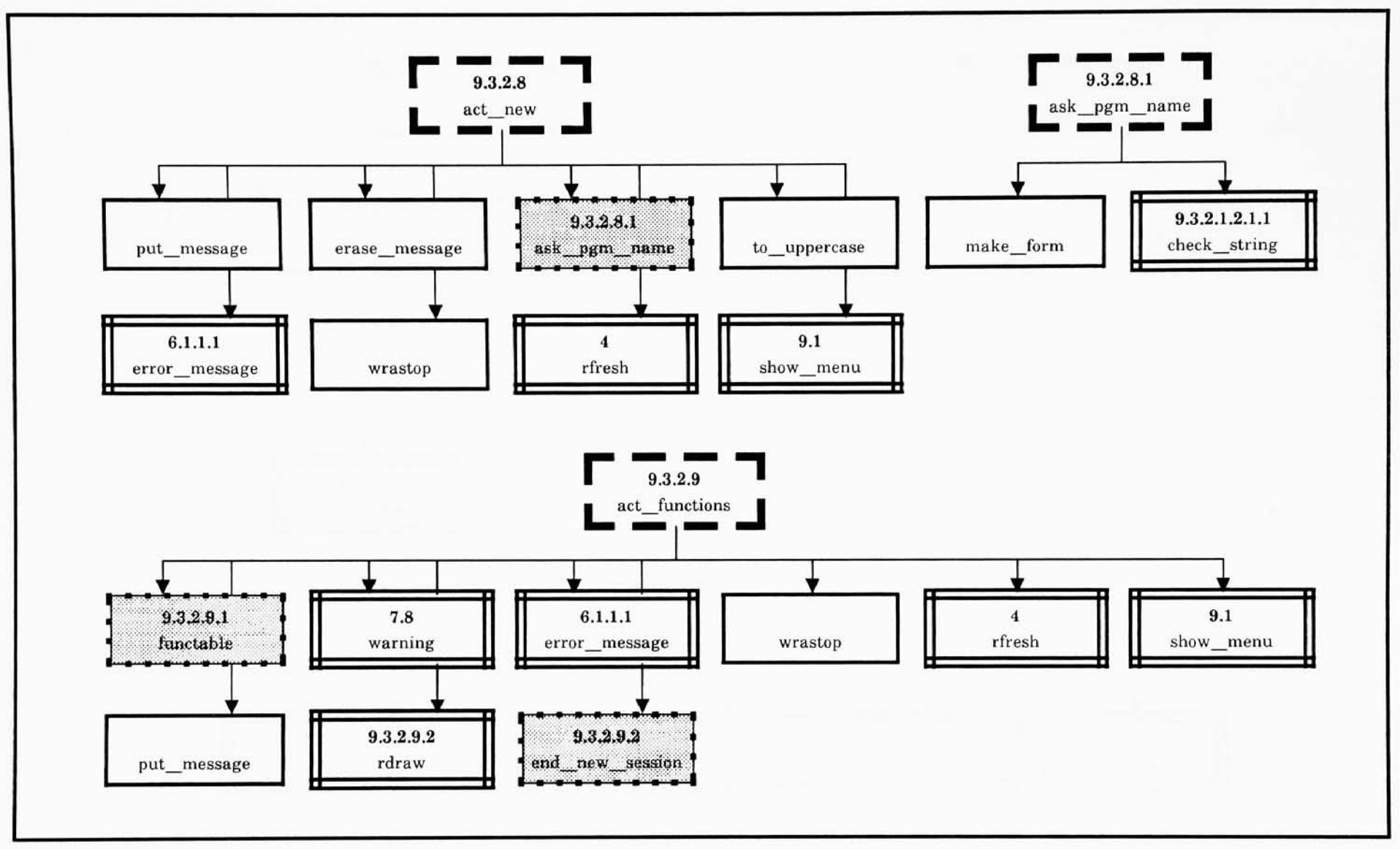

9.3.2.1.2.1
form__call
make__form
9.3.2.1.2.1.1
check__string
9.3.2.1.2.1.1
check__string
to__uppercase
9.3.2.1.2.1.1.1
valid__string
7.8
warning
9.3.2.1.2.1.1.1
valid__string
letter
digits
9.3.2.1.2.2
form__const
make__form
9.3.2.1.2.1.1
check__string
9.3.2.1.2.3
form__input
make__form
9.3.2.1.2.4
form__output
make__form
9.3.2.1.3
setup__node
error__message
9.3.2.2
act__select
search__node
9.3.2.1.1
dsply__node
9.3.2.2.1
search__line__segment
9.3.2.1.2.1.1
check__string
9.3.2.2.1
search__line__segment
closer

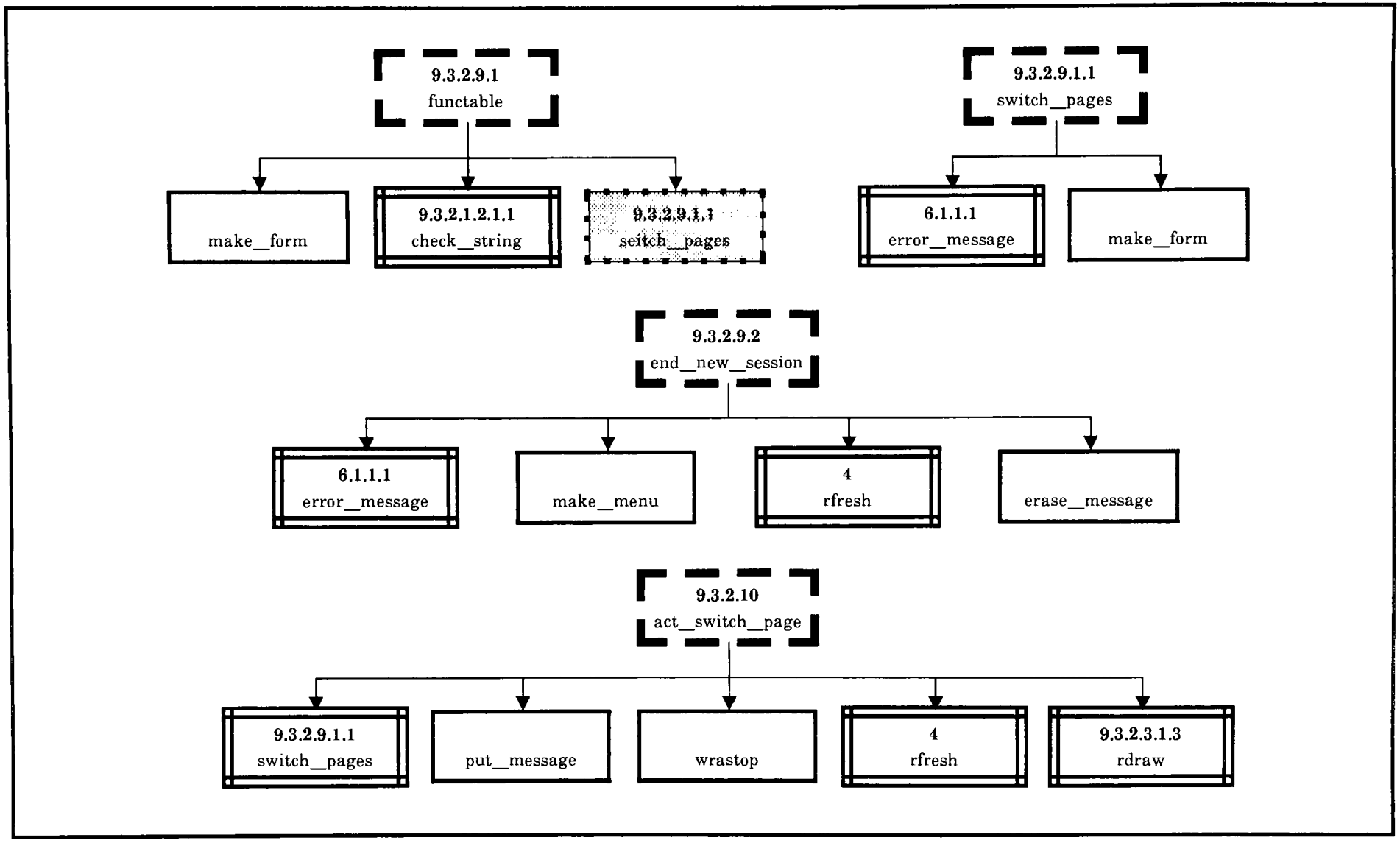

9.3.2.3
act__move
search__node
6.1.1
enable__mouse
read__locator
4
rfresh
9.3.2.3.1
move__connections
has__property
9.3.2.3.2
erase__name
9.3.2.3.1.3
rdraw
9.3.2.3.1
move__connections
9.3.2.3.1.1
drawline
9.3.2.3.1.1
drawline
9.3.2.3.1.2
draw__arrow
4
rfresh
9.3.2.3.1.3
rdraw
9.1
show__menu
wr__pixel
9.3.2.3.1.2
draw__arrow
9.3.2.3.1.3
rdraw
9.3.2.3.2
erase__name
9.3.2.3.1.1
drawline
9.3.2.1.1
dsply__node
9.3.2.3.1.1
drawline
9.3.2.3.1.2
draw__arrow
4
rfresh
get__char__map
4
rfresh

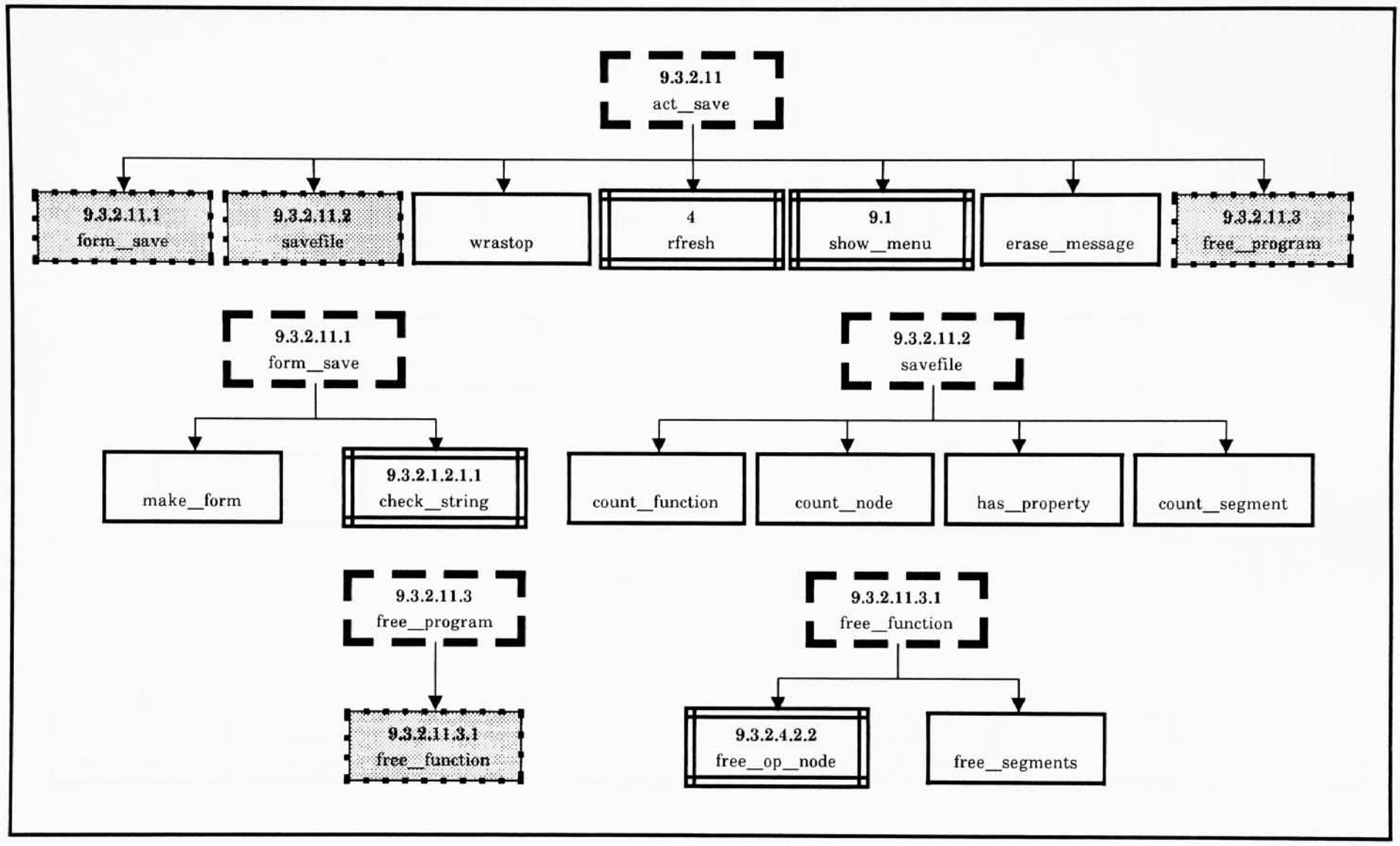

9.3.2.4
act__delete
9.3.2.4.1
cut__connections
9.3.2.4.2
delete__node
9.3.2.4.3
delete_line_segment
9.3.2.4.1
cut__connections
9.3.2.3.1.1
drawline
9.3.2.3.1.2
draw__arrow
4
rfresh
9.3.2.3.1.3
rdraw
9.1
sow__menu
9.3.2.4.2
delete__node
9.3.2.4.2.1
clear__ports
has__property
9.3.2.3.2
erase__name
9.3.2.4.2.2
free__op__node
4
rfresh
9.3.2.3.1.3
rdraw

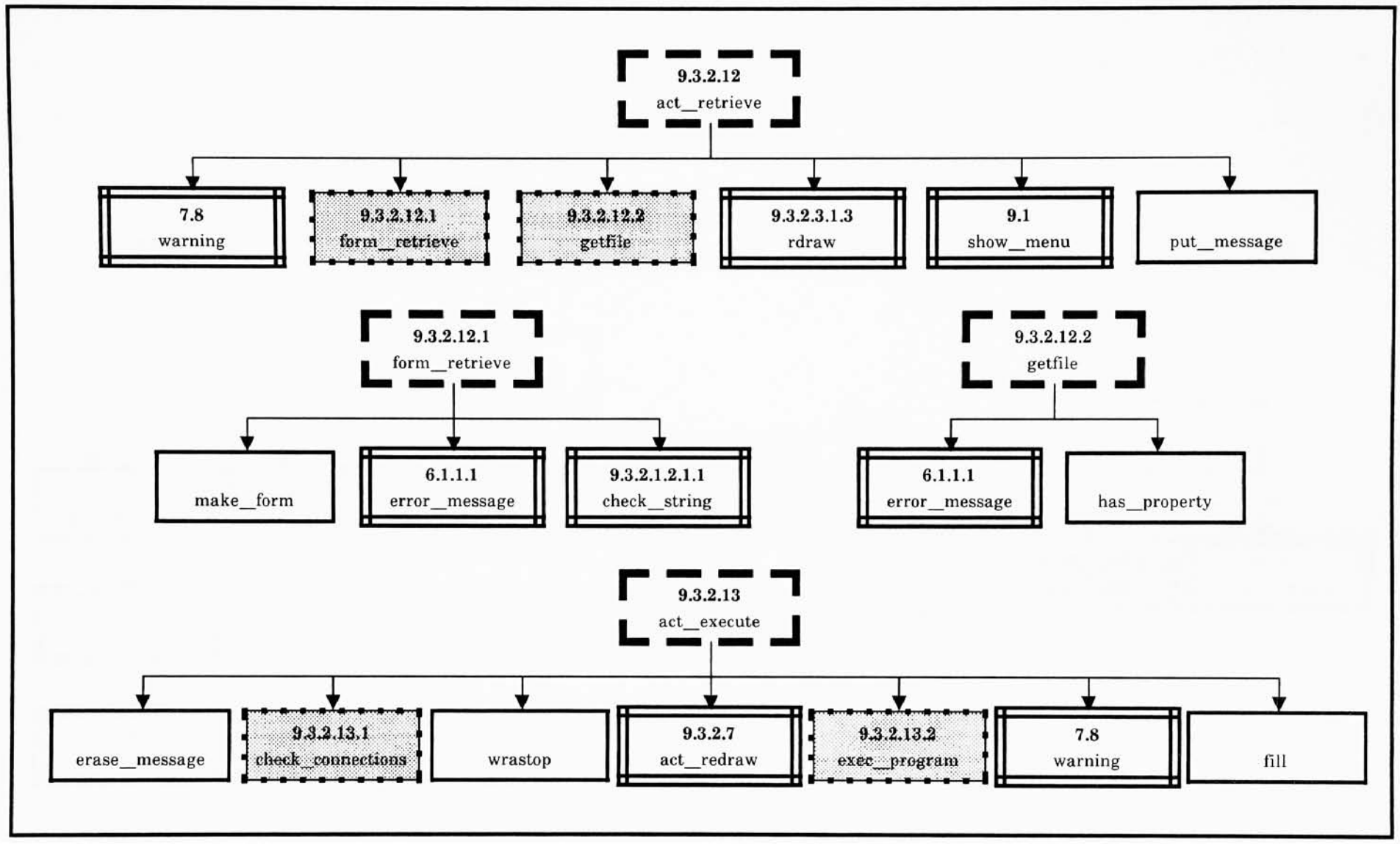

9.3.2.4.2.1
clear__ports
9.3.2.4.2.1.1
find__related__port
del__from__gate
clr__gate__port
9.3.2.4.2.1.1
maps__set__up
7.8
warning
9.3.2.4.2.2
free__op__node
has__property
free__arc
9.3.2.4.3
delete__line__segment
9.3.2.4.3.1
clear__segment
9.3.2.3.1.1
drawline
9.3.2.3.1.2
draw__arrow
4
rfresh
9.3.2.3.1.3
rdraw
9.1
show__,menu
9.3.2.4.3.1
clear__segment
connect__point
9.3.2.4.2.1.1
find__related__port

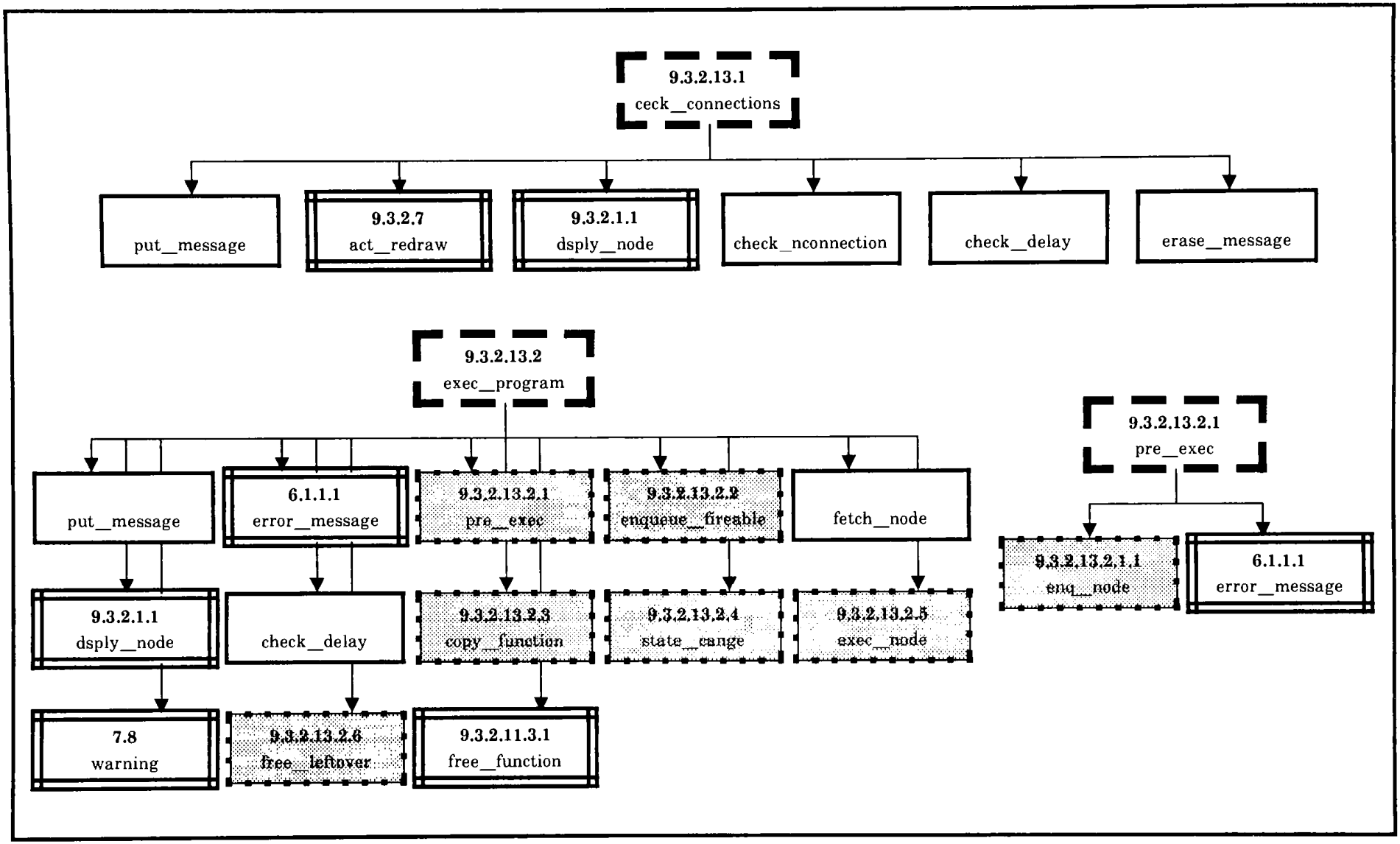

9.3.2.5
act__connect
search__node
connect__point
in__out__available
7.8
warning
9.3.2.1.1
dsply__node
4
rfresh
9.3.2.3.1.1
drawline
9.3.2.3.1.2
draw__arrow
9.3.2.5.1
add_line_segment
9.3.2.5.2
setup__connection
9.3.2.5.1
add_line_segment
9.3.2.5.2
setup__connection
9.3.2.5.2.1
get__port
6.1.1
error__message
7.8
warning
line__exist
9.3.2.5.2.1
get__port
6.1.1.1
error__message
9.3.2.6
act__property
9.3.2.7
act__redraw
search__node
9.3.2.1.2
update__property
4
rfresh
wrastop
4
rfresh
9.1
show__menu
9.3.2.3.1.3
rdraw

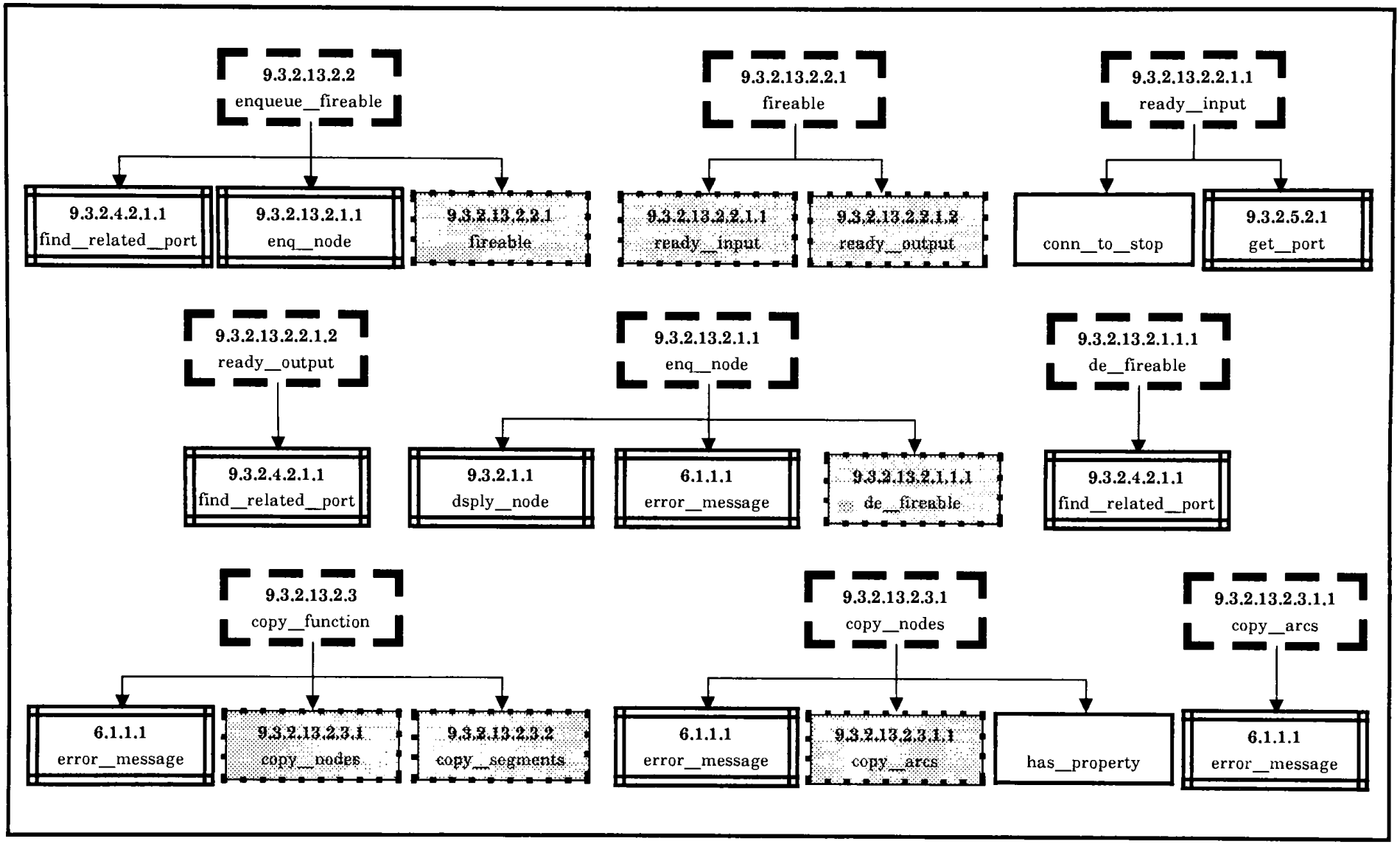

9.3.2.8
act__new
put__message
erase__message
9.3.2.8.1
ask__pgm__name
to__uppercase
6.1.1.1
error__message
wrastop
4
rfresh
9.1
show__menu
9.3.2.8.1
ask__pgm__name
make__form
9.3.2.1.2.1.1
check__string
9.3.2.9
act__functions
9.3.2.9.1
functable
7.8
warning
6.1.1.1
error__message
wrastop
4
rfresh
9.1
show__menu
put__message
9.3.2.9.2
rdraw
9.3.2.9.2
end__new__session

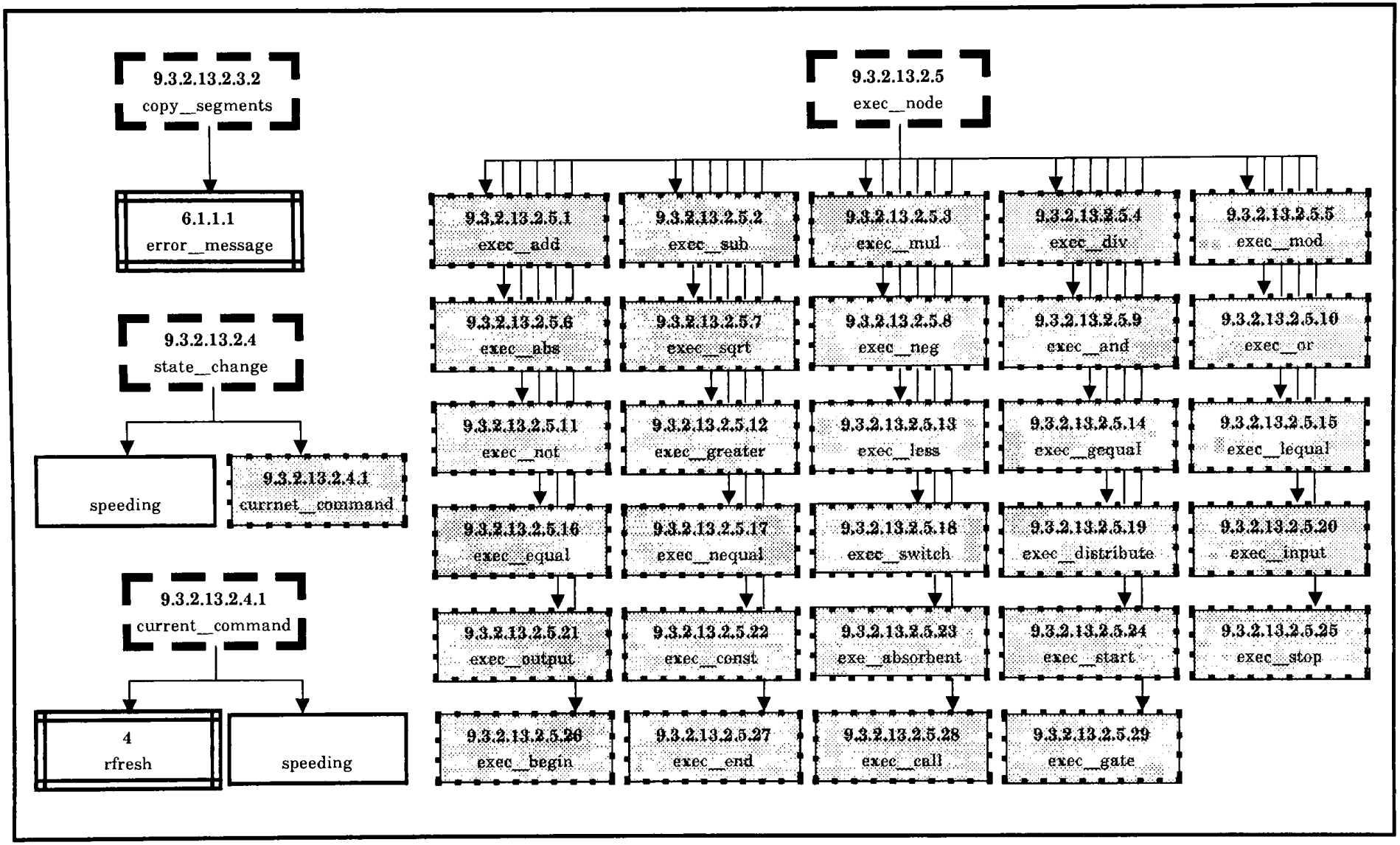

9.3.2.9.1
functable
make_form
9.3.2.1.2.1.1
check_string
9.3.2.9.1.1
switch_pages
9.3.2.9.1.1
switch_pages
6.1.1.1
error_message
make_form
9.3.2.9.2
end_new_session
6.1.1.1
error_message
make_menu
4
rfresh
erase_message
9.3.2.10
act_switch_page
9.3.2.9.1.1
switch_pages
put_message
wrastop
4
rfresh
9.3.2.3.1.3
rdraw

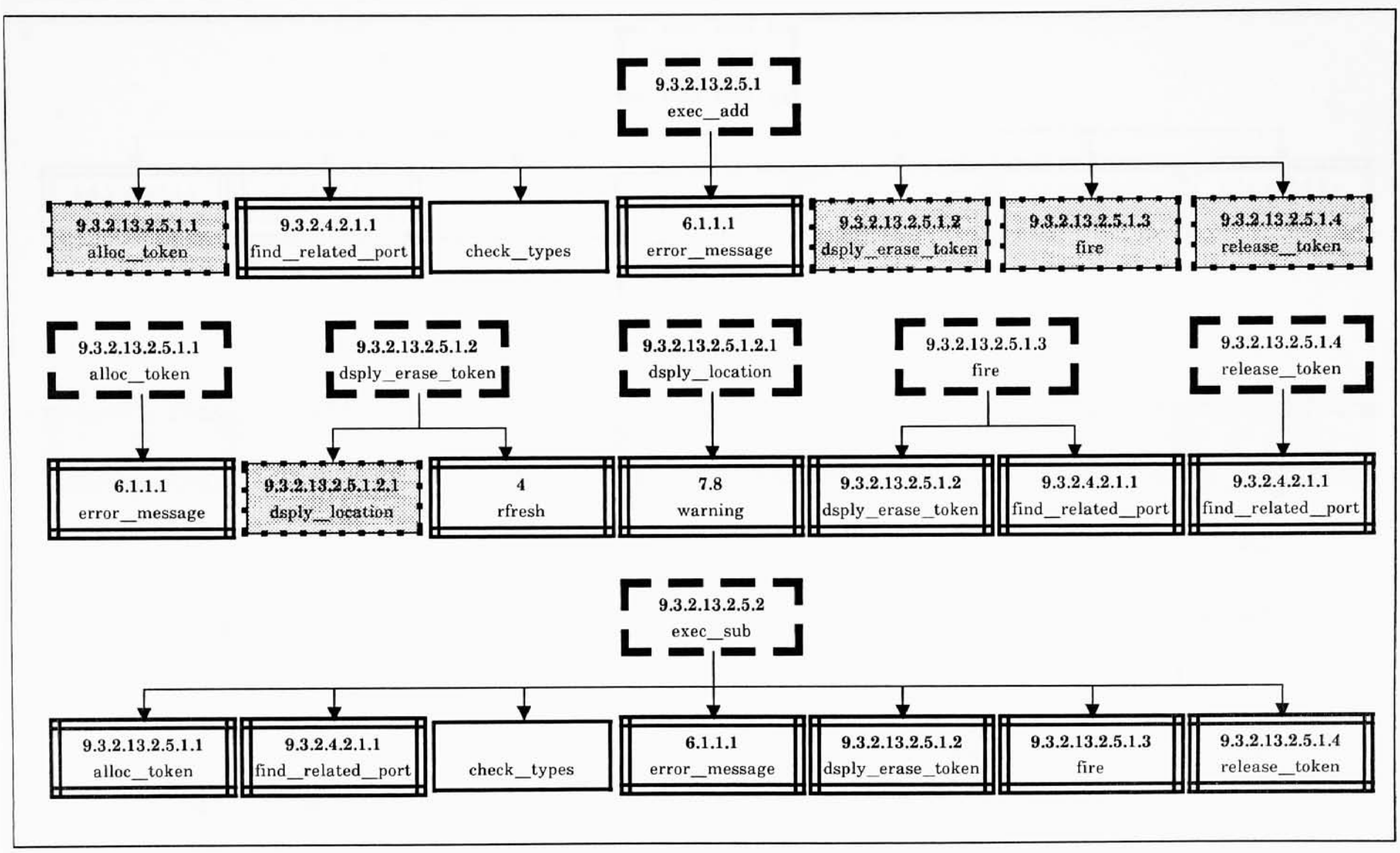

9.3.2.11
act__save
9.3.2.11.1
form__save
9.3.2.11.2
savefile
wrastop
4
rfresh
9.1
show__menu
erase__message
9.3.2.11.3
free__program
9.3.2.11.1
form__save
make__form
9.3.2.1.2.1.1
check__string
9.3.2.11.2
savefile
count__function
count__node
has__property
count__segment
9.3.2.11.3
free__program
9.3.2.11.3.1
free__function
9.3.2.11.3.1
free__function
9.3.2.4.2.2
free__op__node
free__segments

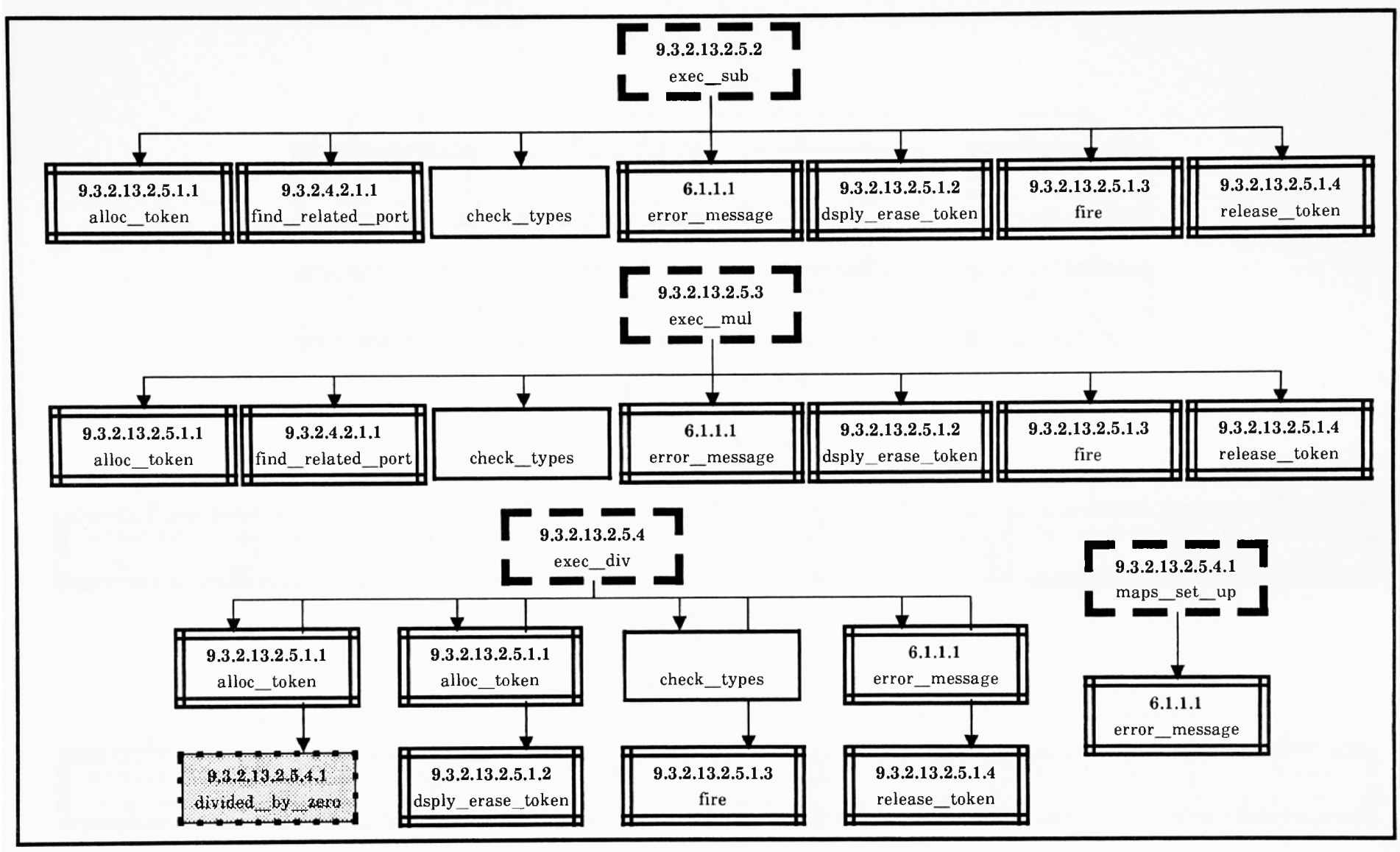

9.3.2.12
act__retrieve
7.8
warning
9.3.2.12.1
form__retrieve
9.3.2.12.2
getfile
9.3.2.3.1.3
rdraw
9.1
show__menu
put__message
9.3.2.12.1
form__retrieve
make__form
6.1.1.1
error__message
9.3.2.1.2.1.1
check__string
9.3.2.12.2
getfile
6.1.1.1
error__message
has__property
9.3.2.13
act__execute
erase__message
9.3.2.13.1
check__connections
wrastop
9.3.2.7
act__redraw
9.3.2.13.2
exec__program
7.8
warning
fill

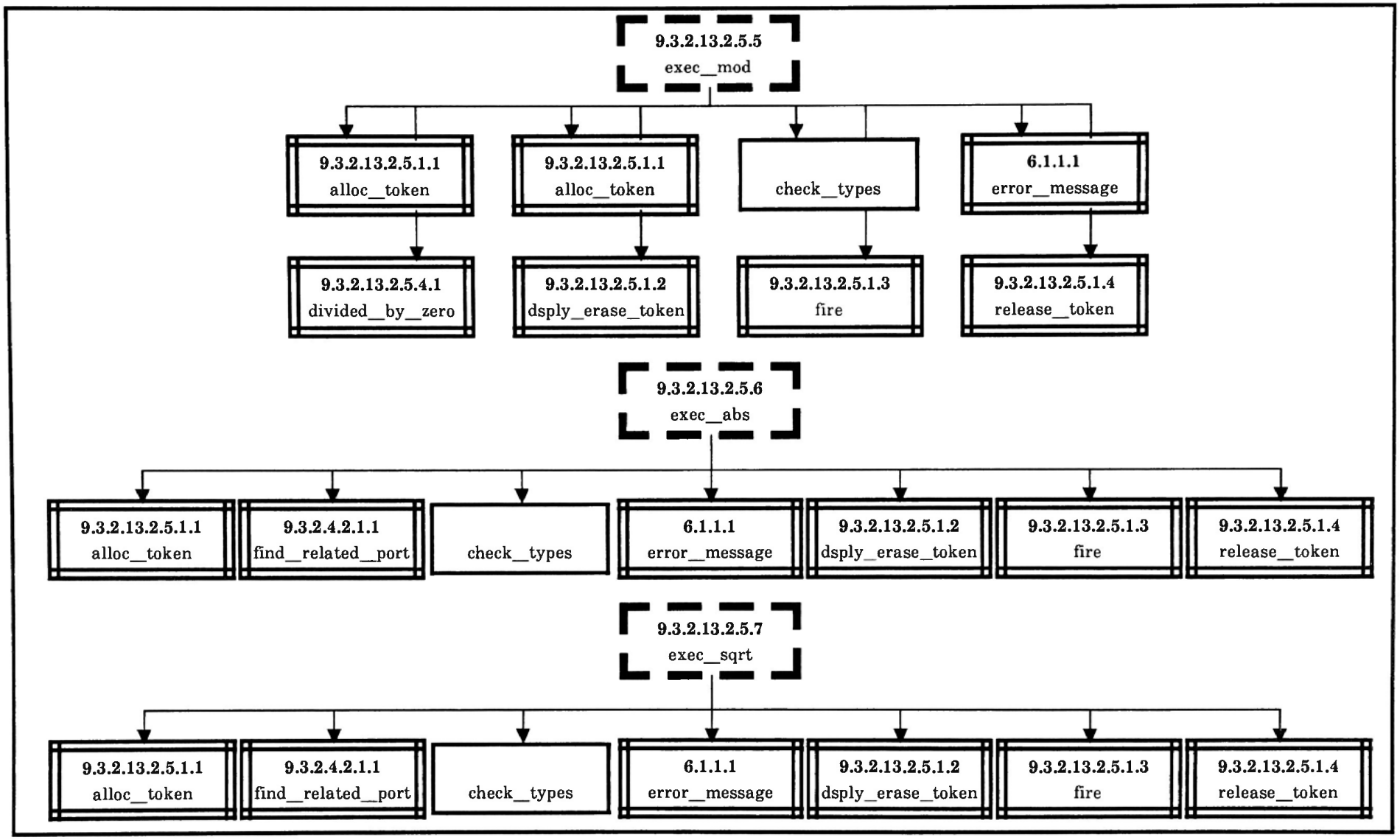

9.3.2.13.1
ceck__connections
put__message
9.3.2.7
act__redraw
9.3.2.1.1
dsply__node
check_nconnection
check__delay
erase__message
9.3.2.13.2
exec__program
put__message
6.1.1.1
error__message
9.3.2.13.2.1
pre__exec
9.3.2.13.2.2
enqueue__fireable
fetch__node
9.3.2.1.1
dsply__node
check__delay
9.3.2.13.2.3
copy__function
9.3.2.13.2.4
state__cange
9.3.2.13.2.5
exec__node
7.8
warning
9.3.2.13.2.6
free__leftover
9.3.2.11.3.1
free__function
9.3.2.13.2.1
pre__exec
9.3.2.13.2.1.1
enq__node
6.1.1.1
error__message

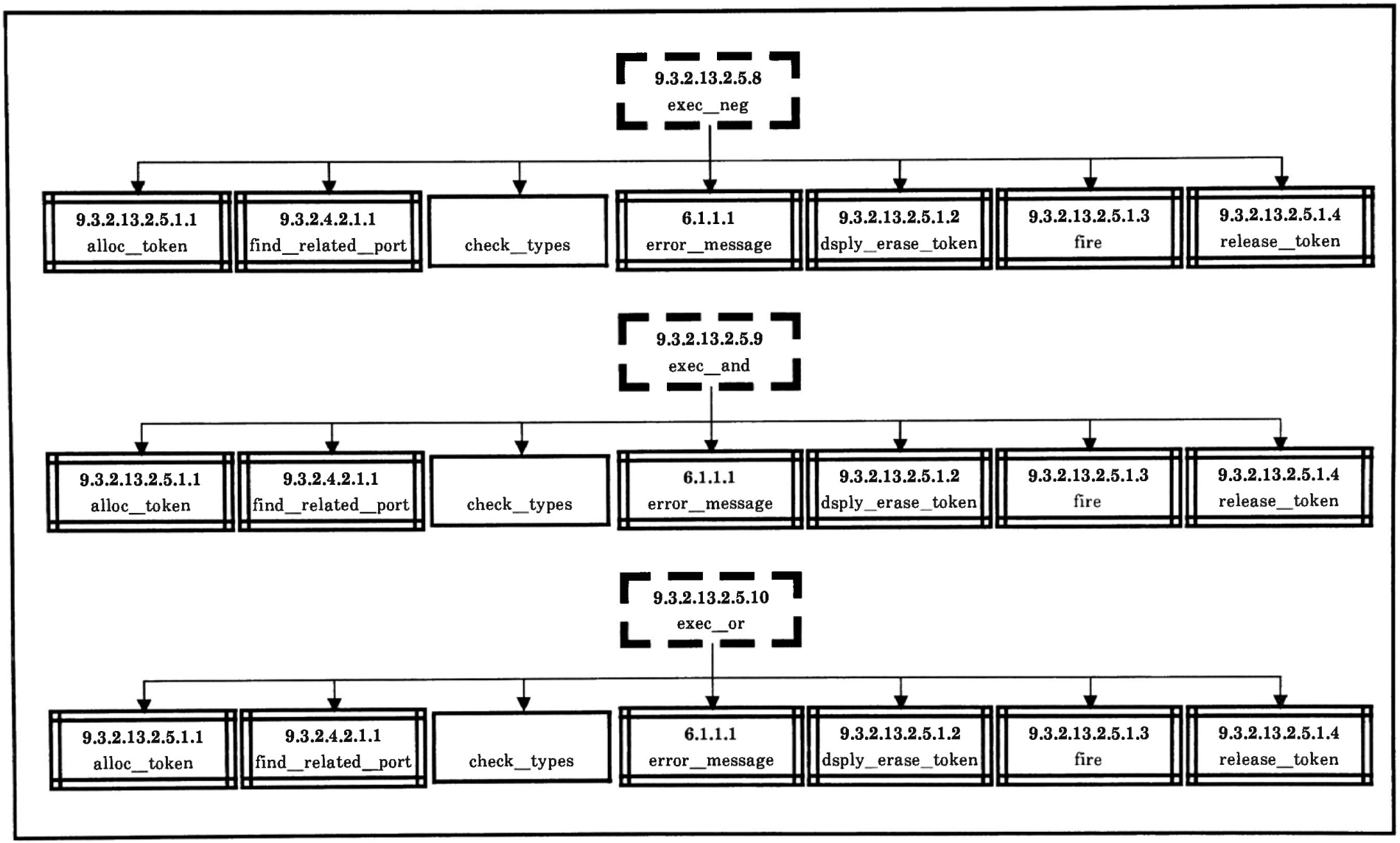

9.3.2.13.2.2
enqueue__fireable
9.3.2.4.2.1.1
find__related__port
9.3.2.13.2.1.1
enq__node
9.3.2.13.2.2.1
fireable
9.3.2.13.2.2.1
fireable
9.3.2.13.2.2.1.1
ready__input
9.3.2.13.2.2.1.2
ready__output
9.3.2.13.2.2.1.1
ready__input
conn__to__stop
9.3.2.5.2.1
get__port
9.3.2.13.2.2.1.2
ready__output
9.3.2.4.2.1.1
find__related__port
9.3.2.13.2.1.1
enq__node
9.3.2.1.1
dsply__node
6.1.1.1
error__message
9.3.2.13.2.1.1.1
de__fireable
9.3.2.13.2.1.1.1
de__fireable
9.3.2.4.2.1.1
find__related__port
9.3.2.13.2.3
copy__function
6.1.1.1
error__message
9.3.2.13.2.3.1
copy__nodes
9.3.2.13.2.3.2
copy__segments
9.3.2.13.2.3.1
copy__nodes
6.1.1.1
error__message
9.3.2.13.2.3.1.1
copy__arcs
has__property
9.3.2.13.2.3.1.1
copy__arcs
6.1.1.1
error__message

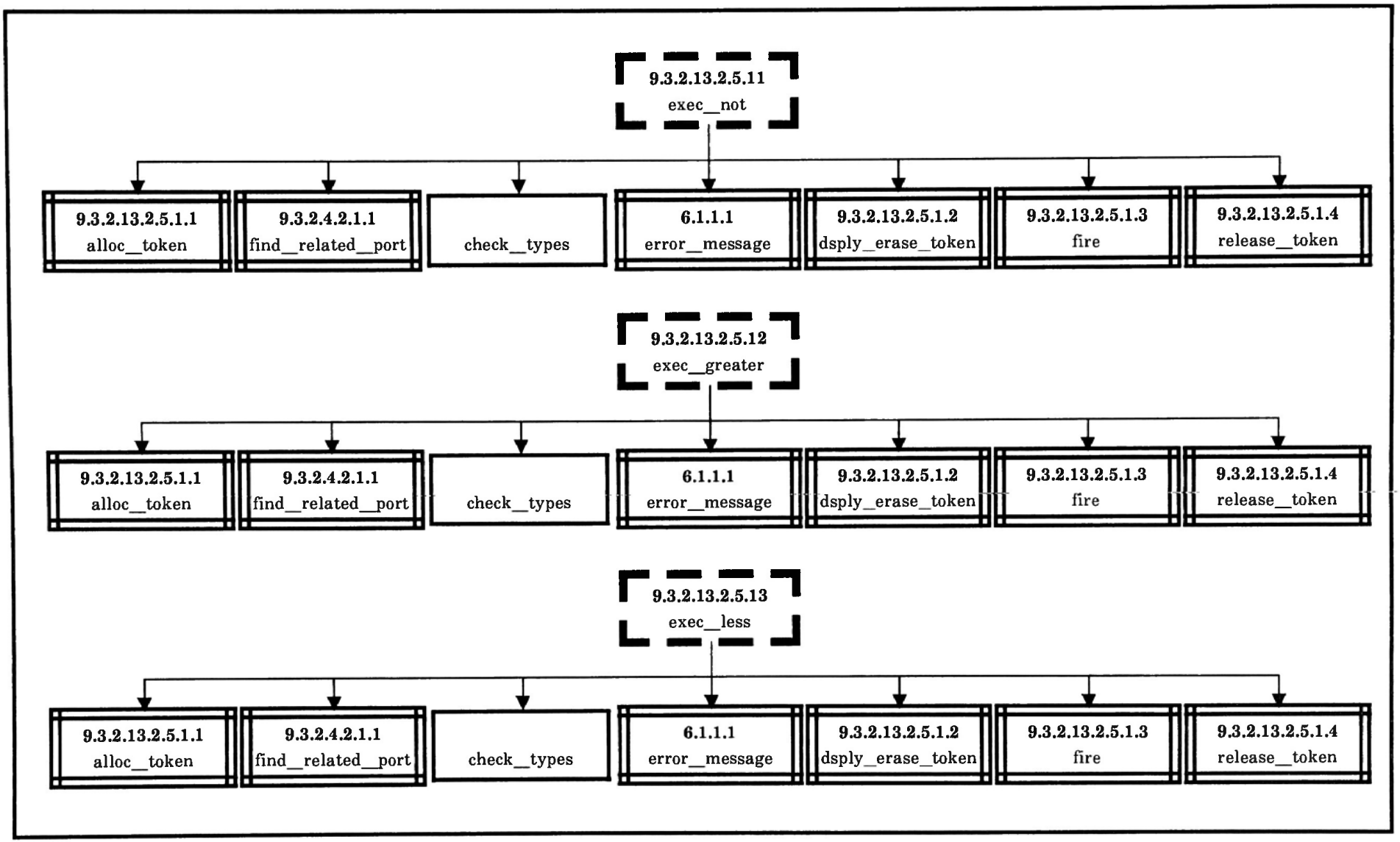

9.3.2.13.2.3.2
copy__segments
6.1.1.1
error__message
9.3.2.13.2.4
state__change
speeding
9.3.2.13.2.4.1
currnet__command
9.3.2.13.2.4.1
current__command
4
rfresh
speeding
9.3.2.13.2.5
exec__node
9.3.2.13.2.5.1
exec__add
9.3.2.13.2.5.2
exec__sub
9.3.2.13.2.5.3
exec__mul
9.3.2.13.2.5.4
exec__div
9.3.2.13.2.5.5
exec__mod
9.3.2.13.2.5.6
exec__abs
9.3.2.13.2.5.7
exec__sqrt
9.3.2.13.2.5.8
exec__neg
9.3.2.13.2.5.9
exec__and
9.3.2.13.2.5.10
exec__or
9.3.2.13.2.5.11
exec__not
9.3.2.13.2.5.12
exec__greater
9.3.2.13.2.5.13
exec__less
9.3.2.13.2.5.14
exec__gequal
9.3.2.13.2.5.15
exec__lequal
9.3.2.13.2.5.16
exec__equal
9.3.2.13.2.5.17
exec__nequal
9.3.2.13.2.5.18
exec__switch
9.3.2.13.2.5.19
exec__distribute
9.3.2.13.2.5.20
exec__input
9.3.2.13.2.5.21
exec__output
9.3.2.13.2.5.22
exec__const
9.3.2.13.2.5.23
exe__absorbent
9.3.2.13.2.5.24
exec__start
9.3.2.13.2.5.25
exec__stop
9.3.2.13.2.5.26
exec__begin
9.3.2.13.2.5.27
exec__end
9.3.2.13.2.5.28
exec__call
9.3.2.13.2.5.29
exec__gate

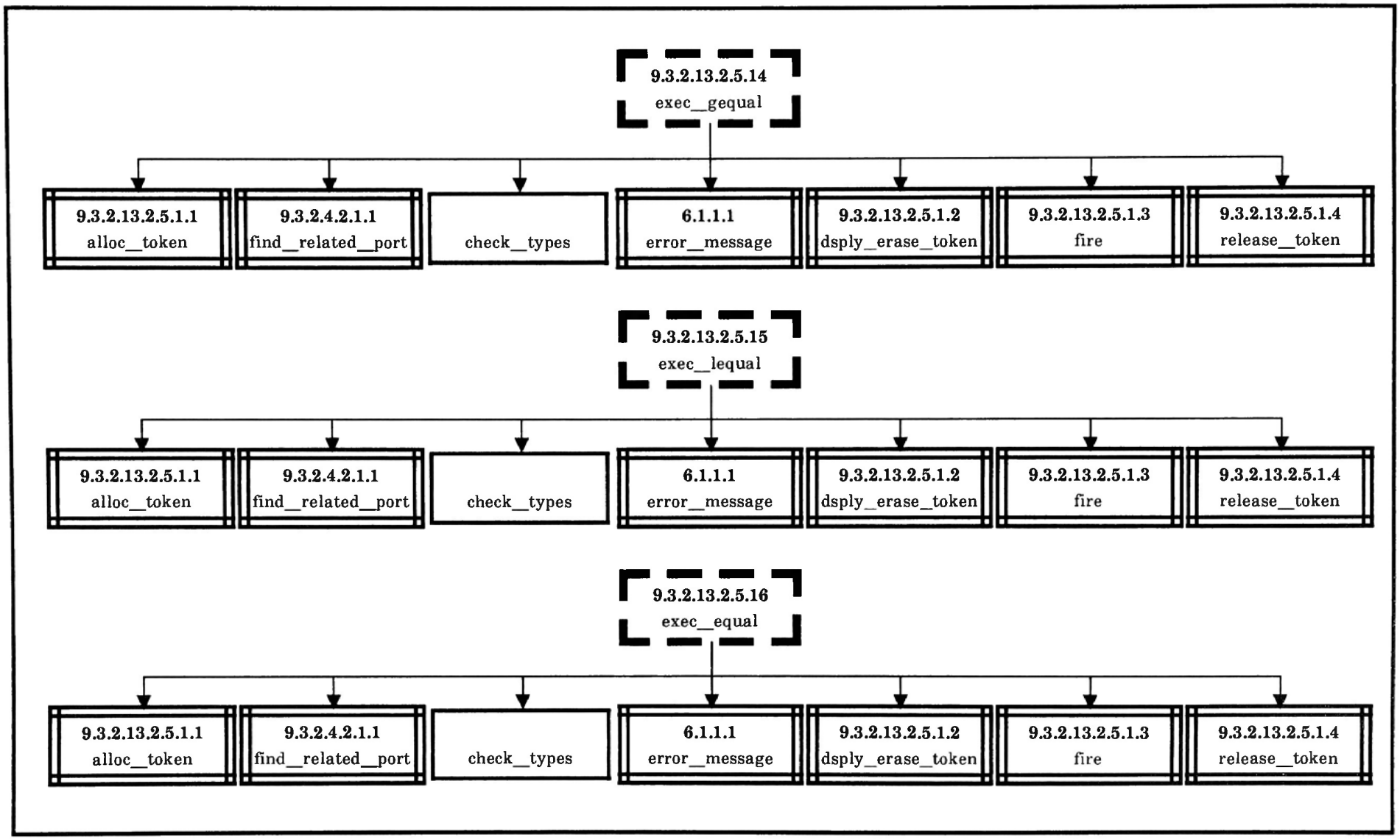

9.3.2.13.2.5.1
exec__add
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
check__types
6.1.1.1
error__message
9.3.2.13.2.5.1.2
dsply__erase__token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token
9.3.2.13.2.5.1.1
alloc__token
9.3.2.13.2.5.1.2
dsply_erase_token
9.3.2.13.2.5.1.2.1
dsply__location
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token
6.1.1.1
error__message
9.3.2.13.2.5.1.2.1
dsply__location
4
rfresh
7.8
warning
9.3.2.13.2.5.1.2
dsply_erase_token
9.3.2.4.2.1.1
find__related__port
9.3.2.4.2.1.1
find__related__port
9.3.2.13.2.5.2
exec__sub
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
check__types
6.1.1.1
error__message
9.3.2.13.2.5.1.2
dsply_erase_token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token

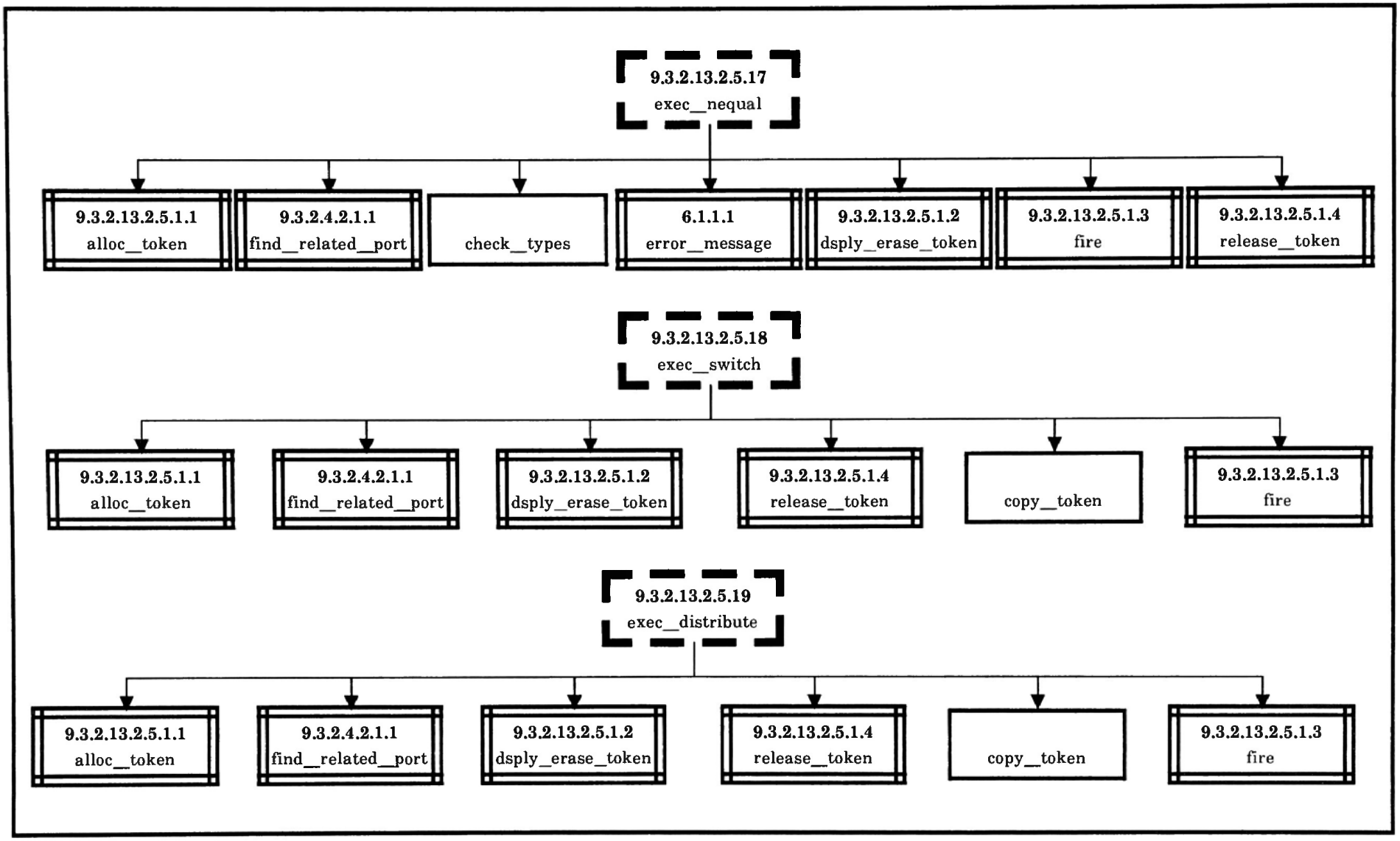

9.3.2.13.2.5.2
exec__sub
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
check__types
6.1.1.1
error__message
9.3.2.13.2.5.1.2
dsply__erase_token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token
9.3.2.13.2.5.3
exec__mul
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
check__types
6.1.1.1
error__message
9.3.2.13.2.5.1.2
dsply__erase_token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token
9.3.2.13.2.5.4
exec__div
9.3.2.13.2.5.1.1
alloc__token
9.3.2.13.2.5.1.1
alloc__token
check__types
6.1.1.1
error__message
9.3.2.13.2.5.4.1
divided__by__zero
9.3.2.13.2.5.1.2
dsply__erase_token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token
9.3.2.13.2.5.4.1
maps__set__up
6.1.1.1
error__message

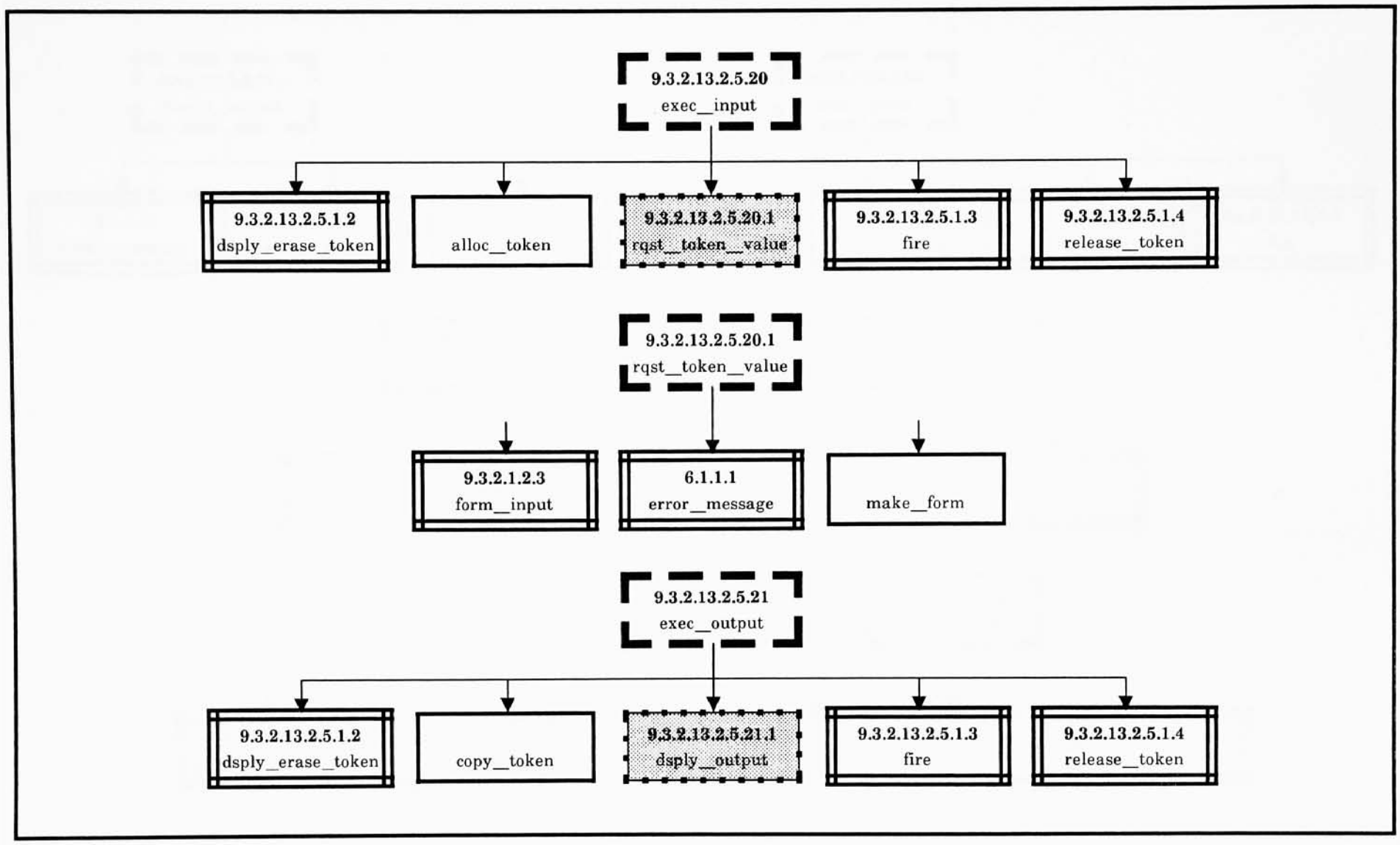

9.3.2.13.2.5.5
exec__mod
9.3.2.13.2.5.1.1
alloc__token
9.3.2.13.2.5.1.1
alloc__token
check__types
6.1.1.1
error__message
9.3.2.13.2.5.4.1
divided__by__zero
9.3.2.13.2.5.1.2
dsply__erase__token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token
9.3.2.13.2.5.6
exec__abs
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
check__types
6.1.1.1
error__message
9.3.2.13.2.5.1.2
dsply__erase__token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token
9.3.2.13.2.5.7
exec__sqrt
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
check__types
6.1.1.1
error__message
9.3.2.13.2.5.1.2
dsply__erase__token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token

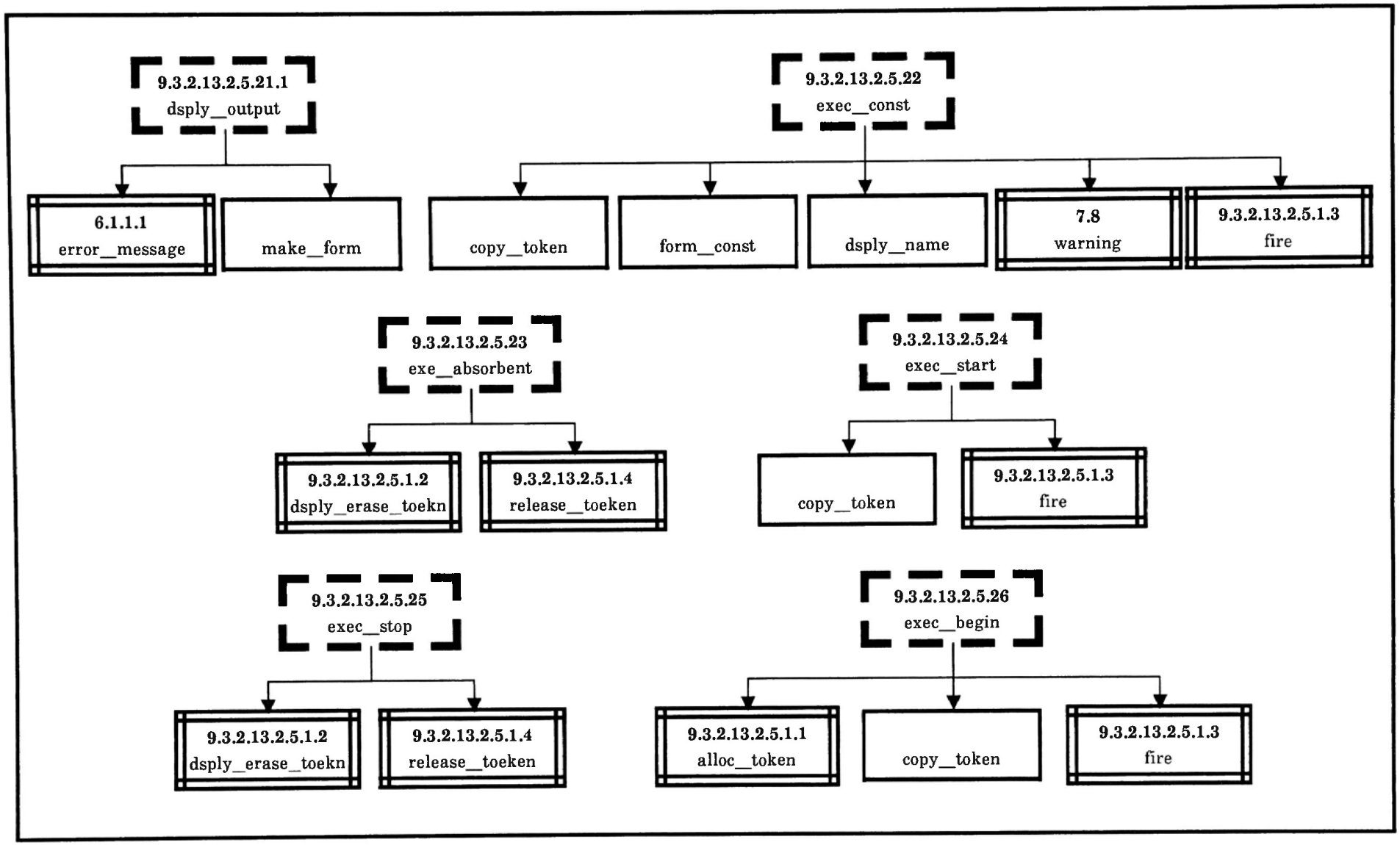

9.3.2.13.2.5.8
exec__neg
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
check__types
6.1.1.1
error__message
9.3.2.13.2.5.1.2
dsply_erase_token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token
9.3.2.13.2.5.9
exec__and
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
check__types
6.1.1.1
error__message
9.3.2.13.2.5.1.2
dsply_erase_token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token
9.3.2.13.2.5.10
exec__or
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
check__types
6.1.1.1
error__message
9.3.2.13.2.5.1.2
dsply_erase_token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token

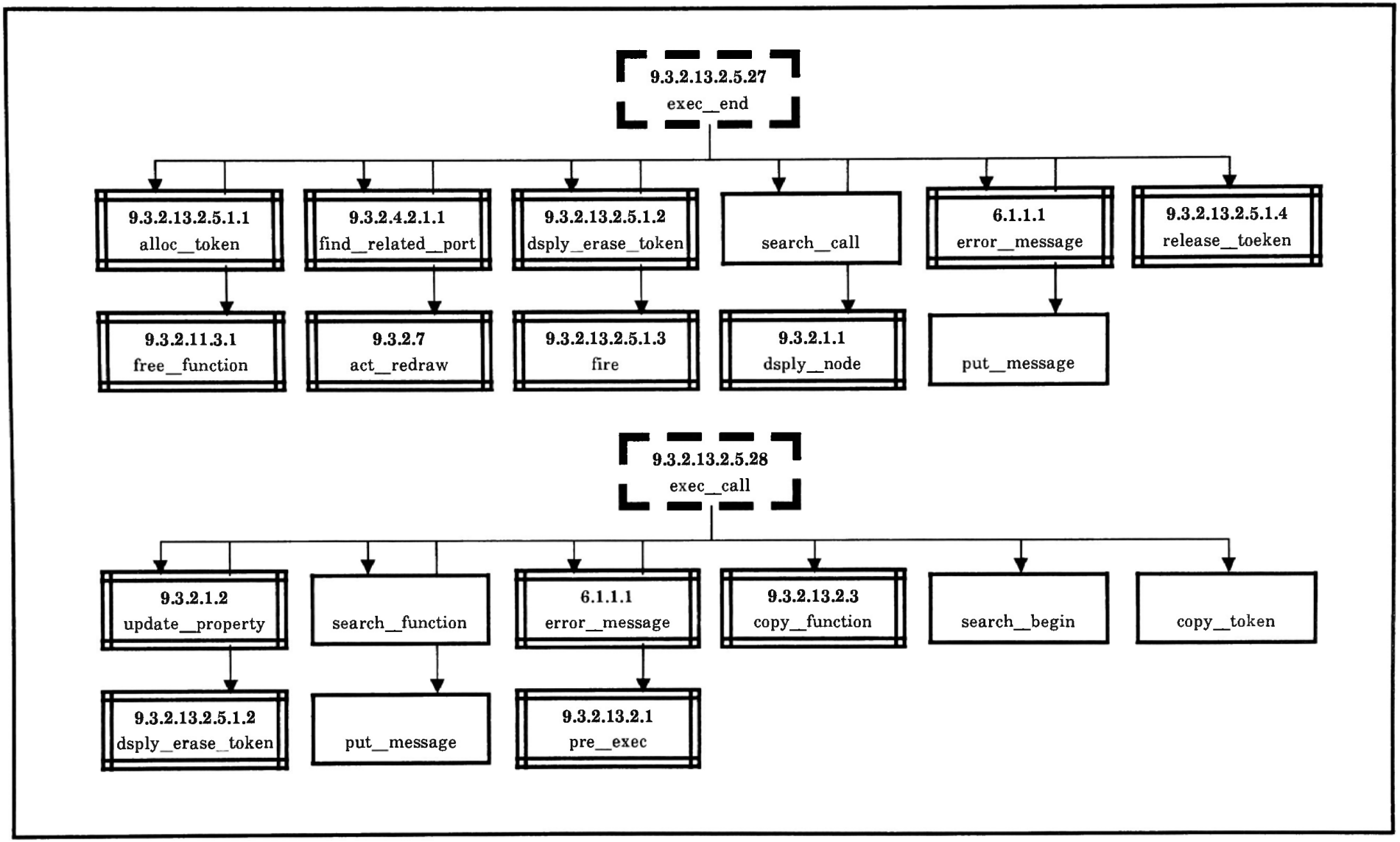

9.3.2.13.2.5.11
exec__not
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
check__types
6.1.1.1
error__message
9.3.2.13.2.5.1.2
dsply_erase_token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token
9.3.2.13.2.5.12
exec__greater
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
check__types
6.1.1.1
error__message
9.3.2.13.2.5.1.2
dsply_erase_token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token
9.3.2.13.2.5.13
exec__less
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
check__types
6.1.1.1
error__message
9.3.2.13.2.5.1.2
dsply_erase_token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token

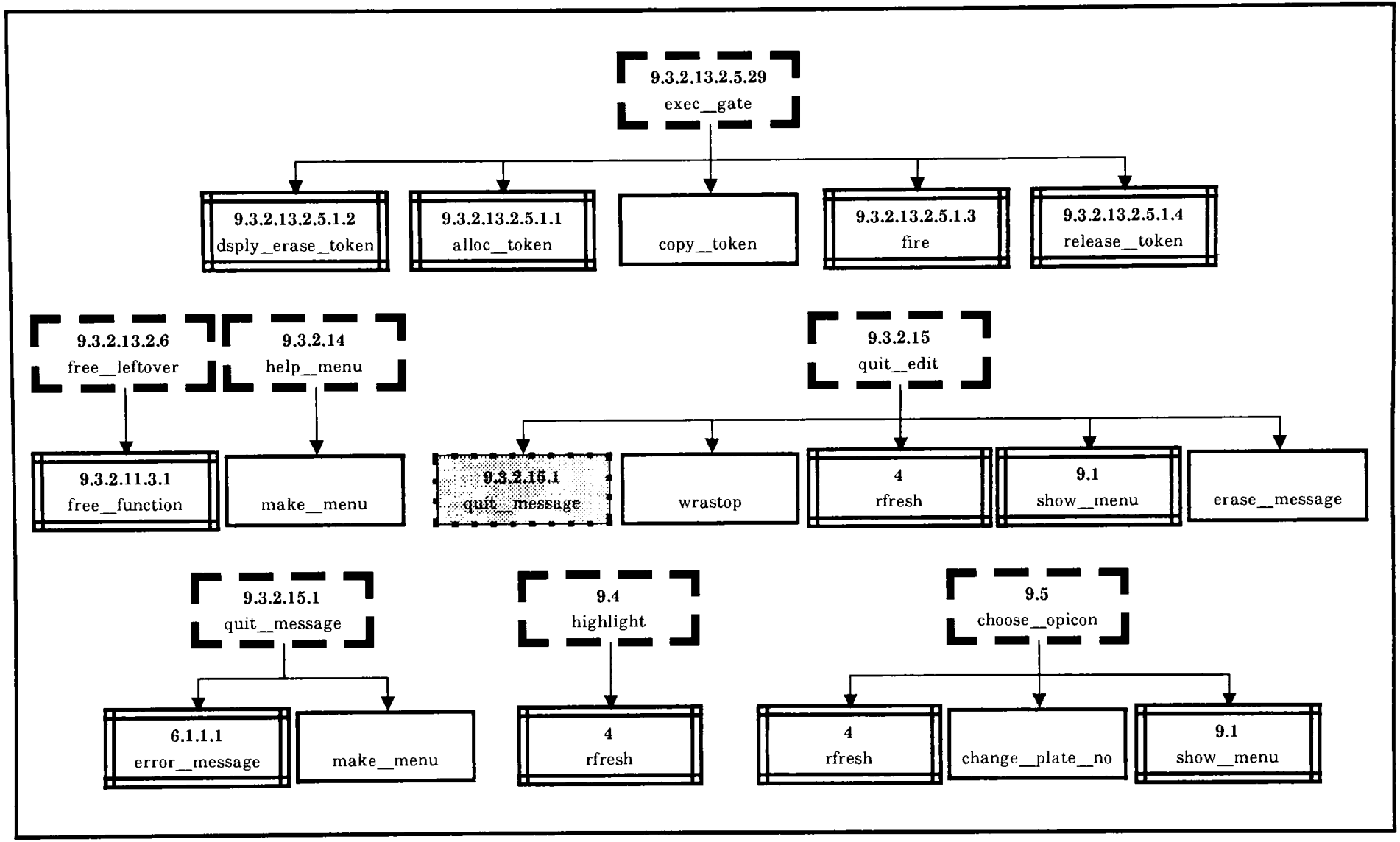

9.3.2.13.2.5.14
exec__gequal
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
check__types
6.1.1.1
error__message
9.3.2.13.2.5.1.2
dsply_erase_token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token
9.3.2.13.2.5.15
exec__lequal
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
check__types
6.1.1.1
error__message
9.3.2.13.2.5.1.2
dsply_erase_token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token
9.3.2.13.2.5.16
exec__equal
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
check__types
6.1.1.1
error__message
9.3.2.13.2.5.1.2
dsply_erase_token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token


9.3.2.13.2.5.17
exec__nequal
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
check__types
6.1.1.1
error__message
9.3.2.13.2.5.1.2
dsply_erase_token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token
9.3.2.13.2.5.18
exec__switch
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
9.3.2.13.2.5.1.2
dsply_erase_token
9.3.2.13.2.5.1.4
release__token
copy__token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.19
exec__distribute
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
9.3.2.13.2.5.1.2
dsply_erase_token
9.3.2.13.2.5.1.4
release__token
copy__token
9.3.2.13.2.5.1.3
fire


9.3.2.13.2.5.20
exec__input
9.3.2.13.2.5.1.2
dsply_erase_token
alloc__token
9.3.2.13.2.5.20.1
rqst__token__value
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token
9.3.2.13.2.5.20.1
rqst__token__value
9.3.2.1.2.3
form__input
6.1.1.1
error__message
make__form
9.3.2.13.2.5.21
exec__output
9.3.2.13.2.5.1.2
dsply_erase_token
copy__token
9.3.2.13.2.5.21.1
dsply__output
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token


9.3.2.13.2.5.21.1
dsply_output
6.1.1.1
error_message
make_form
9.3.2.13.2.5.22
exec_const
copy_token
form_const
dsply_name
7.8
warning
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.23
exe_absorbent
9.3.2.13.2.5.1.2
dsply_erase_toekn
9.3.2.13.2.5.1.4
release_toeken
9.3.2.13.2.5.24
exec_start
copy_token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.25
exec_stop
9.3.2.13.2.5.1.2
dsply_erase_toekn
9.3.2.13.2.5.1.4
release_toeken
9.3.2.13.2.5.26
exec_begin
9.3.2.13.2.5.1.1
alloc_token
copy_token
9.3.2.13.2.5.1.3
fire


9.3.2.13.2.5.27
exec__end
9.3.2.13.2.5.1.1
alloc__token
9.3.2.4.2.1.1
find__related__port
9.3.2.13.2.5.1.2
dsply__erase__token
search__call
6.1.1.1
error__message
9.3.2.13.2.5.1.4
release__toeken
9.3.2.11.3.1
free__function
9.3.2.7
act__redraw
9.3.2.13.2.5.1.3
fire
9.3.2.1.1
dsply__node
put__message
9.3.2.13.2.5.28
exec__call
9.3.2.1.2
update__property
search__function
6.1.1.1
error__message
9.3.2.13.2.3
copy__function
search__begin
copy__token
9.3.2.13.2.5.1.2
dsply__erase__token
put__message
9.3.2.13.2.1
pre__exec


9.3.2.13.2.5.29
exec__gate
9.3.2.13.2.5.1.2
dsply_erase_token
9.3.2.13.2.5.1.1
alloc__token
copy__token
9.3.2.13.2.5.1.3
fire
9.3.2.13.2.5.1.4
release__token
9.3.2.13.2.6
free__leftover
9.3.2.14
help__menu
9.3.2.15
quit__edit
9.3.2.11.3.1
free__function
make__menu
9.3.2.15.1
quit_message
wrastop
4
rfresh
9.1
show__menu
erase__message
9.3.2.15.1
quit__message
9.4
highlight
9.5
choose__opicon
6.1.1.1
error__message
make__menu
4
rfresh
4
rfresh
change__plate__no
9.1
show__menu

6.3 Gdf_*.h Files

6.3.1 Gdf_types.h File

This file contains the definitions of the data structures used in the system.

-- **Bitmap** structure specifies the width of the bitmap in number of unsigned short, the height of the bitmap in pixel, and a memory pointer which points to an area where it contains the actual pixel values.

-- **Funct_head** structure keeps the head of a linked list of a function graph, which contains information about a graph.

-- **Op_node** structure keeps the information of a node in the graph. The information includes the node number, operation code, number of input, number of output, a pointer points to an input port, a pointer points to an output port, coordinate of the first pixel, a pointer points to a property structure, and a pointer points to next node in the node linked list.

-- **Port** structure keeps the information about a port of a node. The information includes a token_exists flag, port number of itself, node number of the node it connects, port number of the port it connects, a pointer points to a token, and a pointer points to next port.

-- **Property** structure keeps the information for each of a CALL, INPUT, OUTPUT, or CONST node. It contains the type of a token which the node has or will have, a string for keeping the name of a function, or the value of an input, or the type of a constant, and another string for keeping prompt string.

-- **Conn_node** structure keeps the information of a node which is connected to some other node. The information includes the port number, node number, and a pointer points to the node.

-- **Node_highlight** structure keeps the information of a highlighted node. The information includes node number, map number (operation code), coordinate, and a selected flag.

-- **End_point** structure keeps the information of the end point of an arc. The information includes node number which the end point belongs to, map number, port number, coordinate of the map, and coordinate of the point.

-- **Line_segment** structure keeps the information of an arc. The information includes end points' coordinates, their node numbers, map numbers, map coordinates, and a pointer points to next segment.

-- **Seg_highlight** structure keeps the information of a selected arc. The information includes end points' coordinates, node numbers, map numbers, and a selected flag.

-- **Tokens** structure keeps the type and the value of a token.

-- **Funct_exec** structure keeps the execution stack information. It includes a pointer points to a function graph linked list, a pointer points to the head of a linked list of node_exec structure, a pointer points to the last executed node_exec structure, and a pointer points to the next execution stack element.

-- **Node_exec** structure keeps a pointer points to the currently executed node, and a pointer points to next node_exec structure.

6.3.2 Gdf_const.h File

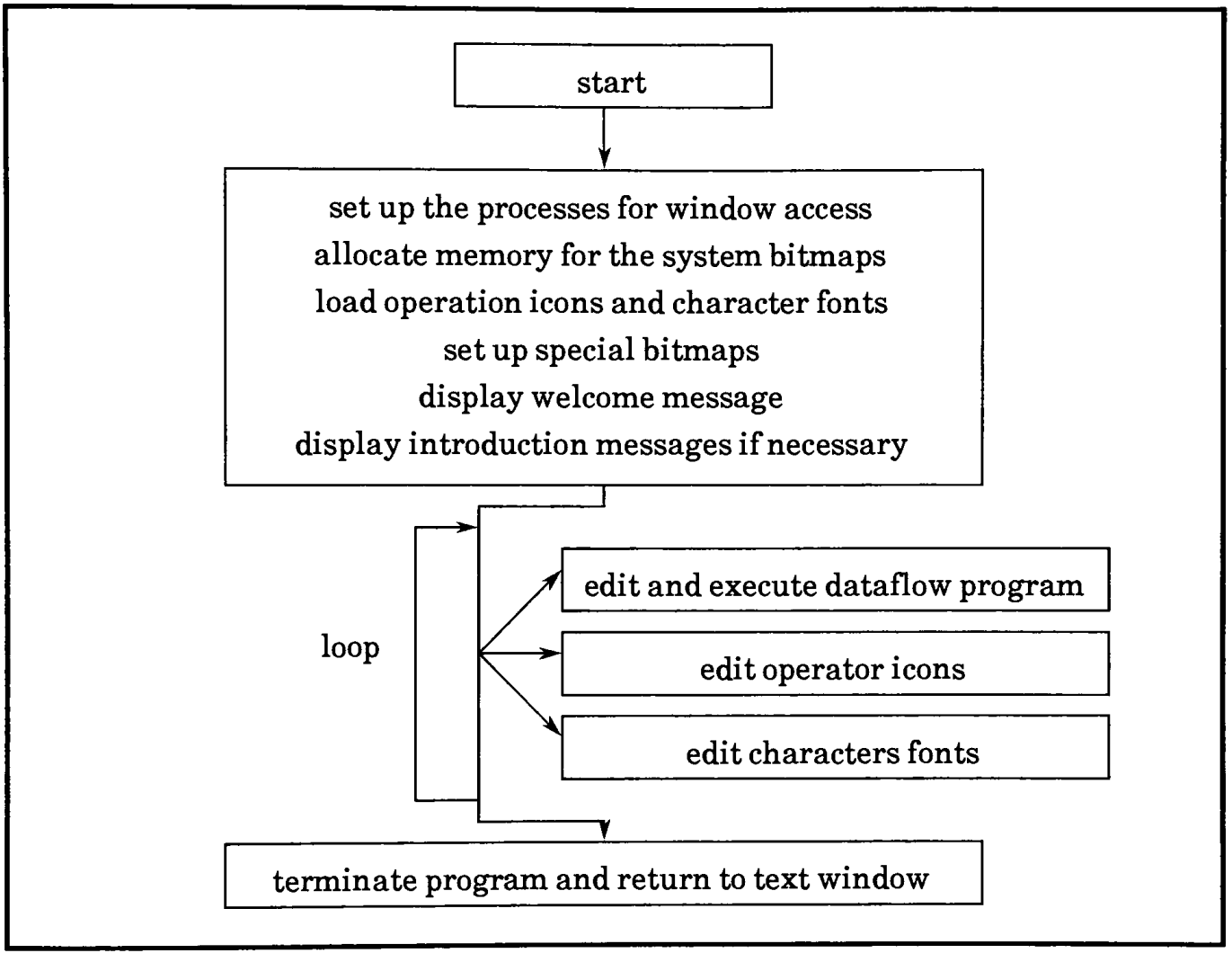
This file contains the definitions of constant names. The constant names include **the names of the operator icons bitmaps, character fonts bitmaps, editing commands, sixteen special bitmaps** which are used in display control, **window names, TRUE, FALSE, ZERO, WAIT, IN_PORT, OUT_PORT, MaxView,** and **MaxMaps.**

Most of the constants used in the GDF Programming System are defined in this file. Some constants names (e.g. constants used in act_execute) are defined in gdf_9.c file locally.

6.4 Main.c File

This file contains the main function of the system. The main function starts with setting up the processes for window access since all the display, editing, and simulation will be done in several different windows. Then the system allocates memory for system bitmaps which include windows' bitmaps, operator icons bitmaps, character fonts bitmaps, and several special bitmaps which helps in displaying options, command request box, and command

list.The system reads all of the information of operator icons and character fonts from external files, and places the pixel values in corresponding bitmaps. Some display screens are then setup successively. Finally, a menu is displayed to let user choose the options he wants. The diagram drawn below is the function diagram of the main function.



Main Function Diagram

6.5 Miscellaneous Files

The functions contained in gdf__1.c through gdf__11.c files are described in this section.

6.5.1 Gdf_1.c

This file contains 18 functions which perform graphics related operations.

-- **new_window:**
This function opens new windows for the system according to the specified border type, size, and location.

-- **select_window:**
This function is called to assign current window.

-- **new_map:**
This functions allocates memory to one new bitmap.

-- **rd_pixel:**
This function reads the pixel value from a specified pixel on the specified bitmap.

-- **wr_pixel:**
This function writes a zero or one to a specified pixel on a specified bitmap.

-- **toggle_pixel:**
This function toggles the pixel value.

-- **toggle_cell:**
This function toggles a cell which is a square and part of a bitmap.

-- **fill:**
This function fills the desired bitmaps with specified pixel value. The size and the location of the area which is to be filled must be specified.

-- **rfresh:**
This function is called to refresh a bitmap on the screen. The bitmap number is passed as a parameter.

-- **wrastop:**
This function is called by function "rfresh", and sets up a data structure, then calls the "ioctl" function to do the raster operation.

-- **clear:**
This function clears a bitmap by assigning zeroes to the memory locations which are used by the bitmap.

-- **enable_mouse:**
This function enables the mouse into operation. Um.um_icon is set to NULL at most of the time, but when the flag CARRYING or MOVING is TRUE, um.um_icon is allocated an area of memory and assigned a new icon to change the mouse cursor.

-- **read_locator:**
This function reports the mouse situation. The location of the mouse, the clicked button and the situation which is one of MouseDown, MouseUp, MouseIn, and MouseOut. This function always reads the first set information about the mouse location from the buffer stream.

-- **put_message**:
This function displays a message on the lower part of the screen. The line number and message have to be specified and passed as parameters. Constants for the line numbers are WTXTPROMPT, WTXTCMD, WTXTSLK1, and WTXTSLK2.

-- **erase_message**:
This function erases one line message from the lower part of the screen.

-- **terminate**:
This function terminates the program by closing window and calling the "ioctl" to return the window to standard display.

-- **alloc_window_maps**:
This function allocates memory to window bitmaps.

-- **alloc_icon_maps**:
This function allocates memory to icons and characters.

6.5.2 Gdf_2.c File

This file contains 14 functions which perform the operations needed in starting the program, editing icons and fonts.

-- **welcome_message**:
This function displays a welcome message on the screen.

-- **draw_frames**:
This function draws the frames for scratchpad and menu on the FRAME bitmap.

-- **make_frame**:
This function makes a frame for any specified bitmap.

-- **maps_setup**:
This function does some preparation work on several bitmaps which are YES_NO_MAP, SPOT, BLACK, FUNCT, WHITE PDMENU, CMD_MASK, FRAME, and LIGHT_FRAME.

-- **make_string**:
This function puts the imported string in the desired location on the specified bitmap.

-- **show_message**:
This function puts text messages on any specified bitmap. The size of the character can be enlarged, and the spacing between two characters is adjustable.

-- **set_char_map**:
This function returns a character bitmap number which corresponds to the actual parameter.

-- **locate_char**:
This function locates one character bitmap on a specified bitmap. The character bitmap can be enlarged to any size according to the assigned factor.

-- **select_char**:
This function allows the user to choose one character from the character option list when editing the character fonts.

-- **write_string**:
This function can write one character string on any selected bitmap then refreshes that bitmap onto the screen.

-- **locate_map**:
This function locates one bitmap onto another bitmap.

-- **select_op**:
This function allows the user to select one operator from the operator option list when editing the operator icons.

-- **confirm**:
This function displays a small box on the upper-right corner of the screen, and asks the user to answer yes or no.

-- **video_reverse**:
This function reverses partial pixel-values of a bitmap.

6.5.3 Gdf_3.c File

This file contains 10 functions which perform the operations needed in starting the system, and editing operator icons and character fonts.

-- **edit_op_icon**:
This function manipulates the editing of the operator icons.

-- **edit_char_fonts**:
This function manipulates the editing of the character fonts.

-- **set_grid**:
This function draws the grid for editing operator icons and for editing character fonts.

-- **show_op_options**:
This function displays a list of the names of the operator icons, which is used in the editing of the operator icons.

-- **show_char_options**:
This function displays a list of characters, which is used in the editing of character fonts.

-- **unload_map_info**:
This function reads the character font or operator icon bitmap information, and displays the desired character or operator icon on the grid.

-- **load_font:**
This function reads the character bitmap information from an external file, then stores the information into the memory location used by the bitmaps.

-- **save_font:**
This function saves the current character font bitmap information into an external file.

-- **load_op_icon:**
This function reads the operator icon bitmap information from an external file, then stores them in the memory locations used by the bitmaps.

-- **save_op_icon:**
This function saves the current bitmap information of the operator icons into an external file.

6.5.4 Gdf_4.c

This file contains several structure initialization and 20 functions which deal with the creation of menus and forms.

-- **main_selection:**
This function calls the built-in "menu" function to display the main menu of the GDF system.

-- **introduction:**
This function asks the user to see if the introduction of GDF system should be displayed or not.

-- **form_retrieve:**
This function calls the built-in function "form" to create a form which is used to ask user to give a file name form for retrieving graph program.

-- **make_form:**
This function is called to create a proper form to request some information. The name of the form is passed as a parameter.

-- **help_menu:**
This function reads help message from an external file and calls "menu" to display it.

-- **intro_menu:**
This function reads introduction text from an external file and calls "menu" to display it.

-- **make_menu:**
This function calls the built-in function "menu" to create proper menu to display messages. The name of the menu is passed as a parameter.

-- **ask_pgm_name:**
This function calls function "make_form" to create a form to ask user to give a name to the graph program.

-- **switch_pages:**
This function calls function "make_menu" to display a menu, allowing user to switch editing pages among several function program graphs.

-- **functable:**
This function calls "menu" to display a menu. The menu choices are ADD a new function, DELETE an existing function, and RENAME an existing function.

-- **end_new_session:**
This function is called when all the program graphs are deleted. It creates a menu to display the message.

-- **form_save:**
This function calls "form" to create a form when the user chooses Command-Save and wants to save the graph into a file. The form asks the user to give a file name.

-- **quit_message:**
This function calls the "make_menu" function to display messages when the user wants to quit the editing session.

-- **form_call:**
This function calls "make_menu" to create a form when the user copies the CALL operator icon from the menu panel, and locates it on the scratchpad. This function is also called when the user selects CALL in the graph and chooses the Command-Property.

-- **form_input:**
This function calls "make_menu" to create a form when the user copies the INPUT operator icon from the menu panel, and locates it on the scratchpad. This function also called when the user selects INPUT in the graph and choose the Command-Property.

-- **form_const:**
This function calls "make_menu" to create a form when the user copies the CONST operator icon from the menu panel, and locates it on the scratchpad. This function is also called when the user selects CONST in the graph and chooses the Command-Property.

-- **form_output:**
This function calls "make_form" to make a form for the user to create his own output token prompt string.

-- **rqst_input_value:**
This function prompts the user a form to enter a value for INPUT operator. The character string value is stored in property for further use, and the converted value is put in the token.

-- **dsply_output:**
This function calls "make_menu" to create a menu window to display the output result.

-- **error_message:**
This function makes a new menu window to display proper run-time error message.

6.5.5 Gdf_5.c File

This file contains 11 functions which perform the operations needed in editing a program graph.

-- **edit_graphs:**
This function manipulates the editing procedures of a program graph.

-- **in_command:**
This function returns TRUE if the mouse is located in the COMMAND REQUEST BOX, otherwise the function returns FALSE.

-- **set_cur_cmd:**
This function sets up a current commands array.

-- **command:**
This function pre-processes the chosen_command.

-- **highlight:**
This function highlights the current selected command and reverses the previous chosen command back to its original un-highlighted display.

-- **in_scratchpad:**
This function checks the current mouse location and returns TRUE if the mouse is in the scratchpad area, otherwise the function returns FALSE.

-- **in_menupanel:**
This function checks the current mouse location and returns TRUE if the mouse is in the menu panel area, otherwise the function returns FALSE.

-- **quit_edit:**
This function manipulates the activities when the user asks for quitting an editing.

-- **choose_opicon:**
This function returns the operator map number, which is chosen from the menu panel. The chosen operator is also highlighted.

-- **show_menu:**
This function refreshes the menu panel and the Command Request Box icon on the upper-left corner of the screen.

-- **change_plate_no:**
This function handles the variable PLATE_NO, this number keeps track of the group number of the current displayed operator icons. "Show_menu" function uses it.

6.5.6 Gdf_6.c File

This file contains 15 functions which handle the command actions.

-- **action:**
This function takes proper actions after the user chooses certain operators and commands.

-- **act_copy_locate:**
This function handles the operations to copy and locate an operator icon. If the mouse location is within the scratchpad area, a new node memory is allocated. Proper information is assigned to the new node, then the new node is added to the linked list. Screen is also refreshed.

-- **act_select:**
This function handles the operations for selecting an operator icon or selecting a line segment from the scratchpad area.

-- **act_move:**
This function handles the operations for moving an icon in the scratchpad area. A flag is set to TRUE in order to invoke "enable_mouse" to change the mouse icon.

-- **act_delete:**
This function handles the operations for deleting an operator icon or line segment from the scratchpad area.

-- **act_connect:**
This function handles the operations for connecting two nodes. These selected nodes are highlighted.

-- **stop_connect:**
This function is called when a connection is not completed.

-- **act_property:**
This function handles the operations for changing or assigning node property for CALL, INPUT, OUTPUT, and CONST.

-- **act_redraw:**
This function handles the operations for redrawing the screen according to the current graph information.

-- **act_new:**
This function handles the operations for creating a new program. A program can be composed of several function graphs.

-- **act_functions**:
This function handles the operations needed when a user wants to add a new function graph, delete an existing function, or rename an existing function graph.

-- **act_switch_pages**:
This function handles the operations needed when a user wants to switch the current editing screen to some other function graph.

-- **act_save**:
This function handles the operations needed when a user wants to save the program graph information.

-- **act_retrieve**:

This function handles the operations needed when a user wants to retrieve a program graph from an external file.

-- **act_execute**:
This function handles the operations needed when a user wants to execute a dataflow program which currently resides in the system.

6.5.7 Gdf_7.c File

This file contains 24 functions which perform actions related to the editing of a graph.

-- **add_node**:
This function adds a new node to the operator node linked list.

-- **delete_node**:
This function deletes a node from the node linked list. If the node to be deleted is the first node of the list, along with deleting the node, it also changes the current node head which is a pointer, that will always point to the first node of the linked list. When freeing the node memory, the program also frees other pointer variables, such as the input arc, the output arc, and the property.

-- **search_node**:
This function searches the node linked list by the x, y coordinates of the mouse location. If the mouse is located inside of a node bitmap, this function returns a pointer points to the node, otherwise, it returns NULL.

-- **cut_connections**:
This function handles the operations for deleting all the edges which are connected to a node which is to be deleted.

-- **move_connections**:
This function redraws all the connections which are connected to the node which was moved in the previous action.

-- **add_line_segment**:
This function adds a new line segment to the segment linked list. The new line segment is created to connect two nodes in the previous action.

-- **delete_line_segment:**
This function deletes the selected line segment from the linked list, and erase the line (and/or arrow) from the screen.

-- **search_line_segment:**
The function checks the distance from a specified point to every line segment in the line segment linked list. If the distance is short enough (means the point is near enough to the line segment), a pointer to that line segment is returned. If no distance is near enough, a NULL is returned.

-- **closer:**
This function calculates the distance from a point to a line segment. If the distance is smaller than 9, it returns TRUE, otherwise it returns FALSE.

-- **rdraw:**

This function redraws all the graph information on the screen. All the nodes are refreshed, segments and arrows are redrawn.

-- **line_exist:**
This function checks the duplications for the line segment. If any duplication occurs, TRUE is returned, otherwise FALSE is returned.

-- **warning:**
This function prints out a warning message on the line WTXTPROMPT, delays 2 seconds, then erases it.

-- **setup_node:**
This function sets up memory locations for a new node before it is added to the node linked list. The allocations include the input and output connection ports.

-- **setup_connections:**
This function updates the connection information in every connected port. The information includes the connected node number and the connected port number.

-- **get_port:**
This function returns a port pointer. The port is found based on the node number, input or output port, the port number, and the op_code of the node.

-- **free_program:**
This function handles the operations to free the memory occupied by a program graph.

-- **free_function:**
This function frees the memory occupied by a function graph linked list.

-- **free_op_node:**
This function handles the operations to free the memory occupied by one node.

-- **free_segments:**
This function handles the operations to free the memory occupied by the line segments linked list.

-- **free_arc:**
This function handles the operations to free the memory occupied by ports.

-- **count_function:**
This function returns a counter which is the total number of functions contained in an entire program.

-- **count_node:**
This function returns a counter which is the total number of op_node contained in a graph function.

-- **count_segment:**
This function returns a counter which is the total number of line segments in a graph function.

-- **has_property:**
This function checks the op_code of a node, if the node is one of CALL, INPUT, OUTPUT, or CONST, then it returns TRUE, otherwise it returns FALSE.

6.5.8 Gdf_8.c File

This file contains 14 functions which perform operations related to the editing of a graph.

-- **num_input:**
This function returns the total number of input ports of an operator.

-- **num_output:**
This function returns the total number of output ports of an operator.

-- **connect_point:**
This function reports the exact location for the connecting arc.

-- **draw_line:**
This function calculates the pixels coordinates which lays on a line between the beginning point and the ending point.

-- **draw_arrow:**
This function calculates two points' coordinates for drawing an arrow.

-- **savefile:**
This function saves the entire program graphs into an external file for further use.

-- **getfile:**
This function reads an entire program graph from an external file, and sets the current function and the current node linked list head, and the current line segment linked list head.

-- **in_out_available:**
This function returns a boolean value to indicate the availability of the port of a node.

-- **clean_ports:**
This function clears the information contained in the port structure list.

-- **find_related_port:**
This function returns a port pointer based on the node number, port number, and an input/output flag.

-- **clear_segment:**
This function clears the connection information of two nodes when one arc is deleted.

-- **update_property:**
This function handles the operations to update the property for either of one CALL, INPUT, OUTPUT, or CONST.

-- **del_from_gate:**
This function returns a node pointer which points to GATE node whose input port or output port arc is going to be deleted, or returns NULL if the port does not belong to a GATE.

-- **clr_gate_port:**
This function finds one arc of a GATE node and resets the linking pointer, and frees the deleted port memory.

6.5.9 Gdf_9.c File

This file contains 30 functions which perform the operations related to the execution of a graph program.

-- **dsply_node:**
This function displays an operation node by refreshing the node's bitmap on the screen. A flag is needed to determine the way for refreshing the node, which is either DSTXORed, DSTORed, or highlighted.

-- **dsply_name:**
This function displays the character string contained in the property field of a node. Only three kinds of nodes contain properties. They are CONST, CALL, and INPUT.

-- **erase_node:**
This function erases a node from the screen by using DSTXOR operation to refresh the bitmap, also if the node is CALL, CONST, or INPUT, erases its property.

-- **erase_name:**
This function erases the property character string of CONST, CALL, and INPUT from the screen by DSTXOR, the bitmap of each character in the string.

-- **to_uppercase:**
This function converts every character in a string from lower case to uppercase. This is because that the GDF system only provides the manipulation on capital letters.

-- **valid_str:**
This function checks the characters of a string, and returns TRUE if all the characters are valid in the system.

-- **letter:**
This function returns a TRUE if the character is one of the valid letters.

-- **digits:**
This function returns a TRUE if the character is a decimal point or a digit.

-- **check_string:**
This function checks the validity of a string.

-- **check_delay:**
This function plays a short period of delay whenever it needs to prolong the highlight period.

-- **speeding:**
This function keeps track of the SPEED_LEVEL value. When the user asks to speed the simulation speed up or down, the function will change the SPEED_LEVEL and cause the CHECK_DELAY function to perform a different delay duration.

-- **check_connection:**
This function handles the operations for checking the arc connections of the graphs of a program.

-- **check_nconnection:**
This function checks the connections of a node, to make sure all the input and output ports are connected before executing the program.

-- **exec_program:**
This function handles the operation for executing a program graph.

-- **pre_exec:**
This function pre-executes the graph information by setting the nodes which have no input data to be ready for executing.

-- **exec_node:**
This function calls proper function to execute the operation of a node.

-- **enqueue_fireable_nodes:**
This function puts all the fireable nodes into the queue to wait to be executed. The bitmaps of all the nodes in the queue are highlighted.

-- **fetch_node:**
This function fetches the head node of the queue and returns NULL if the queue is empty.

-- **fireable:**
This function returns a TRUE if the node's inputs are all ready, and the output token has been consumed, otherwise returns FALSE.

-- **ready_input:**
This function checks the input arcs of a node, returns TRUE if the tokens are all ready, otherwise returns FALSE.

-- **ready_output:**
This function checks the output arcs of a node, returns TRUE if the previous output token has been consumed, otherwise returns FALSE.

-- **enq_node:**
This function adds a node to the end of the queue to wait to be executed.

-- **de_fireable:**
This function sets the fireable nodes to un-fireable.

-- **conn_to_stop:**
This function returns TRUE if the GATE node is connected to either STOP or END node, otherwise returns FALSE.

-- **free_leftover:**
This function frees the memory occupied by these function linked list which have not been executed completely because of a run-time error that occurs.

-- **queued:**
This function checks the argument from the waiting queue, if the_node is in the queue, the function returns TRUE, otherwise returns FALSE.

-- **state_changed:**
This function checks the command state changes, and returns the proper indicator in curr_state of calling function.

-- **current_command_state:**
This function enables the mouse and calls READ_CONTROL to determine the current command state when a graph execution is progressing, and returns the proper indicator to the calling function.

-- **read_control:**
This function reads the mouse location from the input stream. Because the most recent data is required, the input stream must be manipulate in a way to get the desired data without any confusion. This function is similar to READ_LOCATOR but with more complicated analysis on the input stream data.

-- **alloc_token:**
This function allocates memory to a new token structure and returns the pointer to its calling function.

6.5.10 Gdf_10.c File

This file contains 17 functions which perform the operations related to the execution of a graph program.

-- **exec_add:**
This function executes an "add" operation. It returns TRUE (if there is an error!) if types are conflicting, otherwise returns FALSE (no error!).

-- **exec_sub:**
This function executes a "subtract" operation. It subtracts the right token from the left token, and puts the result in the output token. It also returns FALSE if an error occurs.

-- **exec_mul:**
This function executes a "multiply" operation, returns TRUE if error occurs.

-- **exec_div:**
This function executes a "divide" operation.

-- **exec_mod:**
This function executes a "modulo" operation. Only two integers tokens are allowed to invoke this operation.

-- **exec_abs:**
This function executes an "absolute" operation. It produces a token which contains a positive token_value.

-- **exec_sqrt:**
This function finds the square root of the input token value, and puts the result in the output token.

-- **exec_neg:**
This function finds the negative of the input token value and puts it in the output token.

-- **exec_and:**
This function executes a logical AND operation.

-- **exec_or:**
This function executes a logical OR operation.

-- **exec_not:**
This function executes a logical NOT operation.

-- **exec_greater:**
This function checks the magnitudes of two input token values. If the left token value is greater than the right input token value, this function generates an output token with token value TRUE, otherwise with FALSE.

-- **exec_less:**
This function checks the magnitudes of two input token values. If the left input token value is less than the right input token value, this function generates an output token with token value TRUE, otherwise with FALSE.

-- **exec_gequal:**
This function checks the magnitudes of two input token values to see the condition "> =" is TRUE or FALSE.

-- **exec_lequal:**
This function checks the magnitudes of two input token values to see the condition "<=" is TRUE or FALSE.

-- **exec_equal:**
This function checks the magnitudes of two input token values to see the condition "=" is TRUE or FALSE.

-- **exec_nequal:**
This function checks the magnitudes of two input token values to see the condition "!=" is TRUE or FALSE.

6.5.11 Gdf_11.c File

This file contains 26 functions which perform the operations related to the execution of a graph program.

-- **exec_switch:**
This function executes a "switch" operation. One of the two input tokens of the SWITCH operator that is put onto the output port depends on the boolean input.

-- **exec_distribute:**
This function executes a "distribute" operation. The input token is put onto one of the two output ports, and depend on the boolean input.

-- **exec_input:**
This function executes an "input" operation, requests the user to input an integer number, or character, or boolean value, or real number.

-- **exec_output:**
This function executes an "output" operation, and opens a window to display the value passed on the input token of a OUTPUT operator.

-- **exec_const:**
This function executes a "constant" operation, and generates a constant as specified.

-- **exe_absorbent:**
This function executes an "absorb" operation, it takes input tokens, and releases and frees their memory.

-- **exec_start:**
This function executes the "start" operation, and generates a dummy token and fires it.

-- **exec_stop:**
This function executes a "stop" operation, and terminates a program.

-- **exec_begin:**
This function executes a "begin" operation. Proper parameters are passed from the calling function.

-- **exec_end:**
The functions executes an "end" operation. A sub-function is terminated and the top of function_stack is popped.

-- **exec_call:**
This function executes a function call operation.

-- **exec_gate:**
This function executes a "gate" operation. The input token is passed to each of the output ports.

-- **fire:**
This function performs the action for firing a token. It finds out where the token is going to be (finds the input port of another node), sets token_exists flag, and makes the pointer field point to the specific token.

-- **release_token:**
This function releases the memory used by the input token, and is called after an operator node fires the output token.

-- **copy_function:**
This function duplicates the whole data structure of an existing graph of a function when function calls are employed.

-- **copy_token:**
This function copies the contents of one token to another token, and duplicates the token.

-- **check_type:**
This function checks the types of two tokens and returns proper indicator.

-- **divide_by_zero:**
This function returns TRUE if the divisor is zero.

-- **search_function:**
This function returns a function head pointer if the specified function does exist, otherwise returns NULL.

-- **search_begin:**
This function searches an op_node linked list of a function graph, and returns a pointer which points to BEGIN node if it finds it, otherwise returns NULL.

-- **copy_nodes:**
This function copies the nodes information from an original function graph to a duplicated function.

-- **copy_segments:**
This function copies the line segments information from an original function graph to a duplicated function.

-- **copy_arcs:**
This function copies the arc information contained in an original node to a new duplicated node.

-- **search_call:**
This function searches the CALL node from the last calling function linked list.

-- **dsply_erase_token_value:**
This function displays the token value on the node's input port and output port by the DSTXOR operation. If the value is displayed twice then it will disappear as if it had been erased.

-- **dsply_location:**
This function finds out the exact location for displaying the token value. The location is determined at the one quarter to the desired end on the arc.

6.6 File Contents Summary and Statistics

The System programs are composed of 199 functions. These functions are grouped in 12 files. The files contents are summarized in this section.

Main.c
main ( )

Gdf_1.c

new_window ( )	select_window ( )	new_map ( )
rd_pixel ( )	wr_pixel ( )	toggle_pixel ( )
toggle_cell ( )	fill ( )	rfresh ( )
wrastop ( )	clear ( )	enable_mouse ( )
read_locator ( )	put_message ( )	erase_message ( )
terminate ( )	alloc_window_maps ( )	alloc_icon_maps ( )

Gdf_2.c

welcome_message ( )	draw_frames ( )	make_frame ( )
maps_set_up ( )	make_string ( )	show_message ( )
get_char_map ( )	locate_char ( )	select_char ( )
write_string ( )	locate_map ( )	select_op ( )
confirm ( )	video_reverse ( )	

Gdf_3.c

edit_op_icons ( )	edit_char_fonts ( )	set_grid ( )
show_op_options ( )	show_char_options ( )	unload_map_info ( )
load_font ( )	save_font ( )	load_op_icon ( )
save_op_icon ( )		

Gdf_4.c

main_selection ( )	introduction ( )	form_retrieve ( )
make_form ( )	help_menu ( )	intro_menu ( )
make_menu ( )	ask_pgm_name ( )	switch_pages ( )
functable ( )	end_new_session ( )	form_save ( )
quit_message ( )	form_call ( )	form_input ( )
form_const ( )	form_output ( )	rqst_input_value ( )
dsply_output ( )	error_message ( )	

Gdf_5.c

edit_graph ( )
command ( )
in_menupanel ( )
show_menu ( )
in_command ( )
highlight ( )
quit_edit ( )
change_plate_no ( )
set_cur_cmd ( )
in_scratchpad ( )
choose_opicon ( )

Gdf_6.c

action ( )
act_move ( )
stop_connect ( )
act_new ( )
act_save ( )
act_copy_locate ( )
act_delete ( )
act_property ( )
act_functions ( )
act_retrieve ( )
act_select ( )
act_connect ( )
act_redraw ( )
act_switch_pages ( )
act_execute ( )

Gdf_7.c

add_node ( )
cut_connections ( )
delete_line_segment ( )
rdraw ( )
setup_node ( )
free_program ( )
free_segments ( )
count_node ( )
delete_node ( )
move_connections ( )
search_line_segment ( )
line_exist ( )
setup_connection ( )
free_function ( )
free_arc ( )
count_segment ( )
search_node ( )
add_line_segment ( )
closer ( )
warning ( )
get_port ( )
free_op_node ( )
count_function ( )
has_property ( )

Gdf_8.c

num_input ( )
drawline ( )
getfile ( )
find_related_port ( )
del_from_gate ( )
num_output ( )
draw_arrow ( )
in_out_available
clear_segment ( )
clr_gate_port ( )
connect_point ( )
savefile ( )
clear_ports ( )
update_property ( )

Gdf_9.c

dsply_node ( )
erase_name ( )
letter ( )
check_delay ( )
check_nconnection ( )
exec_node ( )
fireable ( )
enq_node ( )
free_leftover ( )
read_control ( )
dsply_name ( )
to_uppercase ( )
digits ( )
speeding ( )
exec_program ( )
enqueue_fireable_node ( )
ready_input ( )
de_fireable ( )
queued ( )
current_command_state ( )
erase_node ( )
valid_str ( )
check_string ( )
check_connections ( )
pre_exec ( )
fetch_node ( )
ready_output ( )
conn_to_stop ( )
state_change ( )
alloc_token ( )

Gdf_10.c

exec_add ( )
exec_div ( )
exec_sqrt ( )
exec_or ( )
exec_less ( )
exec_equal ( )
exec_sub ( )
exec_mod ( )
exec_neg ( )
exec_not ( )
exec_gequal ( )
exec_nequal ( )
exec_mul ( )
exec_abs ( )
exec_and ( )
exec_greater ( )
exec_lequal ( )

Gdf_11.c

exec_switch ( )	exec_distribute ( )	exec_input ( )
exec_output ( )	exec_const ( )	exe_absorbent ( )
exec_start ( )	exec_stop ( )	exec_begin ( )
exec_end ( )	exec_call ( )	exec_gate ( )
fire ( )	release_token ( )	copy_function ( )
copy_token ( )	check_types ( )	divide_by_zero ( )
search_function ( )	search_begin ( )	copy_nodes ( )
copy_segments ( )	copy_arcs ( )	search_call ( )
dsply_erase_token_value ( )		dsply_location ( )

The data listed below is a statistic report about the size of the system programs.

File name	# of Lines	# of Words	# of Characters
gdf_types.h	175	942	6566
gdf_const.h	176	892	6703
main.c	77	353	3205
gdf_1.c	505	1804	16394
gdf_2.c	483	1936	15679
gdf_3.c	391	1842	15664
gdf_4.c	1052	3514	31536
gdf_5.c	505	2025	17118
gdf_6.c	875	2914	29665
gdf_7.c	847	2914	28658
gdf_8.c	930	3240	28886
gdf_9.c	1081	3568	33117
gdf_10.c	911	2802	33915
gdf_11.c	982	3286	33671
gdf_help	345	1929	12732
gdf_intro	128	733	4979
Total:			
16 Files, 199 Functions	9463	34700	318488

7. Conclusions

A dataflow computer is a special processor for performing highly concurrent applications. A well-designed language and a user friendly interface for a dataflow computer can evidently bring the computer performance into full play. The development of this thesis is based on this essential significance.

This thesis also advocates that interactive graphics can be used to support concurrent programming effectively. The main contributions of GDF programming system are summarized as:

1. a novel way of representing dataflow program graphically,
2. a convenient editing facilities which guarantee syntactically correct dataflow graphs,
3. a vivid animation of the data flowing which helps in debugging, and
4. an easy detection of deadlock during execution of dataflow program

Shortcomings of the system:

A dataflow graph can contain any number of nodes, but the size of the scratchpad area is limited. Hence the size of a dataflow graph is restricted. The operators supported by the GDL do not include LOOP-operators (for-loop, while-loop, or repeat-loop) and structure of array types.

Suggestions for future investigation and improvements:

1. Get rid of the limitation of the size of the graph. make the scratchpad area extensible. This problem involves the manipulations of the bitmaps of the screen and the structure of the objects scrolling on the screen.
2. Eliminate the extra work in firing useless tokens. Imposing some kind of backtracking principle on the firing rule might may solve this problem.
3. Expand the instruction set of GDL. The current implementation only support basic operators. Expand the instruction set to include Loop operators, arrays, structures, and lists. The designer must consider the internal representations of the data for each of these operation.

4. Allow users to include existing program graphs. This would create modularity and reduce overhead of repeatedly editing the same graph.
5. Generate other high level programming language codes. Interactive graphics can be used to design graphical front end for existing textual dataflow languages.

Bibliography

[Ackerman 1982]

William B. Ackerman, "*Data Flow Languages*," Computer, Vol. 15, No. 2, February, pp. 15-25.

[Ackerman, Dennis 1979]

William B. Ackerman and Jack B. Dennis, "*VAL -- A valuel oriented Algorithmic Language: Preliminary Reference manual*," Computation Structure Group, Laboratory for Computer Science, MIT, Cambridge, MA, June 1979.

[Adams 1968]

D. A. Adams, "*A Computation Model With Data Flow Sequencing*," Technical Report CS117, School of Humanities and Science, Stanford University, Stanford, California, December 1968.

[Agerwala 1982]

Tilak Agerwala, "*Data Flow System*," Computer, Vol. 15, No. 2, February 1982, pp. 10-13.

[Agerwala, Arvind 1982]

Tilak Agerwala and Arvind, "*Data Flow Systems*," Computer, Vol. 15, No. 2, February 1982, pp. 10-13.

[Arvind, Gostelow 1977]

Arvind and K. P. Gostelow, "*A Compuater Capable of Exchanging Processors for Time*," Proceedings IFIP Congress (1977), August 1977, pp. 849-854.

[Arvind, Gostelow 1982]

Arvind and Kim P. Gostelow, "*The U-Interpreter*," Computer, Vol. 15, No. 2, February, pp. 42-49.

[Arvind, Kathail 1981]

Arvind and V. Kathail, "*A Multiple Processor Data Flow Machine that Supports Generalized Procedures*," Eighth Annual Symposium on Computer Architecture, Minneapolis, Mn., 12-14 May 1981 (New York: IEEE 1981), pp. 291-302.

[Arvind, Gostelow, Plouffe 1978]

Arvind, K. P. Gostelow, and W. Plouffe, "*The (Preliminary) Id report*," Department of Information and Computer Science (TR114), University of California, Irvine, California, May 1978.

[Bo 1982]

Ketil Bo, "*Human-Computer Interaction*," Computer, Vol. 15, No. 11, November 1982, pp. 9-11.

[Boarder 1981]

J. C. Boarder, "*Graphical Programming for Parallel Processing Systems*," International Conference on Distributed Computing System, July 1981, pp. 467-475.

[Borning 1981]

A. Borning, "*The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory*," ACM Trans. Programming Languages and Systems, Vol. 3, No. 4, October 1981, pp. 353-387.

[Cattaneo, Guercio, Levialdi, Tortora 1986]

G. Cattaneo, A. Guercio, S. Levialdi, and G. Tortora, "*ICONLISP: An example of a Visual Programming Language*," IEEE Computer Society Workshop on Visual Language, February 1986, pp. 22-25.

[Christenson, Wolfberg, Fisher 1971]

C. Christenson, M. S. Wolfberg, and M. J. Fisher, "*AMBIT/G Final Report -- Task Area I*," tech report AD-720-313, Nat'l Tech. Information Service, Springfield, Va., 1971.

[Comte, Hifdi, Syre 1980]

D. Comte, N. Hifdi, and J. C. Syre, "*The Data Driven LAU Mutiprocessor System: Results and Perspectives*," Proceedings IFIP Congress (1980), October 1980, pp. 175-180.

[Coutaz 1985]

Joelle Coutaz, "*Abstractions for User Interface Design*," Computer, Vol. 18, No. 9, September 1985, pp. 21-34.

[Curry 1978]

G. A. Curry, "*Programming by Abstract Demonstration*," Ph.D. dissertation, Dept. of Computer Science, University of Washington, Technical Report 78-03-02, Seattle, Washington, 1978.

[Davis 1978]

A. L. Davis, "*The Architecture and System Methodology of DDM1: A Recursively Structured Data Driven Machine*," Proceedings Fifth Annual Symposium Computer Architecture, (Palo Alto, California, April 3-5) ACM, New York, 1978, pp. 210-215.

[Davis 1979]

A. L. Davis, "*A Data Flow Evaluation System Based on the Concept of Recursive Locality*," Proceedings 1979 National Computer Conference 1979, pp. 1079-1086.

[Davis, Keller 1982]

Alan L. Davis and Robert M. Keller, "*Data Flow Program Graph*," Computer, Vol. 15, No. 2, February, pp. 26-41.

[Denert, Franck, Streng 1974]

E. Denert, R. Franck, and W. Streng, "*PLAN2D -- Towards a Two Dimensional Programming Language*," Proc. Fourth Gesellschaft fur informatik Berlin, 1974, pp. 202-213 (Published by Springer Verlag, Berlin, as Vol. 26 Lecture Notes in Computer Science).

[Dennis, Boughton, Leung 1980]

J. B. Dennis, G. A. Boughton, and C. K. C. Leung, "*Building for Data Flow Prototypes*," Proceedings Seventh Annual Symposium Computer Architecture, May 1980, pp. 1-8.

[Ellis, Harduer, Sibley 1969]

T. O.Ellis, J. F. Haefuer, and W. L. Sibley, "*The GRAIL Project: An Experiment in Man-Machine Communications*," Rand Technical Report RM-5999-ARPT, the Rand Corporation, Santa Monica, California, 1969.

[Finzer, Gould 1984]

W. Finzer and L. Gould, "*Programming by Rehearsal*," Byte, Vol. 9, No. 6, June 1984, pp 187-210.

[Glinert, Tanimoto 1984]

Ephraim P. Glinert and Steven L. Tanimoto, "*Pict: An Interactiave Graphical Programming Environment*," Computer, Vol. 17, No. 11, November 1984, pp. 7-25.

[Gurd, Kirkham, Waton 1985]

J. R. Gurd, C. C. Kirkham, and I. Waton, "*The Manchester Prototype Dataflow Computer*," Communications of thje ACM 28, 1 (January 1985), pp. 34-52.

[Hansen 1975]

P. Brinch Hansen, "*The Programming Language Concurrent Pascal*," IEEE Transactions on Sofeware Engineering, Vol. SE-1, No. 2, June 1975, pp. 199-207.

[Hedelman 1984]

W. Hedelman, "*A data flow approach to procedural modeling*," IEEE Computer graphics and Applications, Vol. 4, No. 1, January 1984, pp. 16-26.

[Hwang, Briggs 1984]

K. Hwang, F. A. Briggs, "*Computer Architecture and Parallel Processing*," McGraw-Hill, New York, 1984.

[Lawrie, Layman, Baer, Randal 1975]

D. H. Lawrie, T. Layman, D. Baer, and J. M. Randal, "*Glypnir -- A Programming Language for Illiac IV*," Comm. ACM, Vol. 18, No. 3, March 1975, pp. 157-164.

[Matwin, Pietrzykowski 1984]

S. Matwin and T. Pietrzykowski, "*Motion-picture debugging in a dataflow language*," Preceedings of Graphics Interface 1984, Ottawa University, Canada, pp. 249-260.

[Miklosko, Kotov 1984]

J. Miklosko and V. E. Kotov, "*Algorithm, Software and Hardware of Parallel Computers*," Springer-Verlag, New York, 1984.

[Misunas 1978]

D. P. Misunas, "*A Computer Architecture for Data Flow Computation*," Technical Report, TM-100, Laboratory of Computer Science MIT, March 1978.

[Misunas 1979]

D. P. Misunas, "*Report on the Second Workshop on Data Flow Computer and Program Organization*," Technical Report, TM-136, Laboratory of Computer Science MIT, June 1979.

[Moriconi, Hare 1986]

Mark Moriconi and Dwight F. Hare, "*The PegaSys System: Pictures as Formal Documentation of Large Programs*," ACM Transactions on Programming Languages and Systems, Vol. 8, No. 4, October 1986, pp. 524-546.

[Newman 1968]

W. M. Newman, "*A Graphical Language for Display Programming*," Int'l Computer Graphics symp., Brunel University, Uxbridge, England, August 1968.

[Raeder 1985]

Gerog Raeder, "*A survey of Current Graphical Programming Techniques*," Computer, Vol. 18, No. 8, August 1985, pp. 11-25.

[Rodriguez 1969]

J. E. Rodriguez, "*A Graph Model for Parallel Computation*," MIT Technical Report TR-64, Laboratory for Computer Science, MIT, Cambridge, Mass, September 1969.

[Smith 1975]

D. C. Smith, "*Pygmalion: A Creative Programming Environment*," Ph.D. dissertation, Dept. of Computer Science, Stanford University, (Technical Report STAN-CS-75-499), 1975.

[Sutherland 1963]

I. B. Sutherland, "*Scratchpad, a Man-Machine Graphical Communication System*," Proc. AFIPS Conf., Vol. 23, 1963 SJCC, AFPIS Press, Reston, Va., 1963, pp. 329-346.

[Thacker 1979]

C.P. Thacker, "*Alto A personal computer*," Technical Report CSL 79-11, Xerox PARC, Palo Alto, California, August 1979.

[Weinreb, Moon 1981]

D. Weinreb and D. Moon, "*Lisp Machine Manual*," M.I.T., 1981.

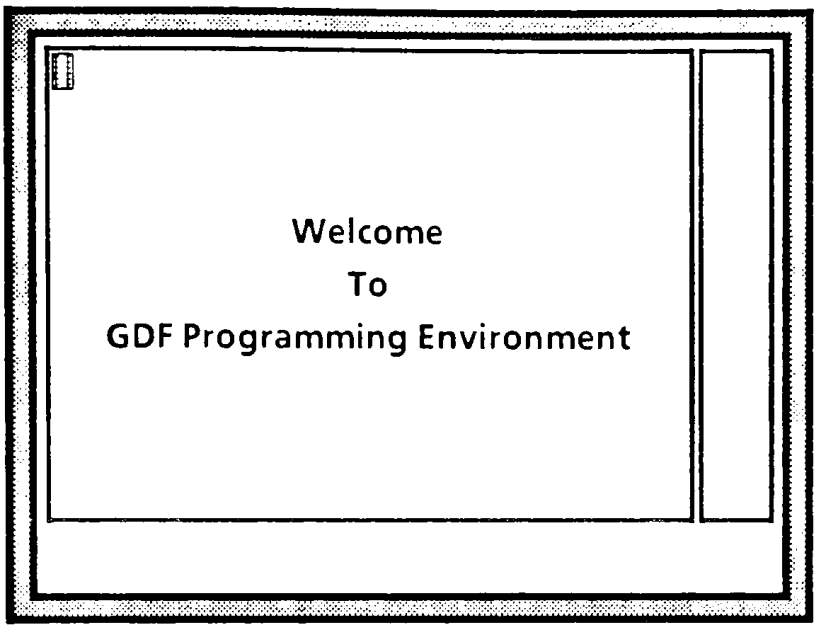
[Yomaguchi, Inamoto 1986]

Kazunori Yamaguchi and Naoto Inamoto, "*A Data Flow Language for Controlling Multiple Interactive Devices*," Computer, Vol. 18, No. 3, March 1986, pp. 28-60.

Appendix A-- GDF Programming System User's Manual

This manual helps the user to step into the world of the Graphical Dataflow Language. It describes the fundamental of dataflow programs, gives a complete and simple step by step example to guide the user in learning and using the system. It also supplies complete information about the graphical editor, which includes the usage of system commands and various of messages. The users can find plenty of of practical tips and advice to help him/her get the most out of the graphical dataflow programming system. Even a novice to graphical programming or dataflow language can become a skillful graphical programmer if he/she can go through the manual very carefully. The knowledge about how the system was implemented is not required.

Graphical Dataflow Programming System
User's Manual



by Jyun-Jier Roland Jehng

Advisors:
Professor Alan R. Kaminsky
Dr. Donald L. Kreher

Rochester Institute of Technology, New York
Computer Science Graduate Department
July, 1988

1. About the Manual

This manual can help you to step into the world of the Graphical Dataflow Language. It describes the fundamental of dataflow languages and programs, gives a complete and simple step-by-step example to guide you in learning and using the graphical programming system. It also supplies complete information about the graphical editor, which includes the usage of commands and various of the messages. You can find plenty of practical tips and advice to help you get most out of the graphical programming system. You do not need to have the knowledge about how the system was implemented. Even you are a novice, you might become a skillful graphical programmer if you can go through this manual very carefully.

Features of the Manual

Section 1 -- **About the manual ...**

Section 2 -- **Preliminaries**

Describes the basic requirements for running the graphical programming system and how to get start.

Section 3 -- **Dataflow Languages and Programs Fundamental**

Gives the basic knowledge about dataflow languages and dataflow programs.

Section 4 -- **What is the Graphical Editor**

Introduces the graphical editor and a sightseeing tour over the system.

Section 5 -- **Step-by-step Example**

Presents a simple example to guide you in learning the system.

Section 6 -- **Graphical Editor Commands**

Summarizes and provides complete information for the graphical editor commands.

Section 7 -- **Error Messages and Warning**

Supplies the explanations of the error messages and warning that you might encounter during the use of the system.

Section 8 -- **Special Notes**

States something that worth for reminding.

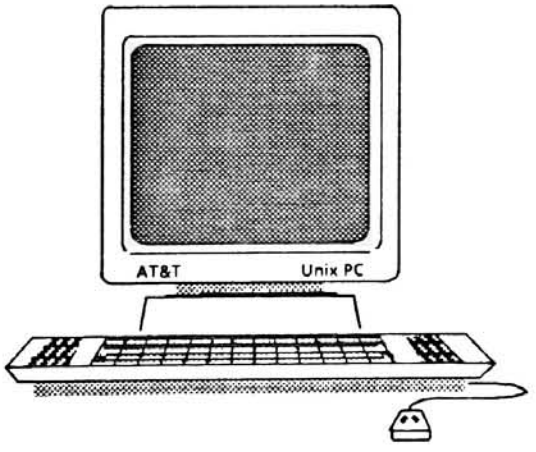
2. Preliminaries

First of all, you must have the Graphical Programming System system floppy disk, a disk contains the system executable program, two help text files, a font file, an operator icon file, and several example programs. The names of these files are:

The names of the files in the system disk are:

run	-	system executable file
font	-	font file
opicon	-	operator icon file
gdf_help	-	help text file
gdf_intro	-	introduction text file
ex*	-	example programs

Secondly, find out the computer you can use. It must be an AT&T Unix™ PC with a hard disk drive, one floppy disk drive, a keyboard, and a mouse. The monitor is 720 by 348 pixels resolution. The Unix™ Operating System is Version 3.50 or newer version.



Then, login your account, mount the floppy disk, and copy all the files contained in the system floppy disk to hard disk. You are suggested to make a sub directory under your home directory, so that you can manage all of your dataflow programs in an independent directory.

```
mkdir gdf
cd gdf
cp /mnt/* .
```

List the files in the current directory, you can see

```
ex1       ex3       font      gdf_intro   run
ex2       ex4       gdf_help  opicon      example
```

Now, you can type

```
run
```

and hit <RETURN> to start enjoying your graphical programming. It's better to have a copy of this manual in your hand when you want to use the system. The manual might assist you in using the system.

3. Dataflow Languages and Programs Fundamental

If you have already had the knowledge about what dataflow language is and how to program a graphical dataflow program graph, you can skip this section. If not, you are recommended to take a few minutes to read this section. In this section, the graphical dataflow language (GDL) is also introduced.

Dataflow languages are computer languages that supports the notion of data flowing from one operation to another. The flow concept gives the dataflow languages the advantages of representing the programs in directed graphs. Each node in the graph stands for an instruction, an operation, or a function; and each arc between two nodes is the medium where data flows. The arrow of an arc is the direction which the data token flows. On the graph, data are treated as tokens. Tokens may carry the values and the types of the data. The execution of instructions of a dataflow program is not controlled by the sequence of the instructions, instead, is controlled by the data availability of the operations in the program. When a node's input data are available, the node is said to be executed or fireable. The output of an instruction depends on the input data and the operation that the instruction performs. Theoretically, several nodes can be executed concurrently if these nodes are fireable at the same time.

For a computer with parallel processing architecture, the way of the execution of dataflow can improve the speed of operations, and explore the performance of the computer. Up to date, dataflow languages have not had a generally recognized format. At the previous study of dataflow languages, a dataflow program is first drawn in the form of a graph. Each node of the graph is assigned a number by the programmer. Then, the programmer translates the graph into a textual form which is a statement-by-statement program. The translation must follow a set of rule stipulated by the simulator or particular application. This kind of statement representation somewhat loses the nature of expressing the idea of dataflow in its program. On the other hand, to debug such a concurrently executed dataflow program is very difficult.

A dataflow program has some important properties: 1) *Free from side-effect.* This property can ensure that data dependencies are the same as the sequencing constraints. 2) *Locality of effect.* Instructions do not have unnecessary far reaching data dependencies. 3) *Equivalence of instruction scheduling constraints with data dependencies.* All the information needed in executing a program is contained in its

dataflow graph. 4) *Single assignment convention.* A variable may appear on the left side if an assignment only once within the area of the program where it is active. It a dataflow program is expressed in its graphical form, these properties can be seen very easily.

The Graphical Dataflow Language (GDL) is defined based on the advantages described on the above. It provides 29 basic operators for users to construct dataflow programs. The operators are icons (see fugure 1). The operation performed by the

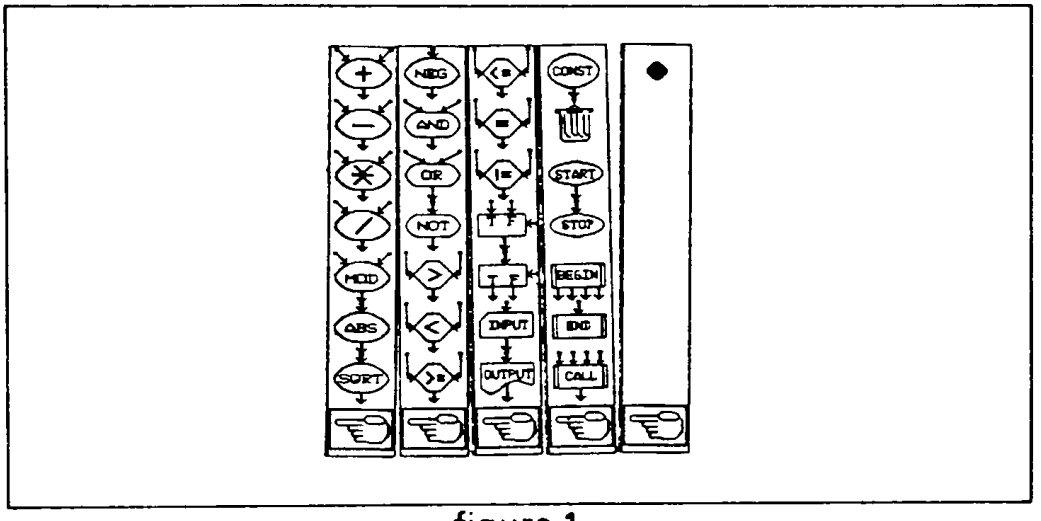


figure 1

operator is expressed by the shape and symbol of the icon intuitively. A program written in GDL only has graphical representations. You do not need to convert the graphs into any other forms of statements manually. The program graphs can be executed by the simulator directly. You can also take the advantages of the simulation control options which offered by the simulator to monitor the progresses of the execution of your dataflow program. The single-step simulation control option makes the debugging of a dataflow program much more easier than that of a traditional textual dataflow program.

In the next three section, you will be taught how to use the GDF system to construct and execute a graphical dataflow program.

4. *What is the Graphical Editor*

The graphical editor is an editor which provides functions for users to edit programs. Unlike the "vi" editor in the UNIX™ system and the "eve" editor in the VAX™ system, the graphical editor in the GDF system is used to draw dataflow program graphs instead of writing textual programs.

There are two major purposes that the graphical editor is supplied. First, you can construct your graphical dataflow programs by making use of the functions supported by the graphical editor. Second, you can run your dataflow program in the environment furnished by the graphical editor.

What does the graphical editor look like? The picture shown in figure 5 is the basic layout. The screen is divided into three parts: the scratchpad area, the operator menu panel, and the system message board. The scratchpad area is the space where you can draw your dataflow programs. The twenty-nine operators provided by the system are displayed in the operator menu panel. Seven operators are shown in one plate on the operator menu panel. You can click the bottom-most icon which stands for "*more*" to see different operators. The system message board is used by the system for displaying hints or warning messages during the progresses in editing or executing dataflow programs.

The function provided by the graphical editor to assist you in editing dataflow program graphs are: *copy__locate*, *select*, *move*, *delete*, *connect*, *property*, *redraw*, *new*, *functions*, *switch__pages*, *save*, *retrieve*, and *help*. On the upper-left corner of the scratchpad area is a small dotted box which is called the **command-request-box**. Every time you press down <B1> on the command-request-box, a pull down menu will show up. This pull down menu contains all the commands available in the system. You can keep pressing the button and moving the mouse cursor up and down inside of the menu, which causes the pointed command to be highlighted. You can release <B1> when the desired command is highlighted. The chosen command will be the **current chosen command**, and is recorded on the system message board. The complete explanations of the command will be discussed in section 6 of this manual.

If you have a program currently existing in the graphical editor, you can choose the command "*execute*" to run the graphical dataflow program. When the command is

chosen, the main graph of the program will be displayed on the screen, and the connections of the graphs contained in the current edited program will be checked first if the current program has not been executed since the last modification. Seven simulation control options will be displayed on the left-hand side of the scratchpad area after the connection checking is done and no connection errors were found. During the execution of the program, the fireable nodes are highlighted and the firing node is flashing. The seven simulation control options are speeding up the simulation speed, pausing the simulation, single-stepping the simulation, continuing or resuming the simulation, invoking a fast simulation, giving up the simulation, and speeding down the simulation speed. Please refer section 6 for details about the control options.

In addition to the method of clicking command-request-box, the system provides another method for you to choose a command. The central mouse button <B2> is called CMD button. When you click <B2>, you can see the message on the system board shown as:

<B1> set command <B2> forward <B3> backward.

<B1>, <B2>, and <B3> represent the left-most mouse button, the central mouse button, and the right-most mouse button respectively. Now, if you click <B1>, the name of the command which displayed on the *current command line* will be set to the current chosen command. If you click <B2> or <B3>, you can see the name displayed on the current command line is changed. Keep clicking <B2> or <B3> until you find the command you want, then click <B1>. The names of commands are displayed in a certain order which is the same as the order shown on the pull down menu mentioned before. The forward order is: *copy_locate*, *select*, *move*, *delete*, *connect*, *property*, *redraw*, *new*, *functions*, *switch_pages*, *save*, *retrieve*, *execute*, *help*, and *quit*. This command choosing method is especially helpful when you want to make minor changes on an existing graph. It is very easy to use this method in choosing "move", "delete", and "select" commands.

5. *Step-by-step Example*

In this section, you will be given a simple dataflow program example which is a program that reads three integer inputs from the keyboard, calculates the multiplication of the first input integer and the sum of the second and third input integers, and displays the result. If this is the first time you use this system, please follow the instructions, take action as told in this example, and compare your screen display with the figures shown in the next few pages. Going over this example carefully will help you a lot in your further utilizing the system. In the following demonstration, when the "click mouse" term is used, it means clicking <B1> if there is not any special comments.

As mentioned in section 2, you can type "run", to start running the GDF programming system. The first screen you can see should look like figure 2.

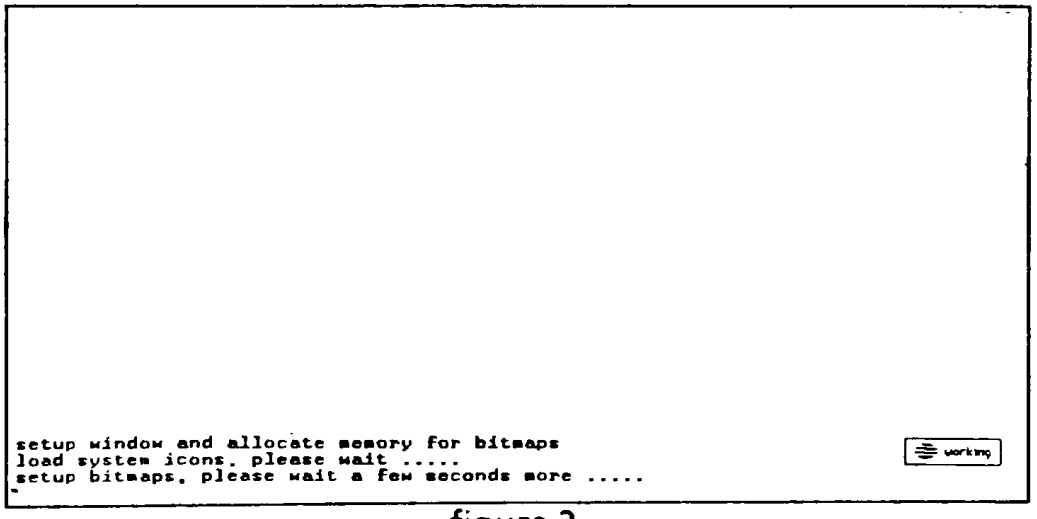


figure 2

After the preparation work is done, a welcome message screen is displayed (see figure 3).

If you want to read a few paragraphs of the system introductions, click mouse on the "YES" window on the upper-right corner on the screen. A new window will be opened and used to display a brief introduction of the GDF programming system. You can click the UP and DOWN arrows to scroll the screen back and forth, or click the CLOSE symbol or any position except the arrows to finish reading the introduction. The next screen should be the same as figure 4.

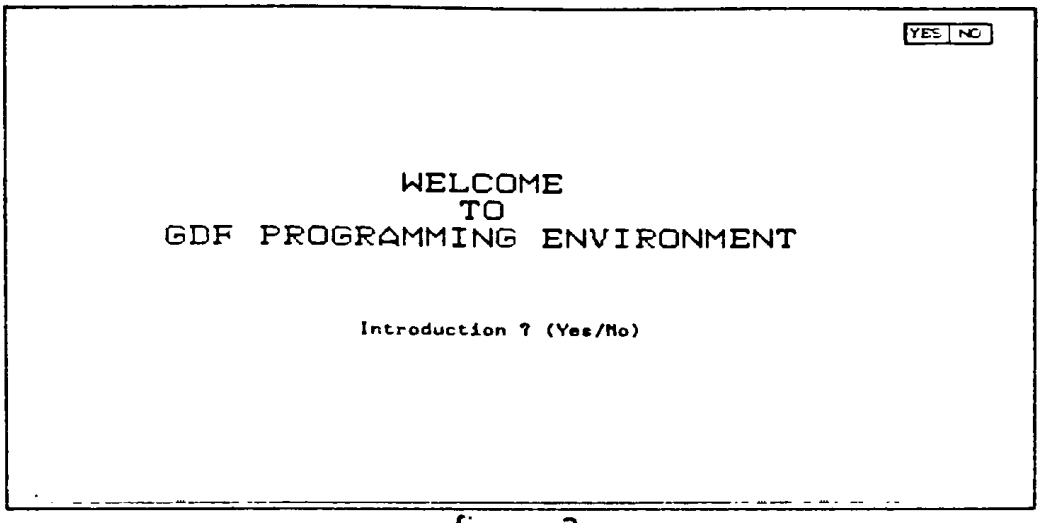


figure 3

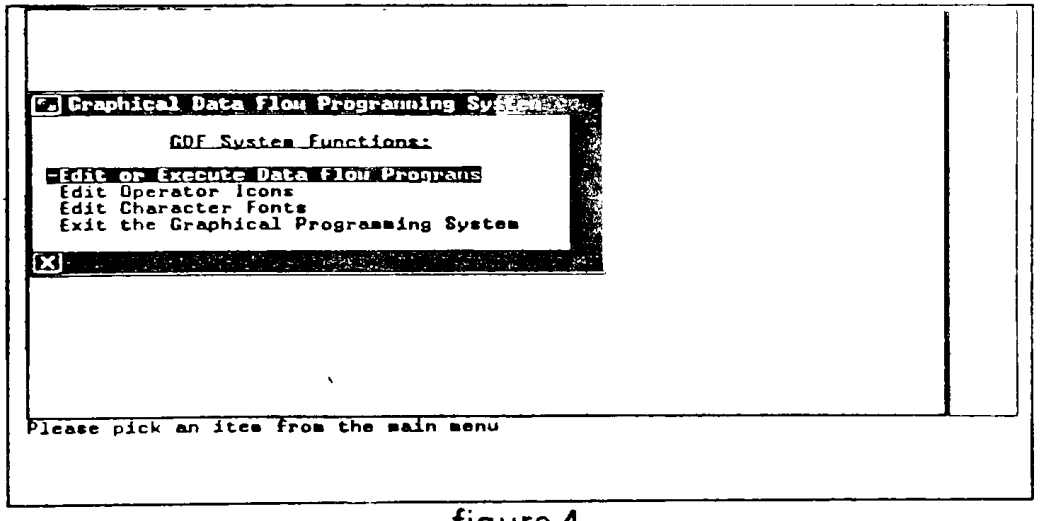


figure 4

A main function menu is displayed. Moving the cursor inside the menu window can cause one of the options to be highlighted. Clicking mouse on the first option, you are able to enter the graphical editor to start editing or executing dataflow programs (See figure 5).

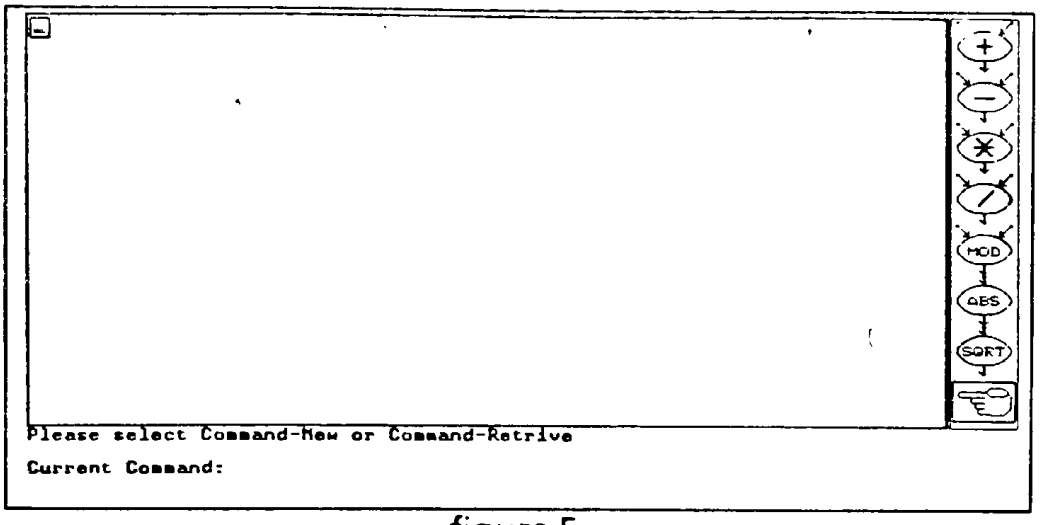


figure 5

In figure 6, you can click mouse on the command-request-box, keep pressing the

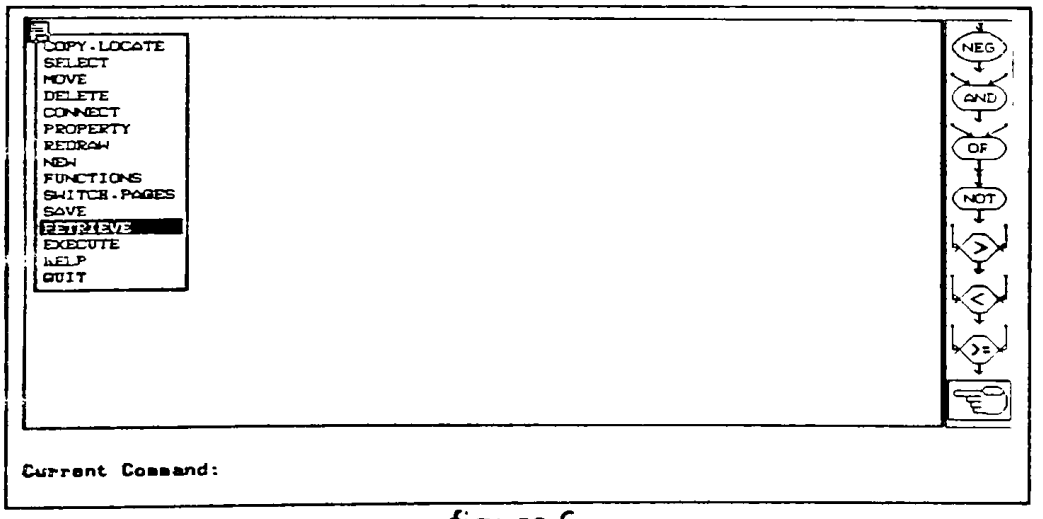


figure 6

mouse button and move the mouse cursor up and down inside the menu. Highlight the "retrieve" command and release the button. A new window is shown on the screen, waiting for you to give the name of the file you want to retrieve (see figure 7). Type in the program file name "ex4", then click the "OK" block. The program graphs

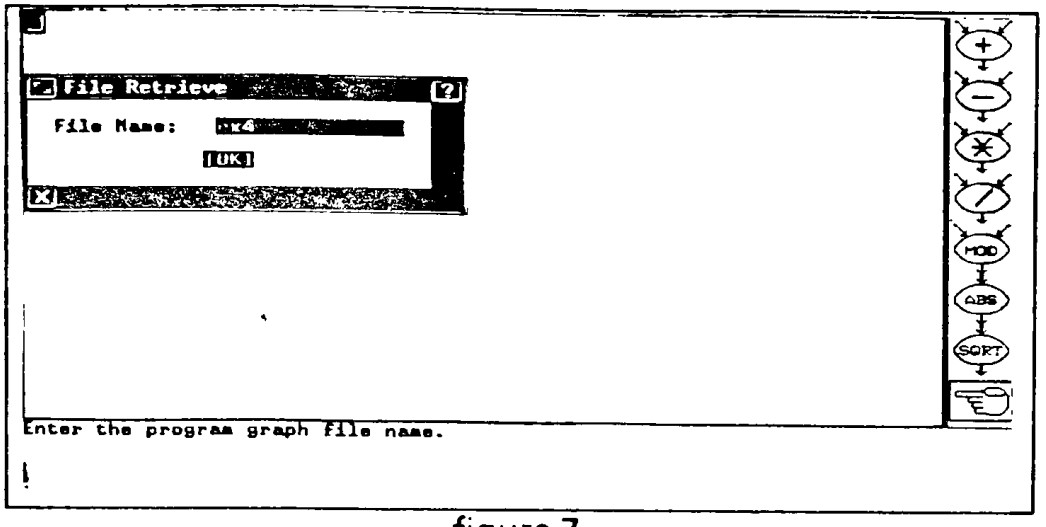


figure 7

of "ex4" are retrieved and the main function graph is displayed on the screen (see figure 8).

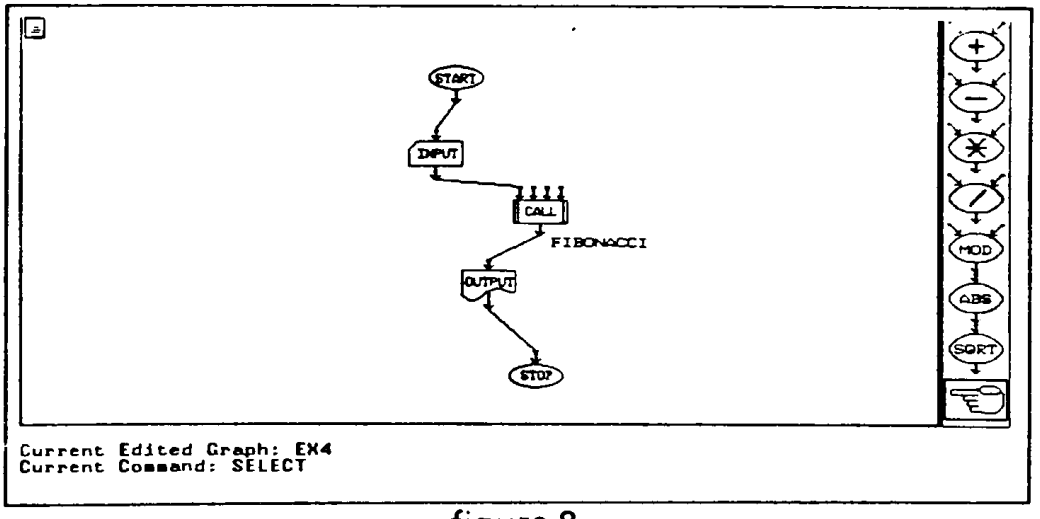


figure 8

Click the command-request-box and choose the "switch_pages" command (see figure 9). The names of the graphs contained in the 'ex4" program are displayed in a new window (see figure 10). Move the mouse cursor to highlight the second one of the graph names. Click the mouse, then you can see the graph of the function "fibbonacci" is shown on the screen to replace the original displayed graph in figure 11. By the way, you can see the system message board is recording the name of graph and the name of the command you chose or selected.

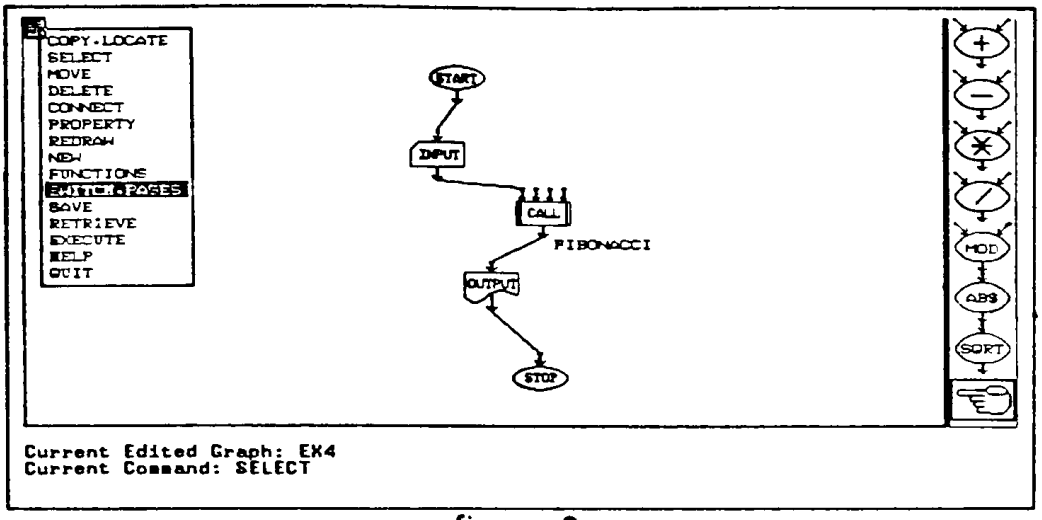


figure 9

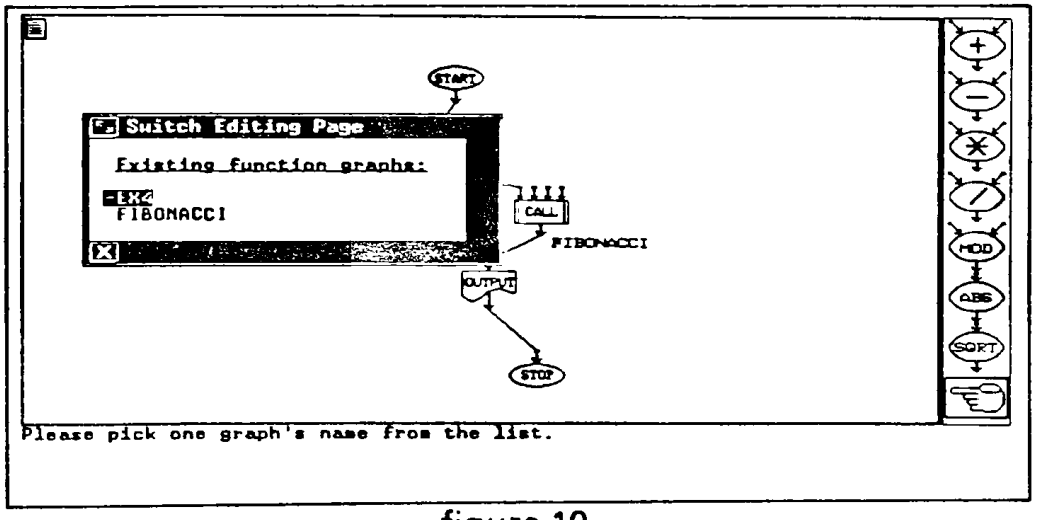


figure 10

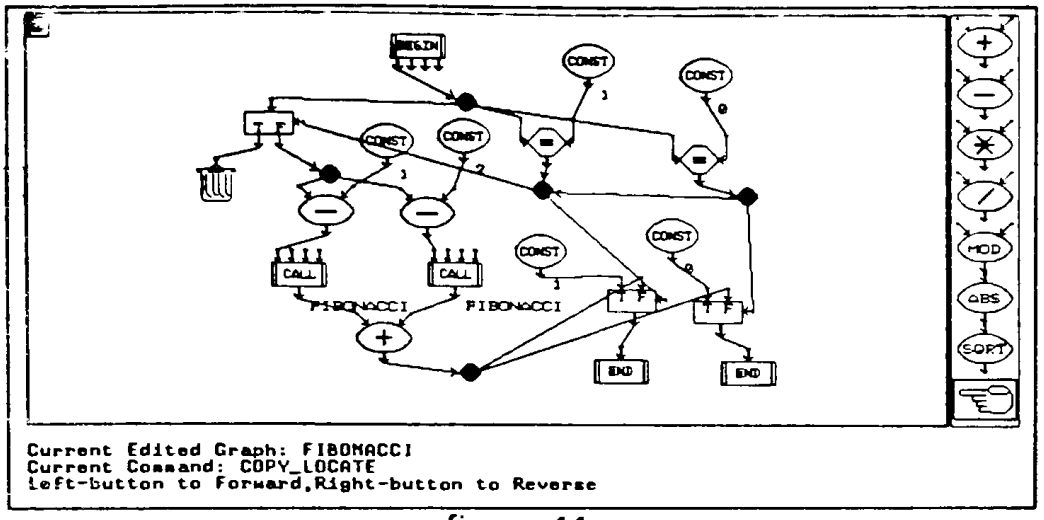


figure 11

Click the command-request-box and choose the "execute" command (see figure 12),

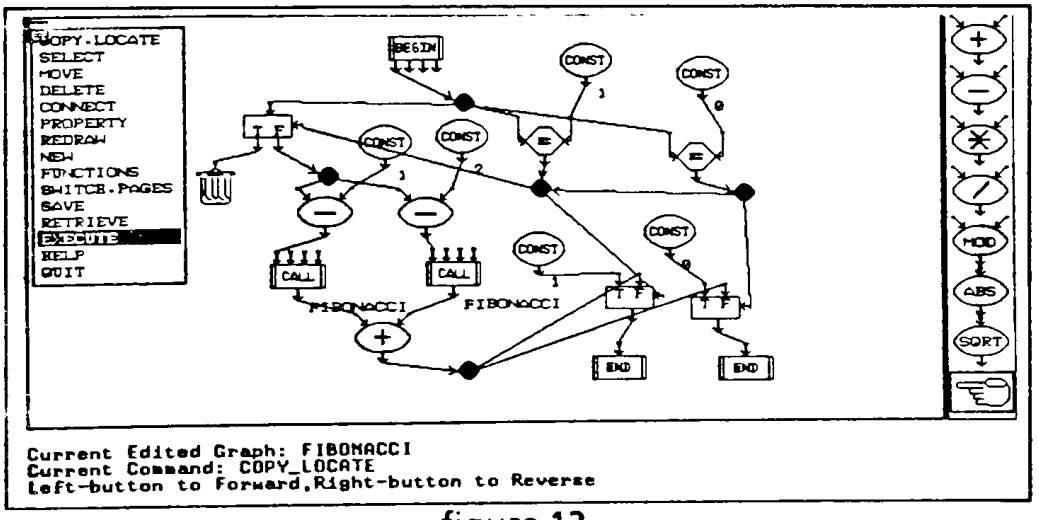


figure 12

you can notice that the screen resumes the main function graph of "ex4" program, seven simulation control options are displayed on the left-hand side of the screen, and a node starts flashing.

It means the dataflow program is being executed. When the CALL node comes to be executed (in figure 13), the graph which the CALL node calls is refreshed on the

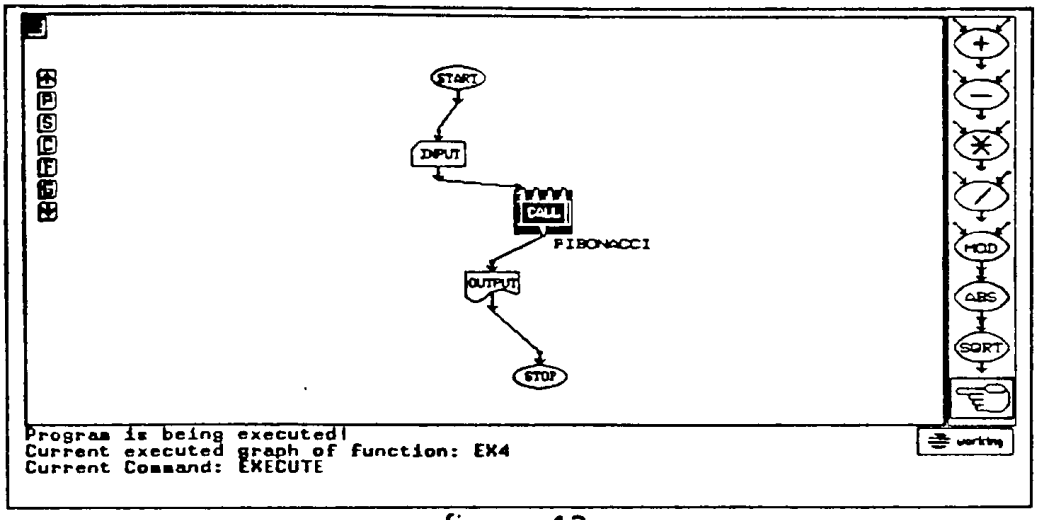


figure 13

screen and the execution continues (in figure 14). At the end of the execution of the

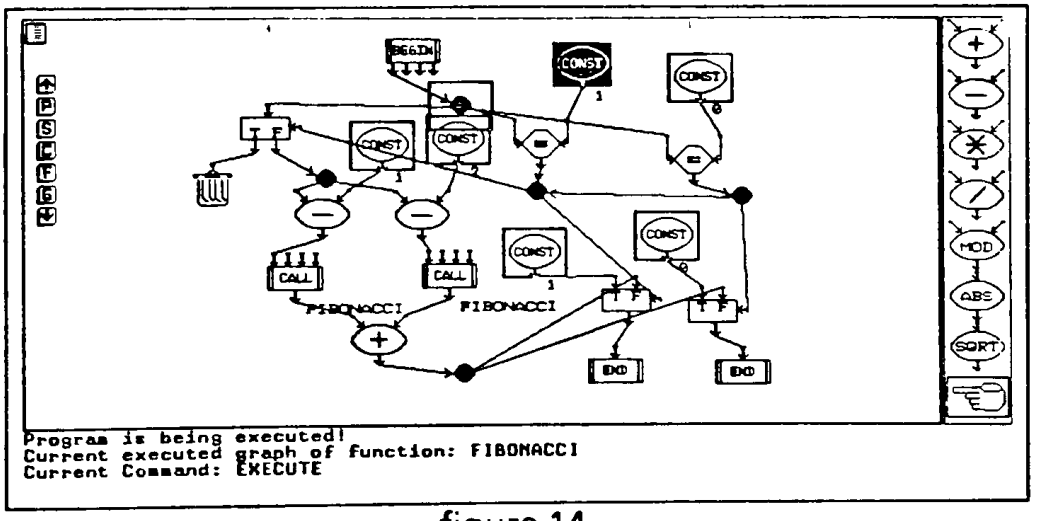


figure 14

"fibbonacci" function, a data token is returned back to the main graph. At this time, main graph is displayed again, and the successive node of the CALL node, which is an OUTPUT node is then executed. A new window is opened to display the result which was returned from the previous CALL function. To close the window (in figure 15), you can hit <CR> key or click the mouse in any position in the screen.

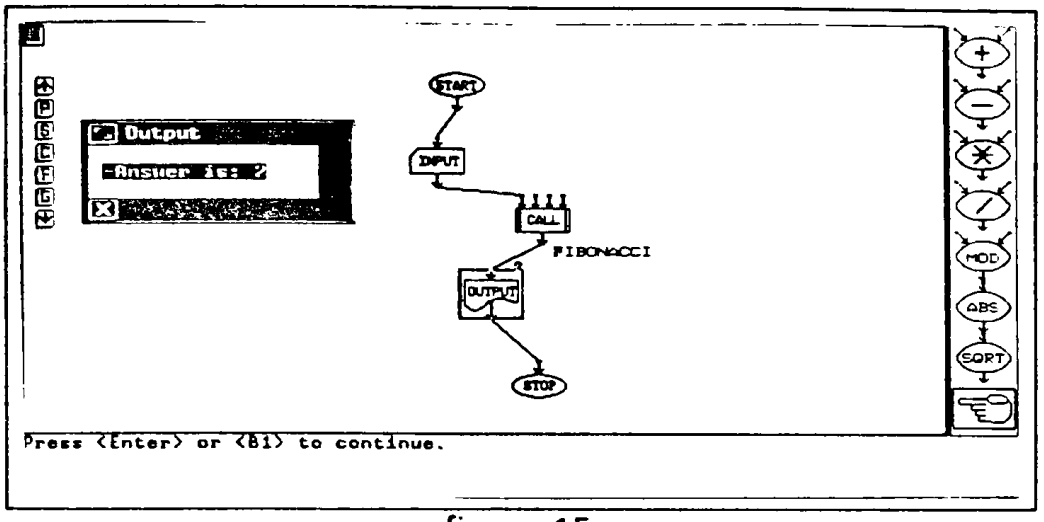


figure 15

Now, we want to leave the current editing session to start another one. Click the command-request-box and choose the "quit" command, like figure 16.

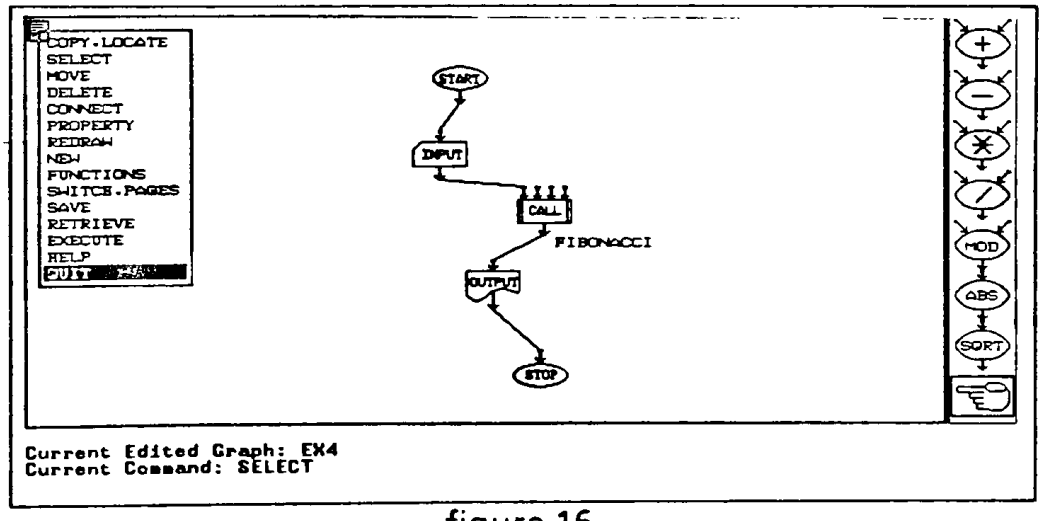


figure 16

A message displayed in an attention window is reminding you that there is a program residing in the graphical editor (see figure 17), you can either cancel the

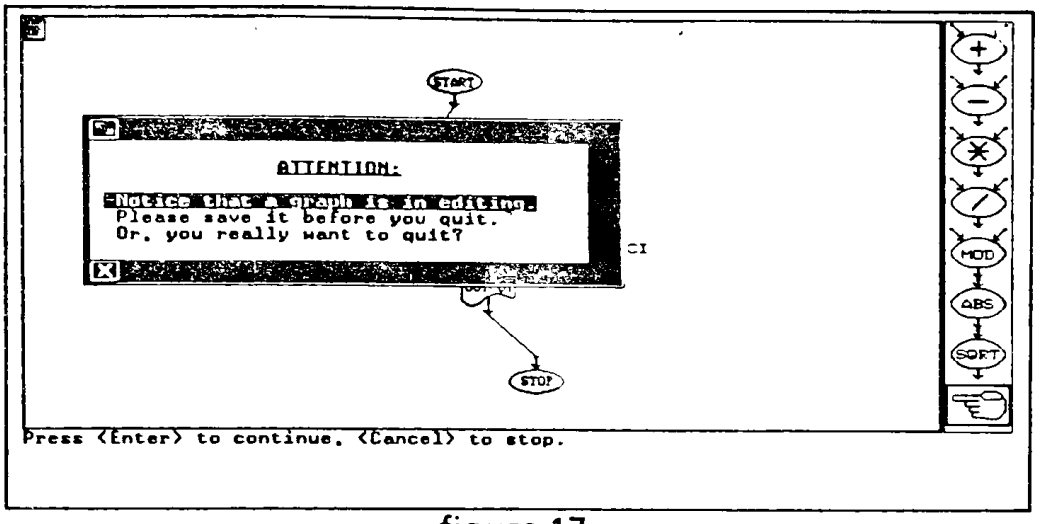


figure 17

"quit" command by clicking the "cancel" symbol which is the cross symbol at the bottom left corner of the message window, or carrying out the command by clicking any other position again. Take the later action. At this time, a different attention window is shown (in figure 18) to make sure that you really want to leave the

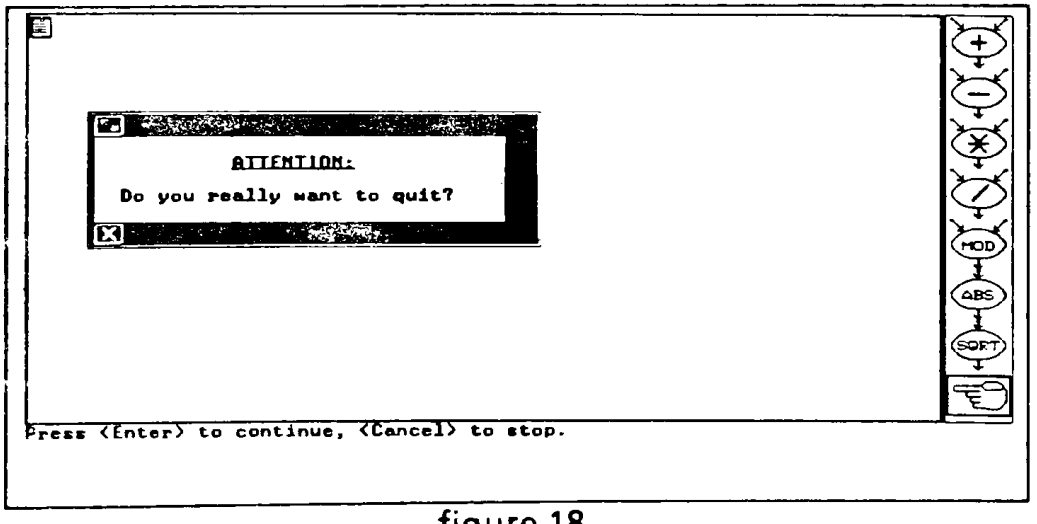


figure 18

graphical editor. Click the "cancel" symbol to cancel the "quit" command. So that you will stay in the graphical editor.

From now on, you are going to start a new program and use the editing commands.

Click the command-request-box to choose the "new" command, a new window will be opened and waiting for you a name for the new program. Give "example" as the program name. This name will also be used as the name of the main graph of the program (see figure 19, 20, and 21). The screen is cleared like figure 22, and the

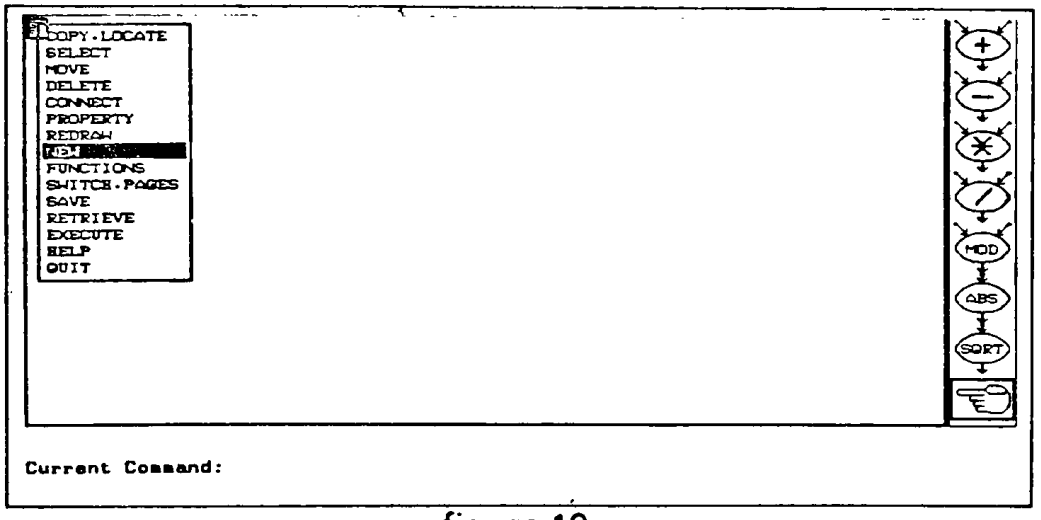


figure 19

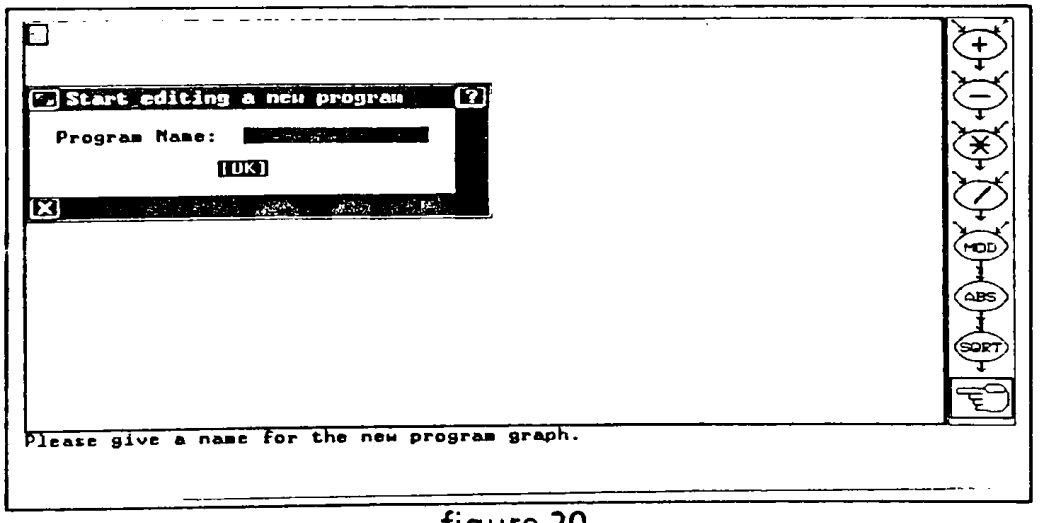


figure 20

name of the graph is recorded in the system message board, also the default current command is set to the "copy_locate".

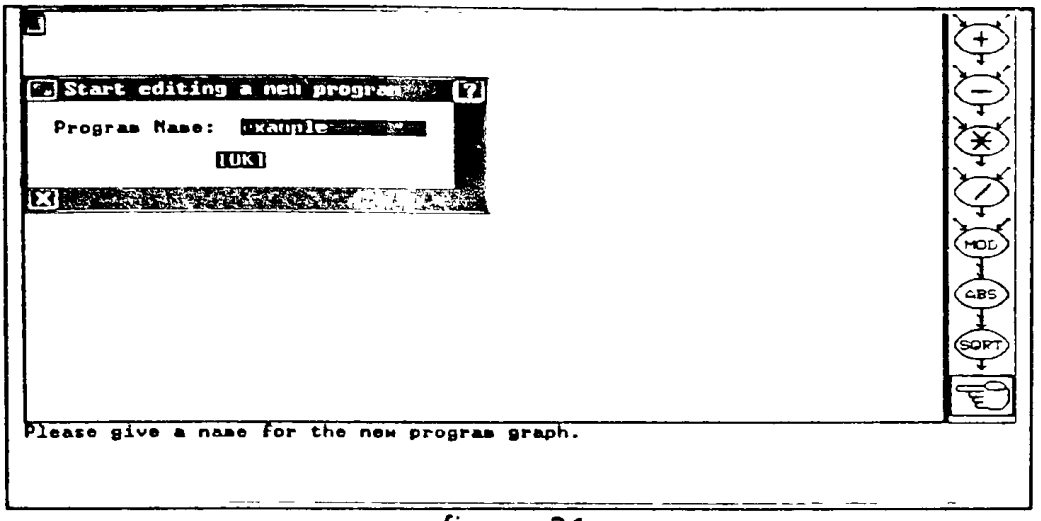


figure 21

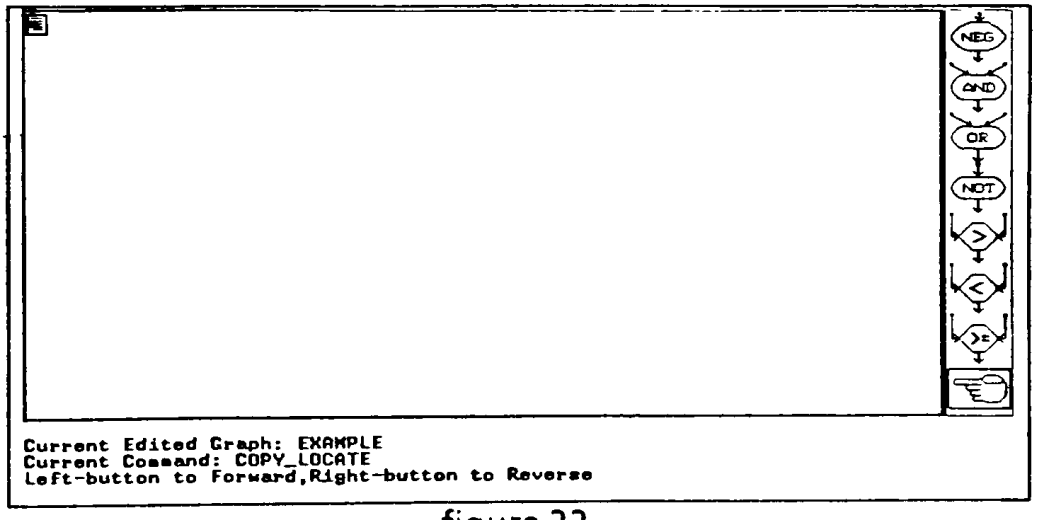


figure 22

Try to click the mouse on the bottom-most icon which is a hand-shape symbol and stands for "more operators". Check figure 23, 24, and 25. From the different icon

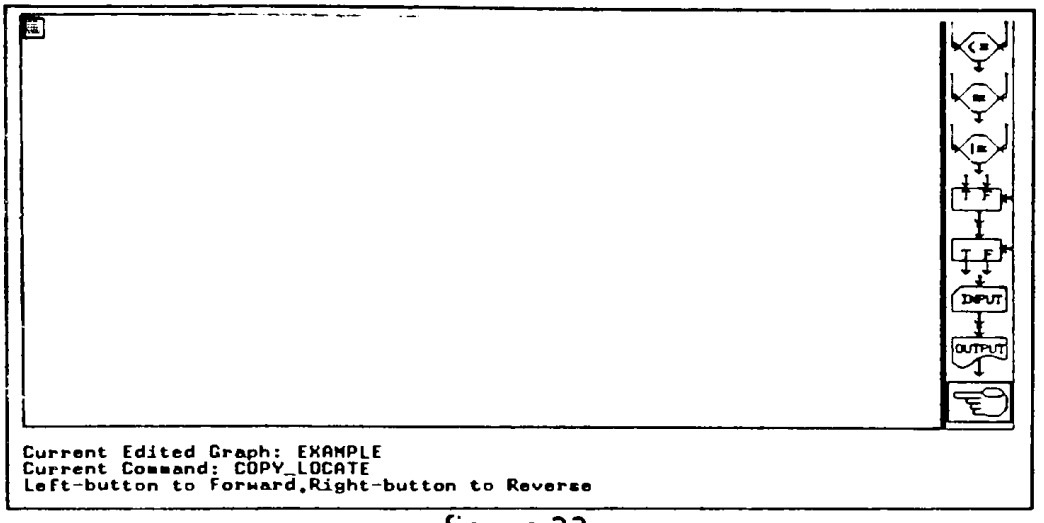


figure 23

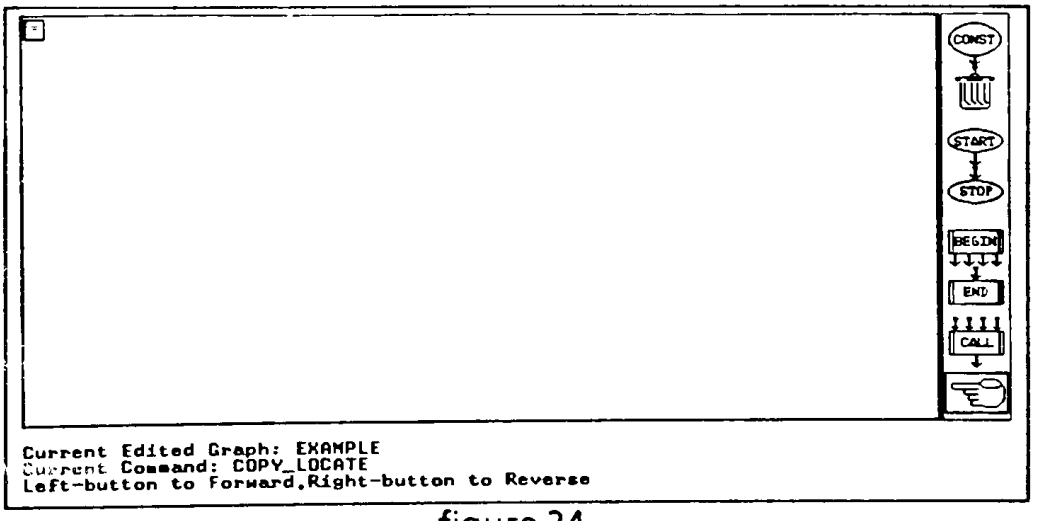


figure 24

menu plate, you can see seven different operators are displayed. If you try to press the <B1> or <B3> several times, you can see the different order of the plates are changed. The operator icon plates are arranged in a certain order. Use those two mouse buttons alternatively at your own convenience. Keep clicking the mouse on "more" icon until you find the first operator on the menu is CONST operator.

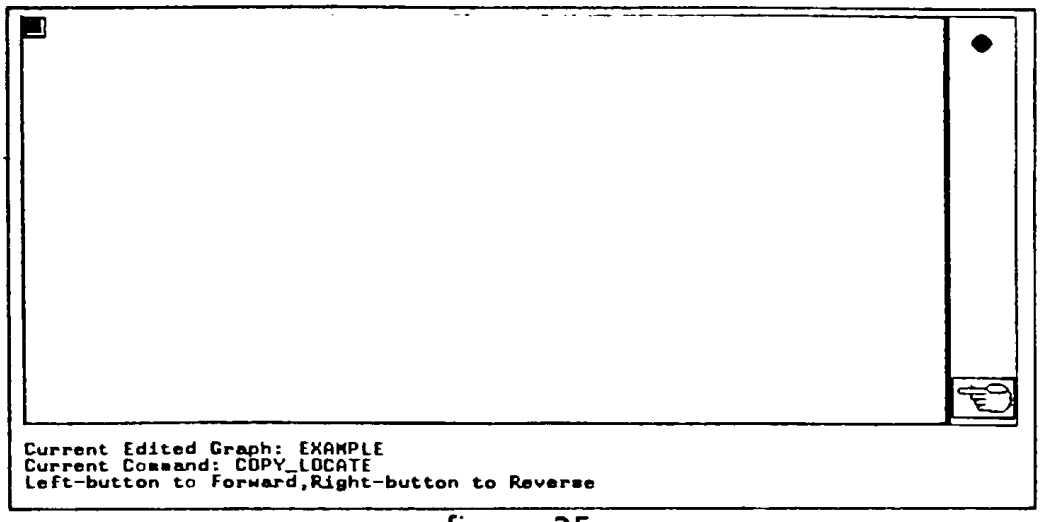


figure 25

Click the mouse cursor on the operator menu panel, you will find that the selected operator is highlighted. The shape of the mouse cursor is changed to a "carrying" icon, and the name of the selected operator is shown in the first line on the system message board. Now, select the START operator, move the cursor whose current shape is a "carrying" icon into the area of scratchpad where you want to locate the START node. Click the mouse on that location, you can locate the node and see the mouse cursor is changed back to the original shape (see figure 26 and 27).

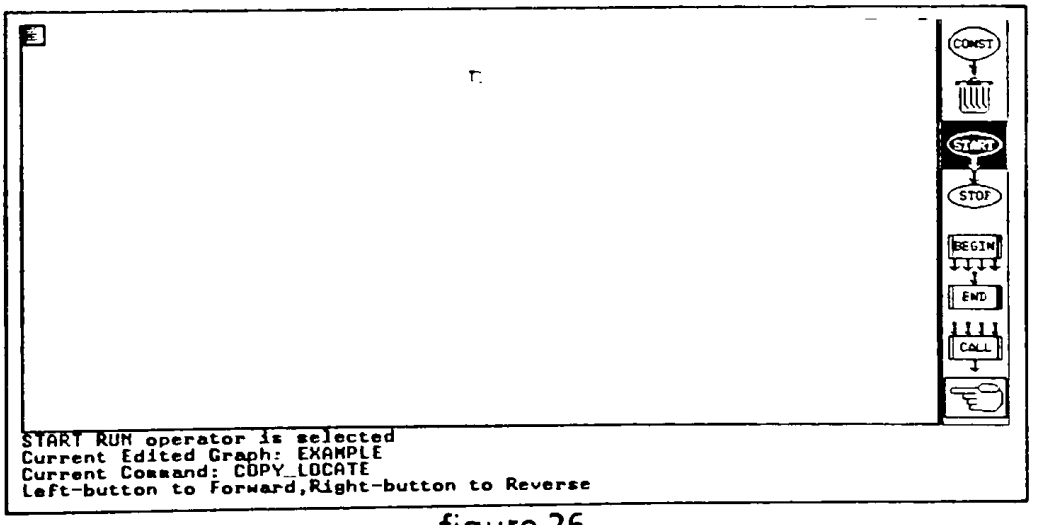


figure 26

Next step, you have to select INPUT operator three times from the menu panel. Whenever after you locates an INPUT node, the system will open a temporary window, and ask for the type of the input token and the input prompt string you like.

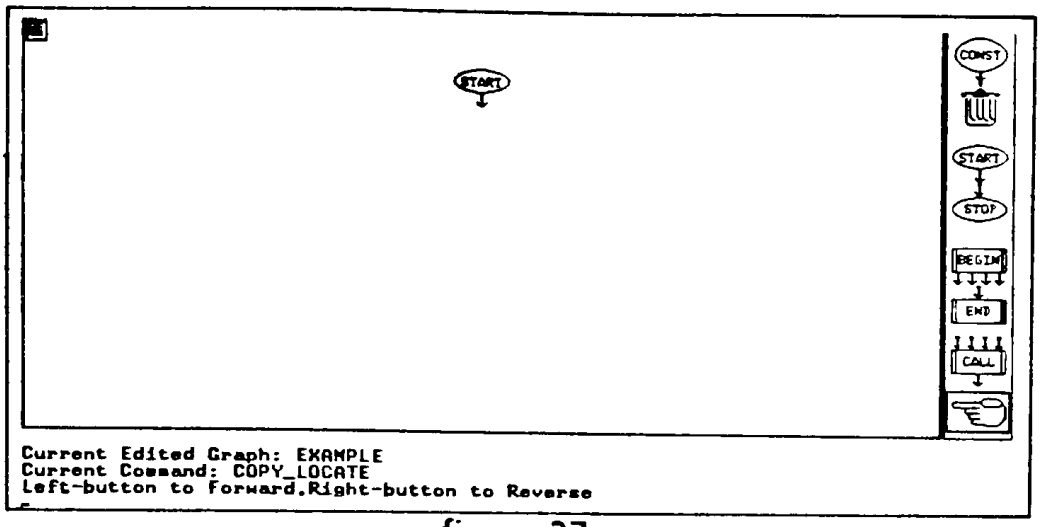


figure 27

The prompt string is used when the program is executed and the INPUT node is firing. You can create any string you like to give you a cue when the program is asking for an input object. You have three ways to tell the types. You can key in the type directly, like "integer" or "character", or you can click <B2> to ask the system to display the possible choices then click the one you want, or you can click <B3> on the highlighting field several times until the choice that you want is displayed in the field. See figure 28, the type of the input token has been set to "integer". In addition

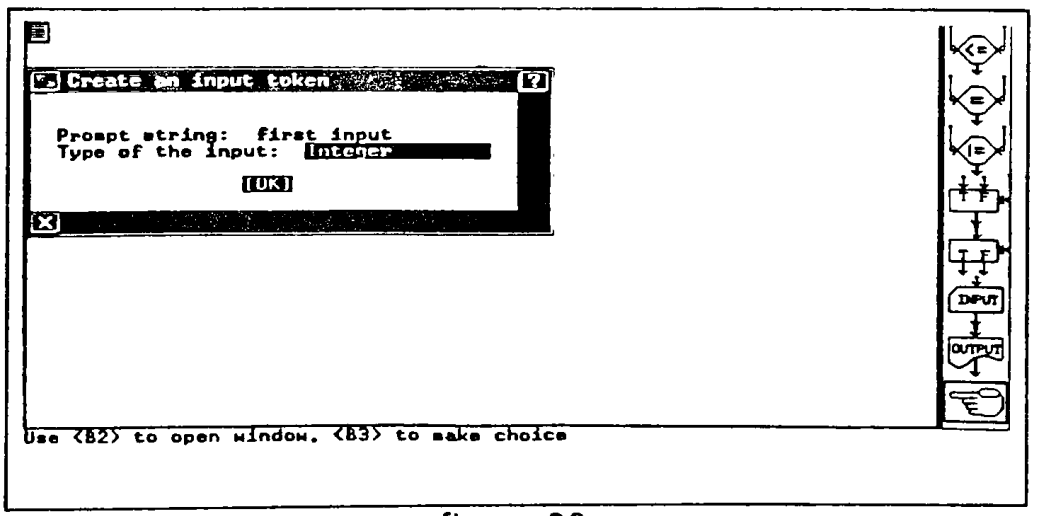


figure 28

to the INPUT operator, when you select CONST operator, OUTPUT operator, or CALL operator, you will be questioning similarly.

Follow the demonstrations in figure 29 and 30 to copy all the necessary operators to

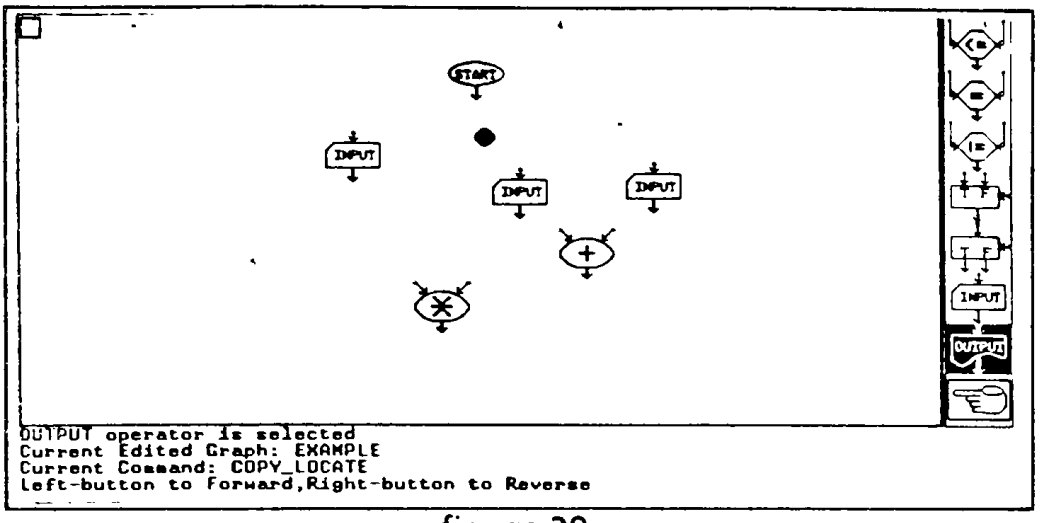


figure 29

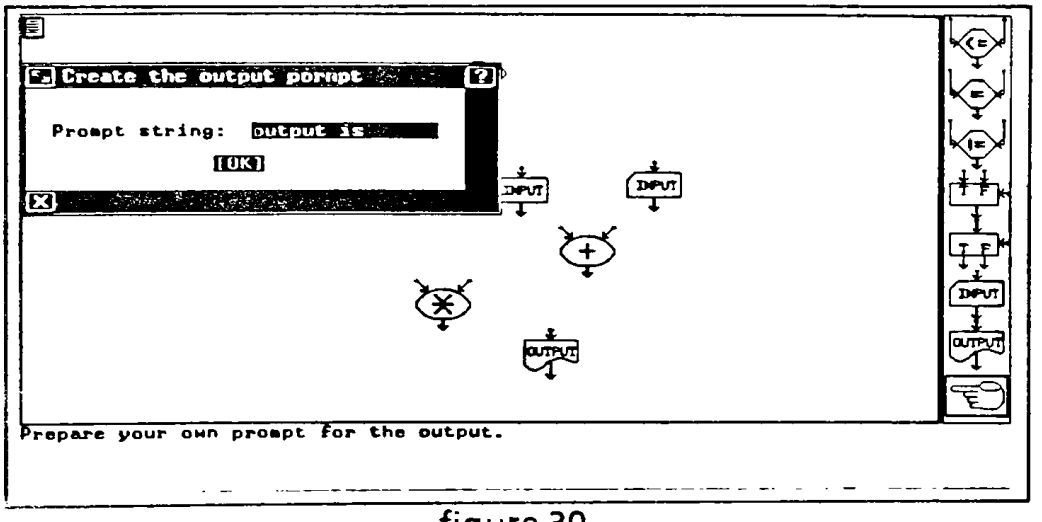


figure 30

the scratchpad. Click the command-request-box to choose the "connect" command, (see figure 31).

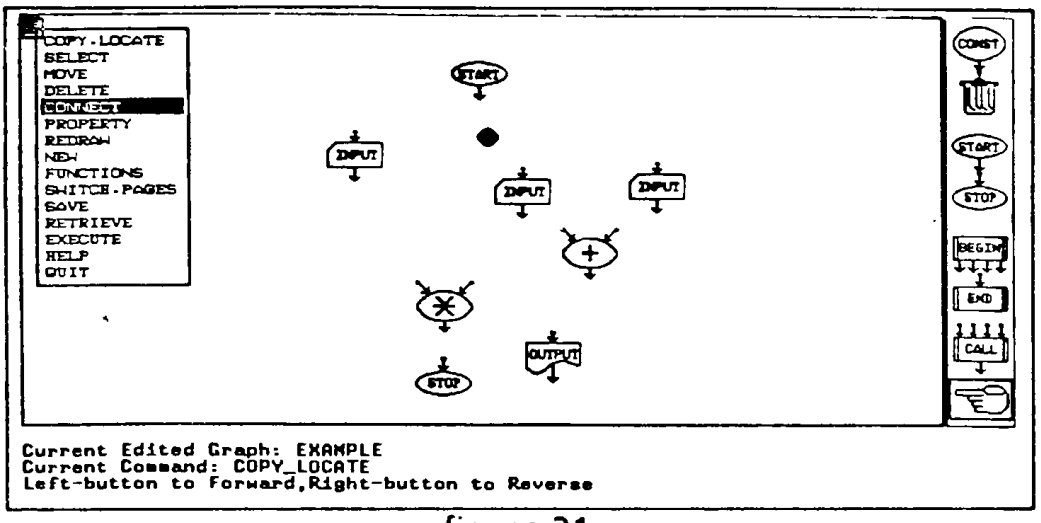


figure 31

you can start to construct necessary arcs between pairs of nodes. Notice that the current command displayed on the system message board is "connect". Click the node with the output port first (see figure 32), and then click the node with the input

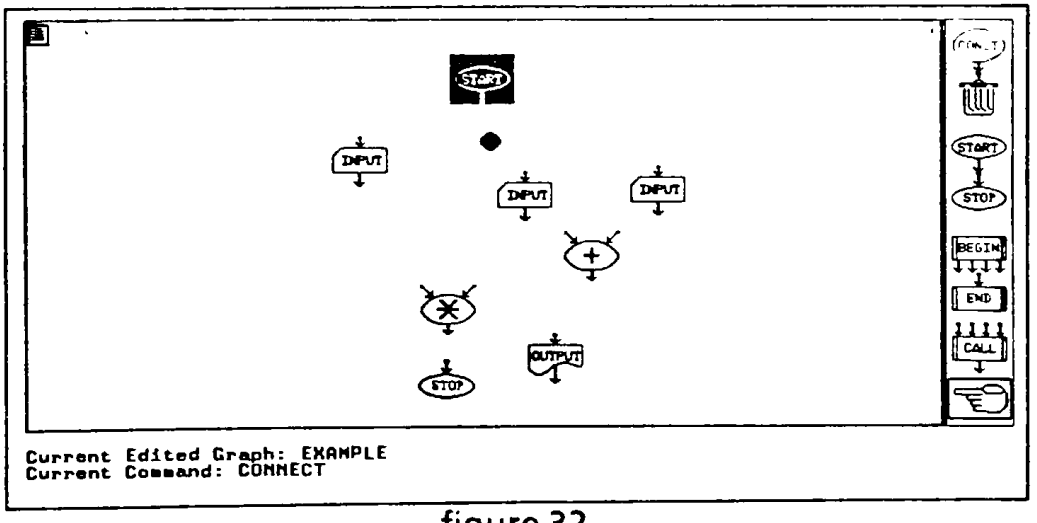


figure 32

port. The arc between these two nodes is constructed (see figure 33). Use this method to connect all the necessary arcs. A complete graph is shown (see figure 34).

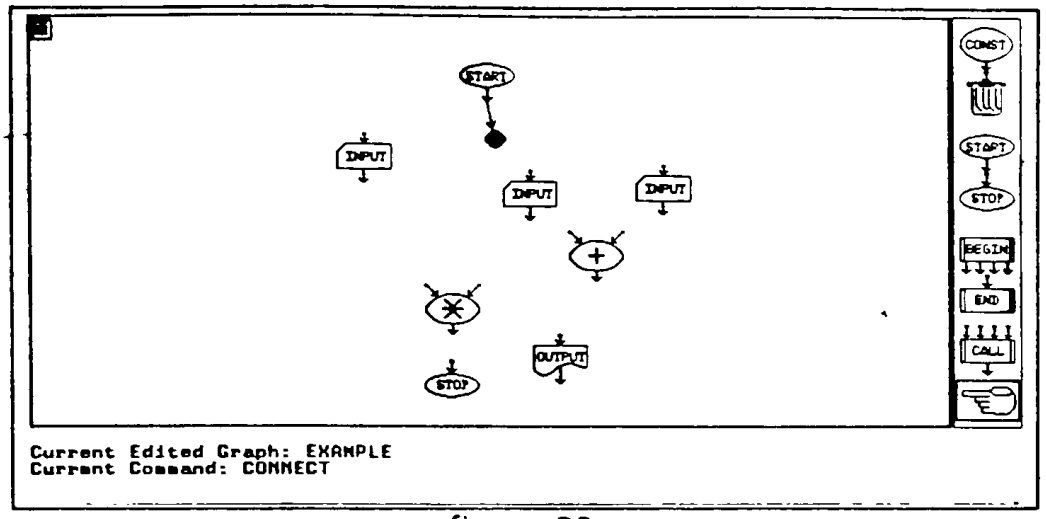


figure 33

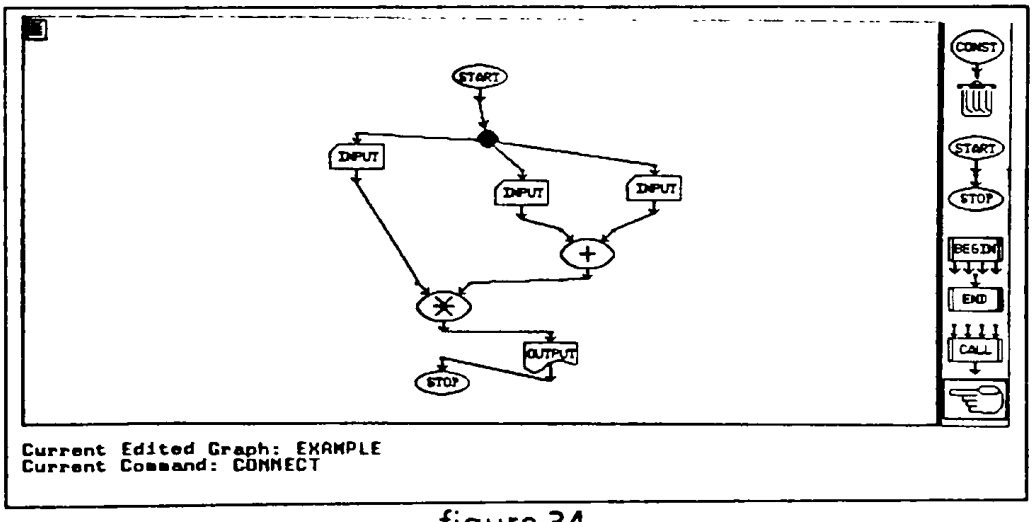


figure 34

In addition to the way by clicking the command-request-box to get a command, you can use <B2> to choose a command. The way of using CMD button has been discussed in previous section of this manual. In figure 35 and 36, the current

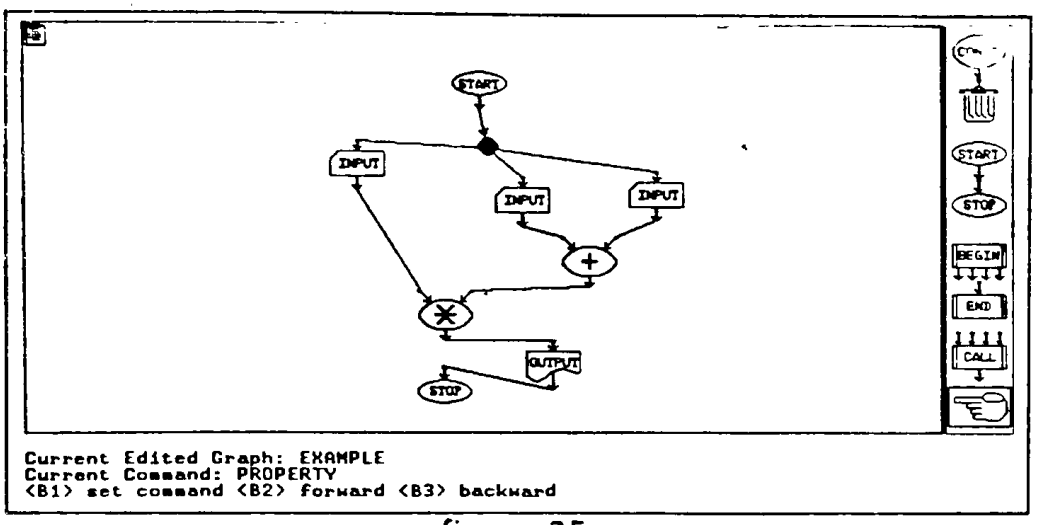


figure 35

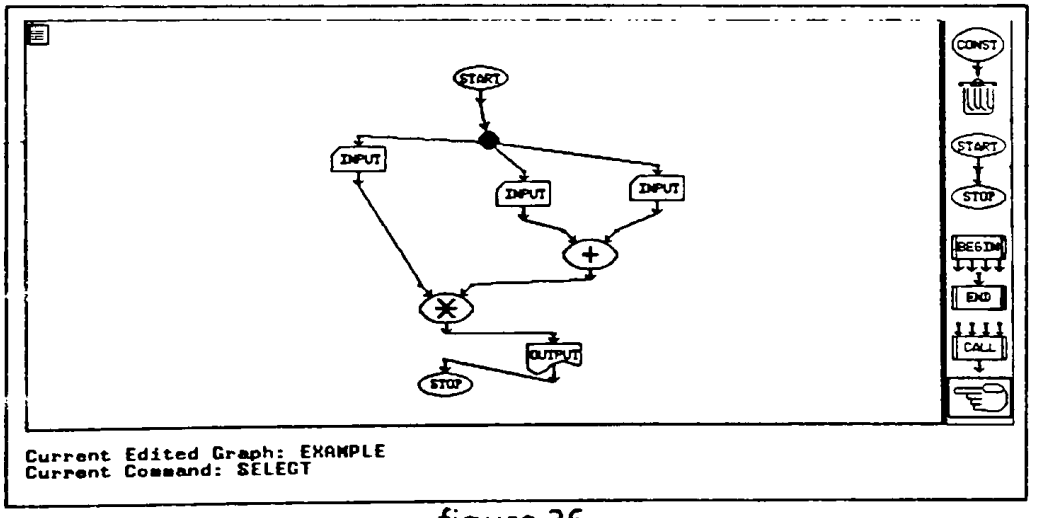


figure 36

command is set to "select" command. Now we want to move the START node to a pertinent location. Click <B2> and keep clicking <B2> or <B3> until the current command displayed is "*move*". Hold down <B1> button and move the cursor, you can see the cursor is changed to a "moving" icon with a bright square, see figure 37. Move the bright square to a pertinent location then release <B1>. The START node is then moved to the new location, and the connections are also moved.

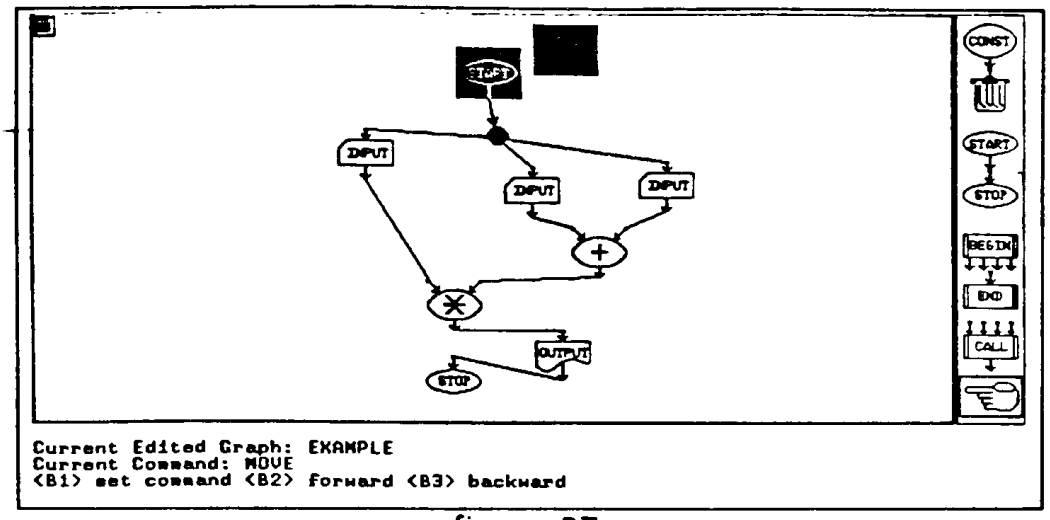


figure 37

Use this method to adjust the locations of all the nodes. An adjusted graph is shown in figure 38.

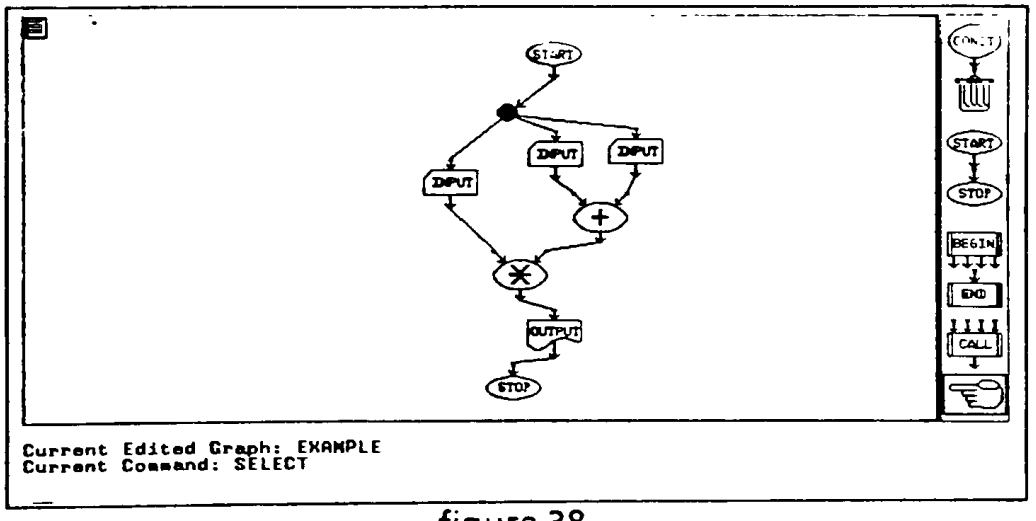


figure 38

Now it is the time to execute the graph program you just edited. Choose the "execute" command, the system will check the connections of the graph (see figure 39). After the checking is done, the graph is executed (see figure 40). The simulation

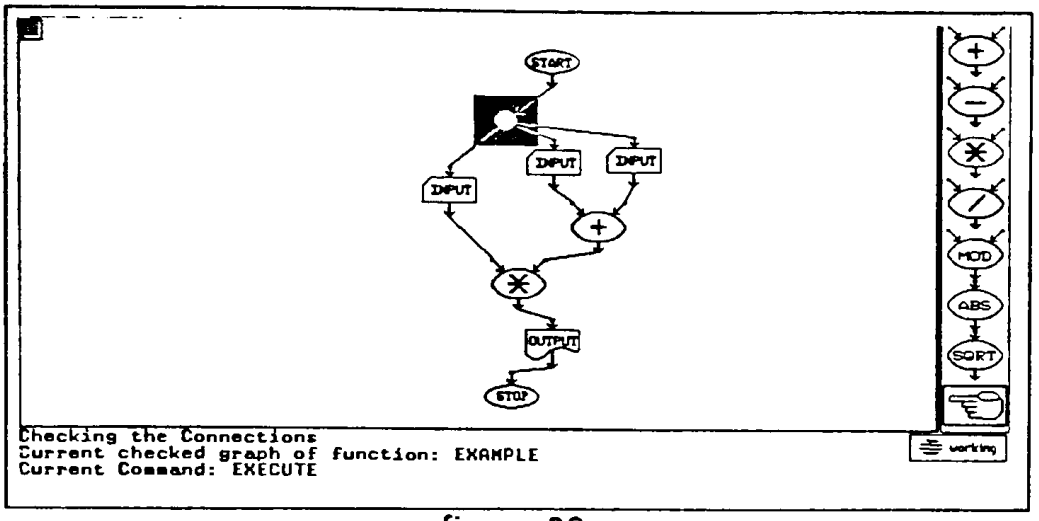


figure 39

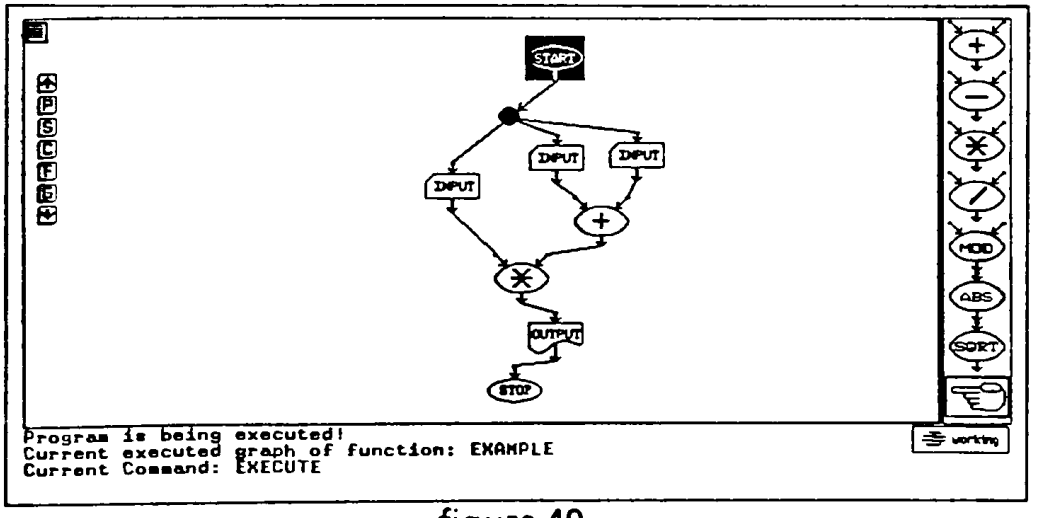


figure 40

control options are displayed on the left-hand side of the screen.

In figure 41, 42, and 43, the simulator is asking for the input values and using the

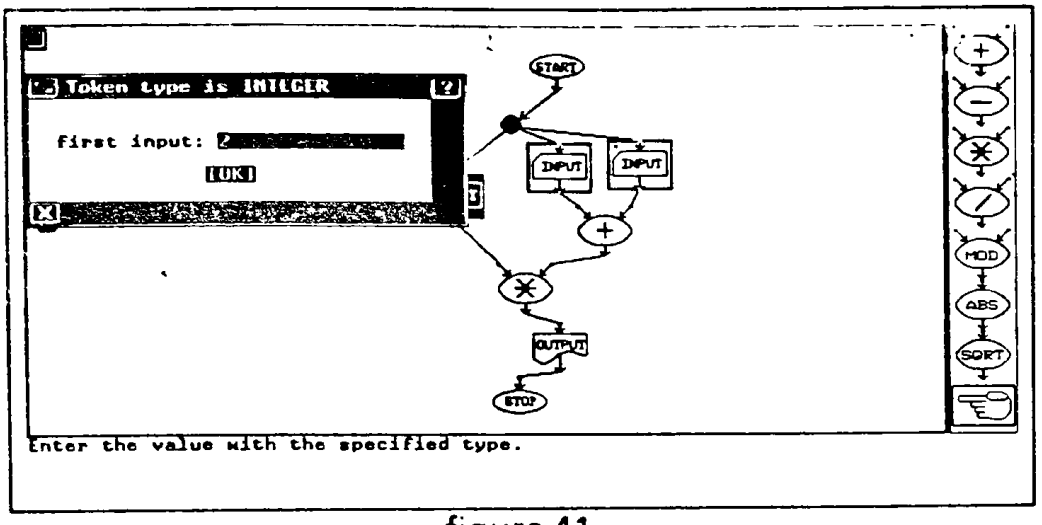


figure 41

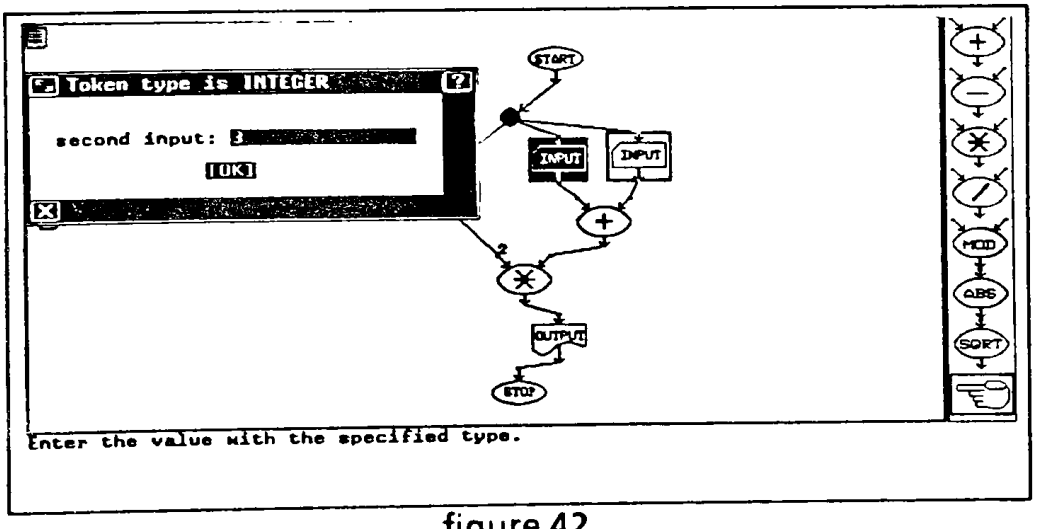


figure 42

prompt strings as the prompts. In figure 44, a result is displayed in an output window.

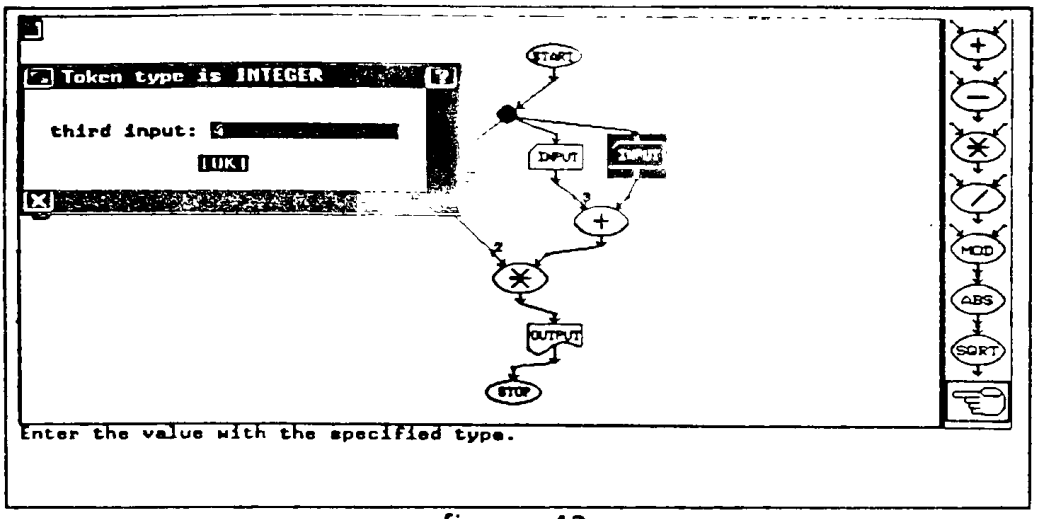


figure 43

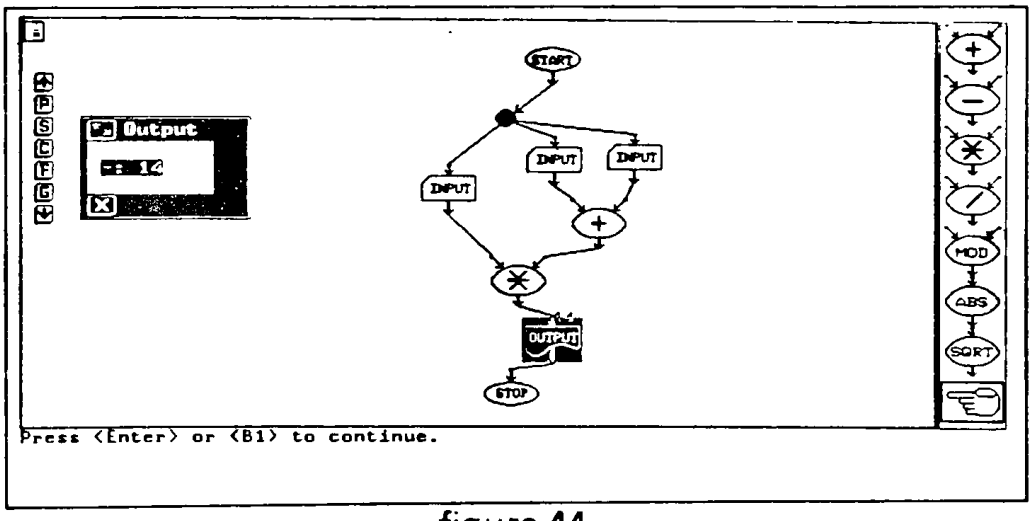


figure 44

Maybe you have found that nothing except the answer is shown in the window. In order to make the output more meaningful, you can use the "property" command to change the prompt string for the OUTPUT node. Select the OUTPUT node then choose the "property" command (see figure 45). In figure 46, you can see that a

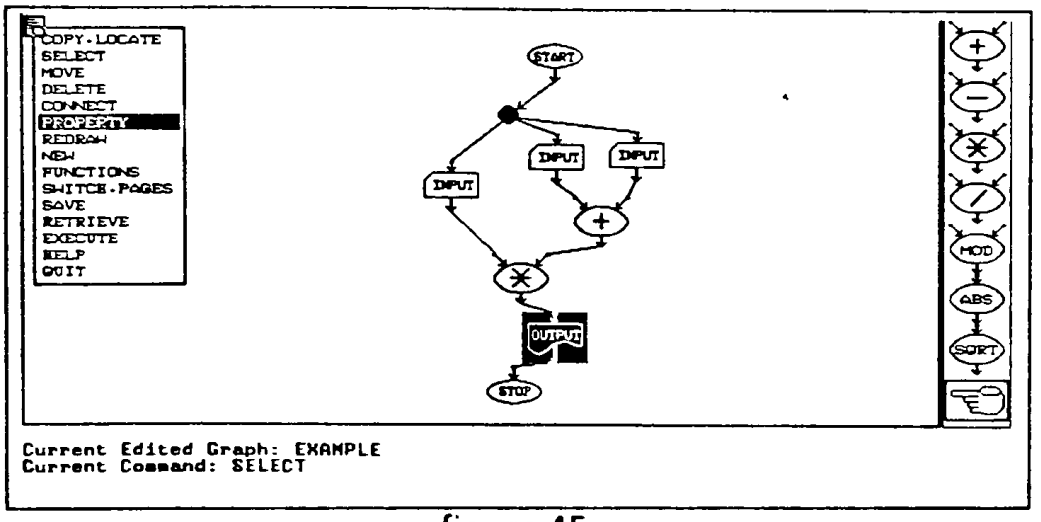


figure 45

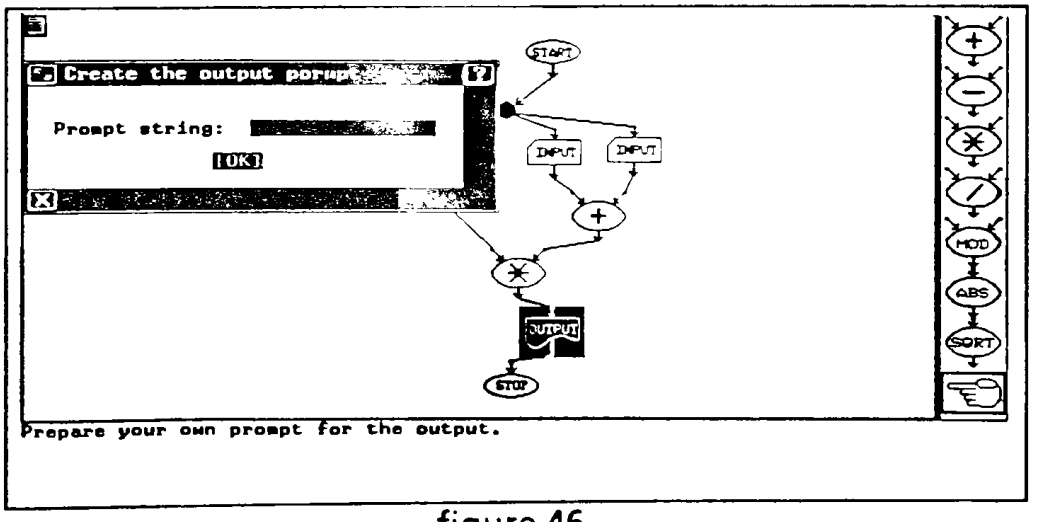


figure 46

window is opened and asks for the output prompt string.

Give a prompt string and choose the "execute" command to run the program again, you will find the final output result has the prompt you just gave (see figure 47 and

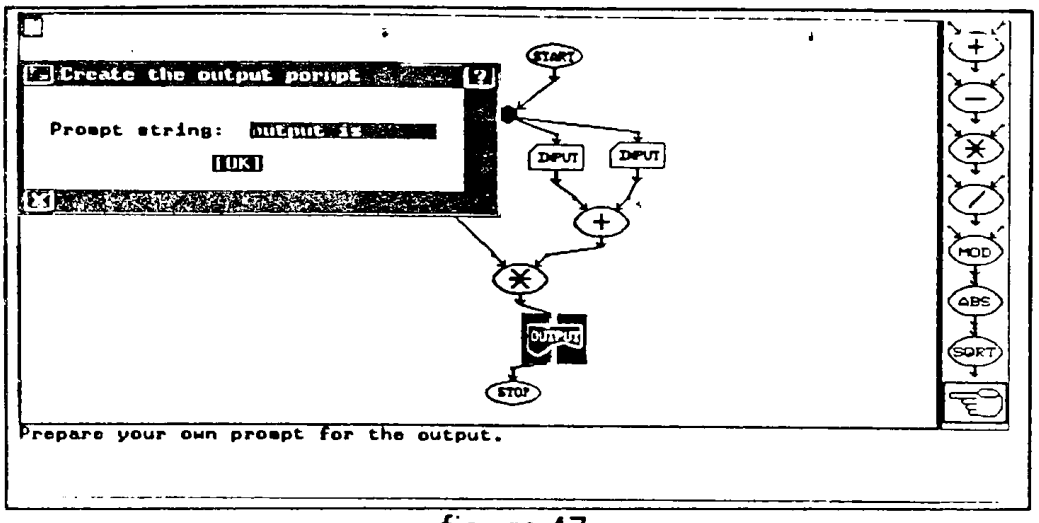


figure 47

48).

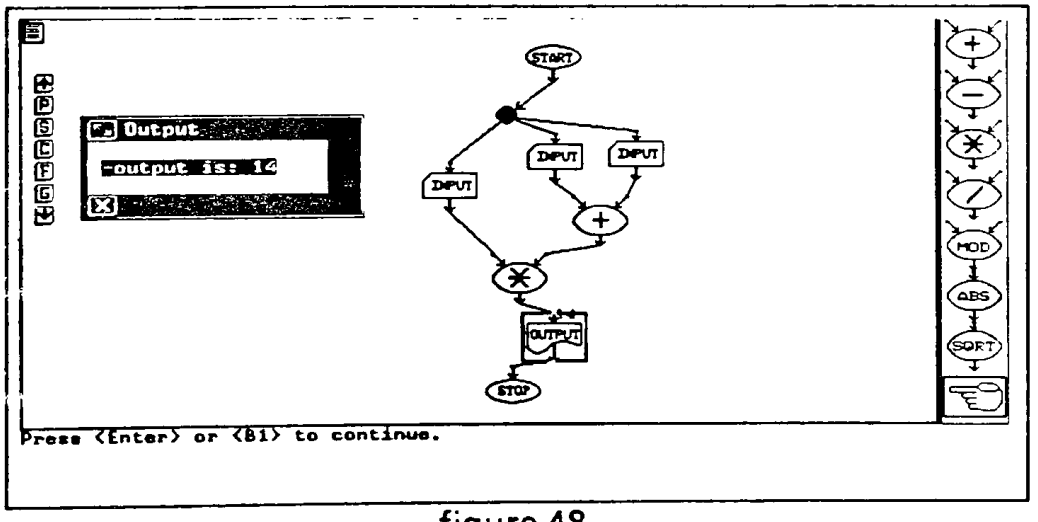


figure 48

Some commands that we have not tried. Choose the "functions" command, in figure 49, you can see a new window is opened and displayed with three options. You can use the "functions" command to add a new graph, delete an existing graph, or rename an existing graph.

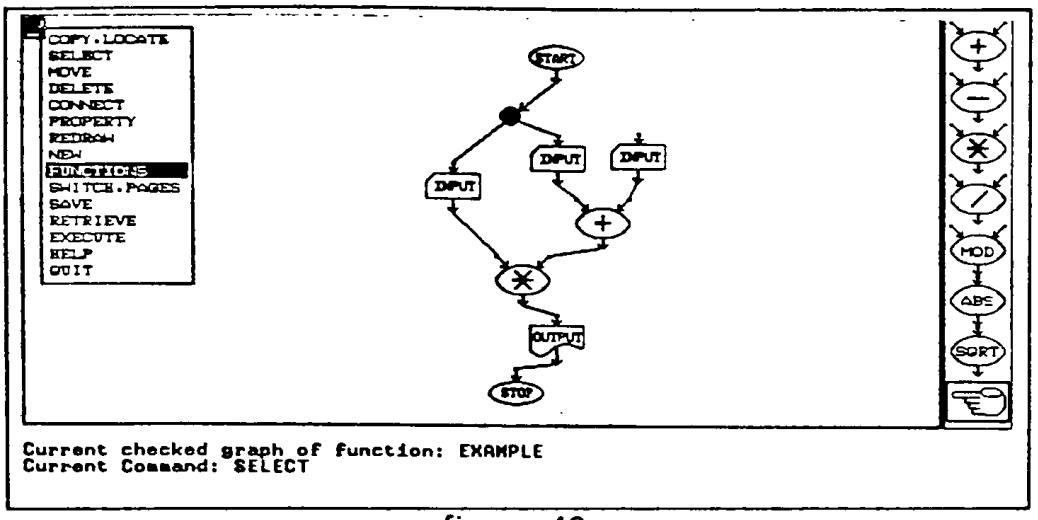


figure 49

Go through figure 50 and 51. The screen is cleared for the new graph. The current

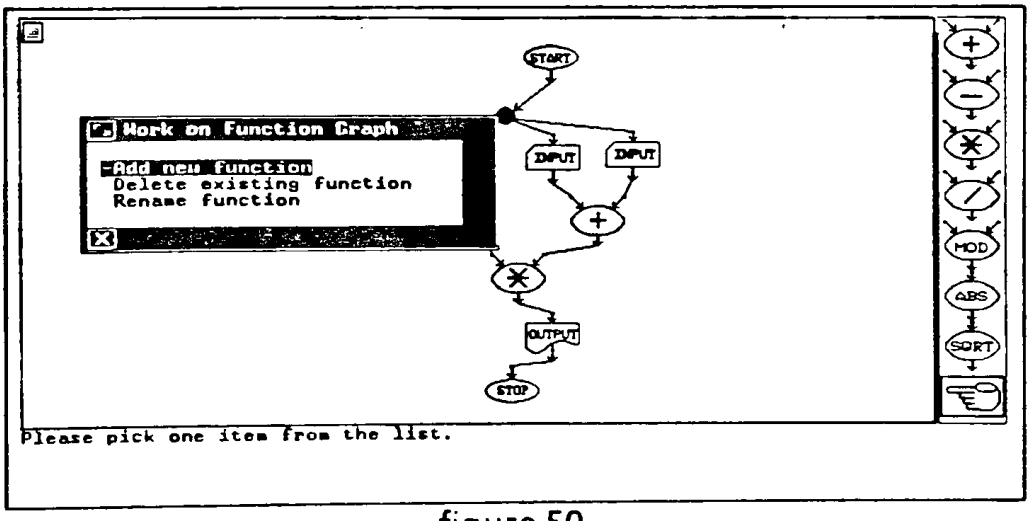


figure 50

edited graph name and the current default command are shown on the system message board (see figure 52). Choose the "functions" command and select the "rename" option, a window displayed with all the names of the graphs contained in the current program is shown. Click the "newgraph" you just added, and give a "newname".

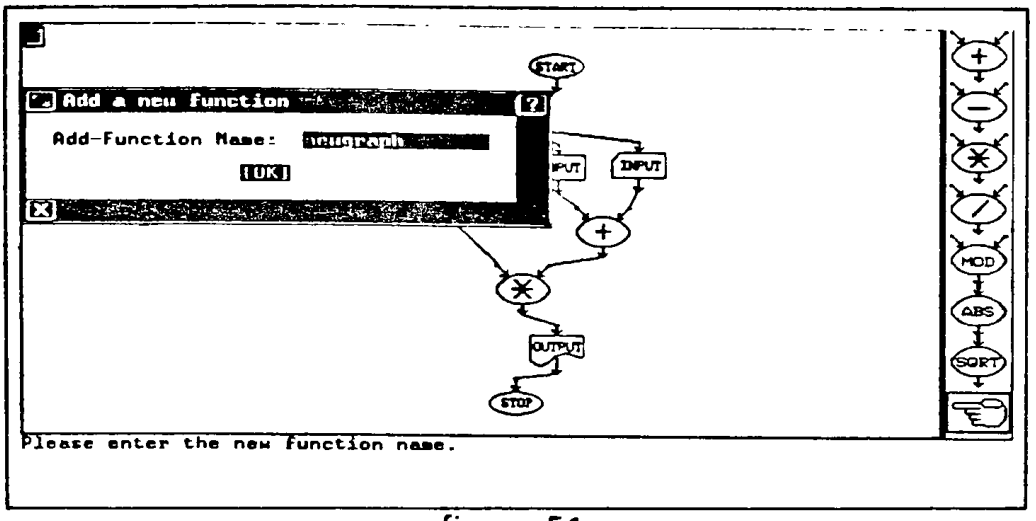


figure 51

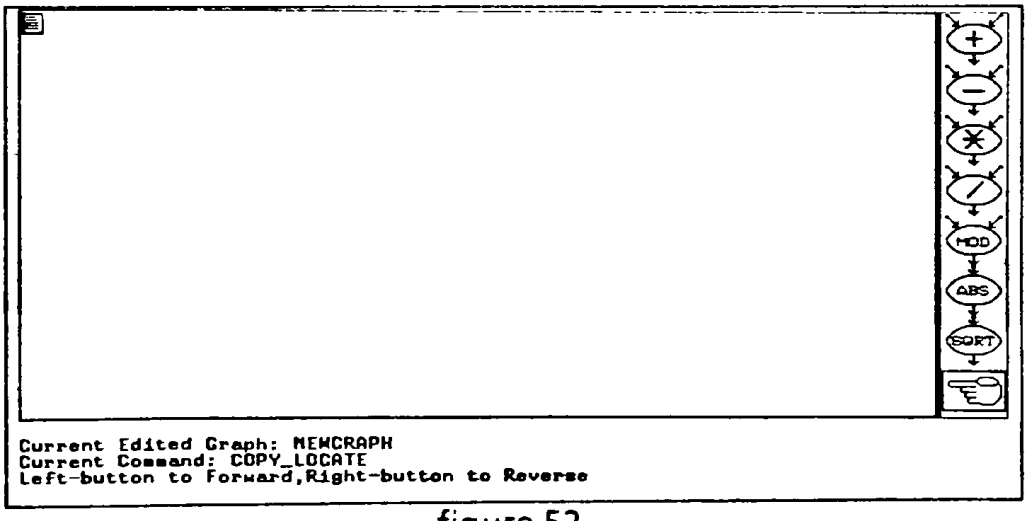


figure 52

Let's try the "switch_pages" command. Choose the "switch_pages" command pick the name "example". The graph named as "example" will be refreshed on the screen. Choose the "functions" command and pick the "delete"option from the function options, then pick the "newname". The graph named as "newname" is deleted from the program. Choose the "switch_pages" command, you can see there is only one graph contained in the current edited program. Compare what you see on the screen with the figure 53, 54, 55, 56, and 57.

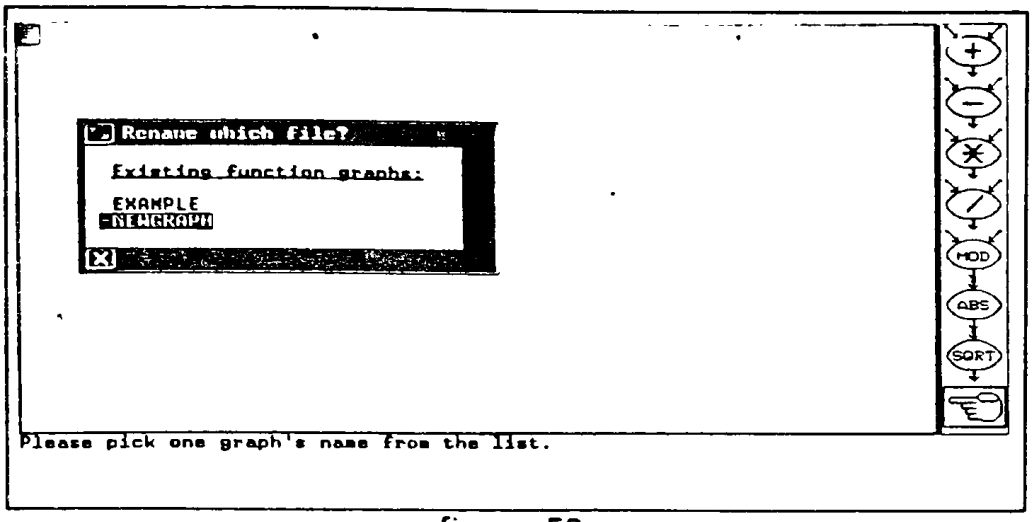


figure 53

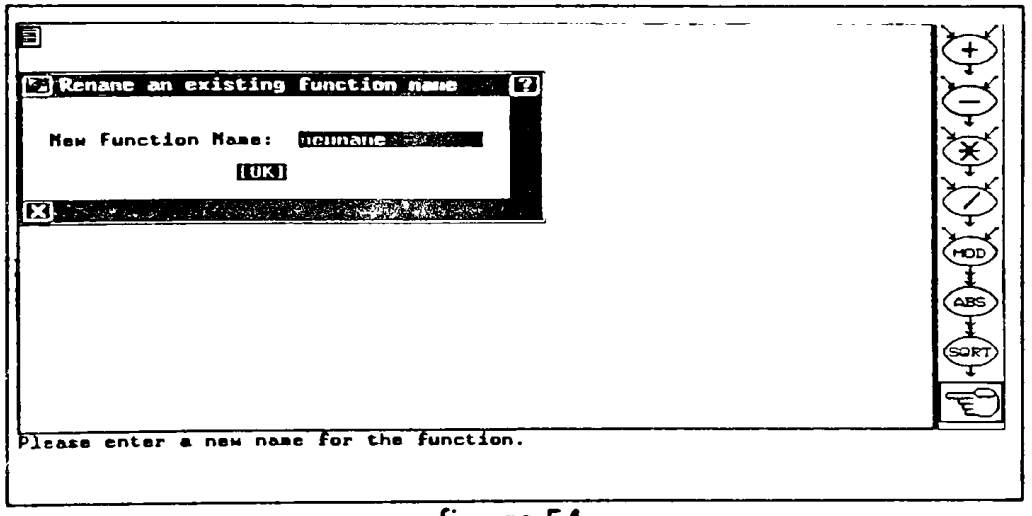


figure 54

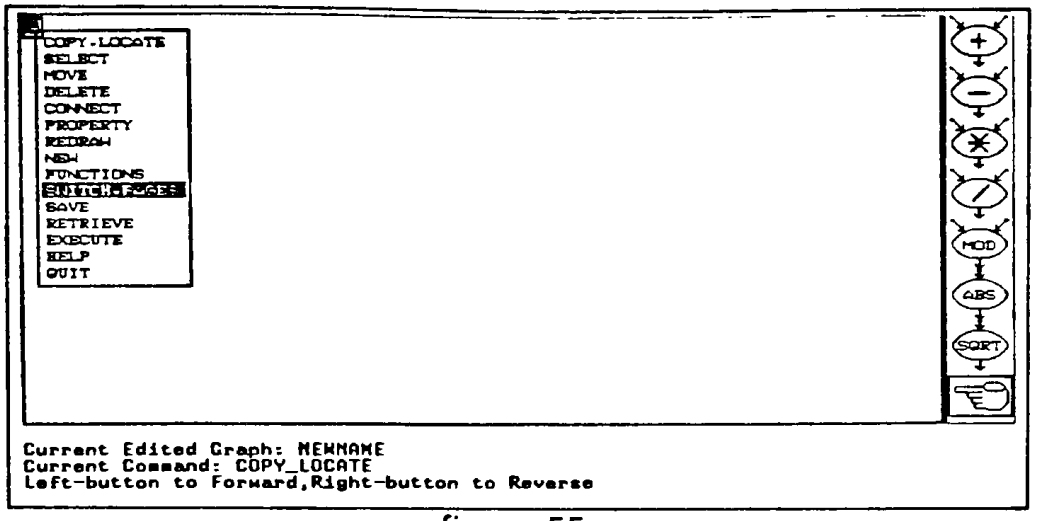


figure 55

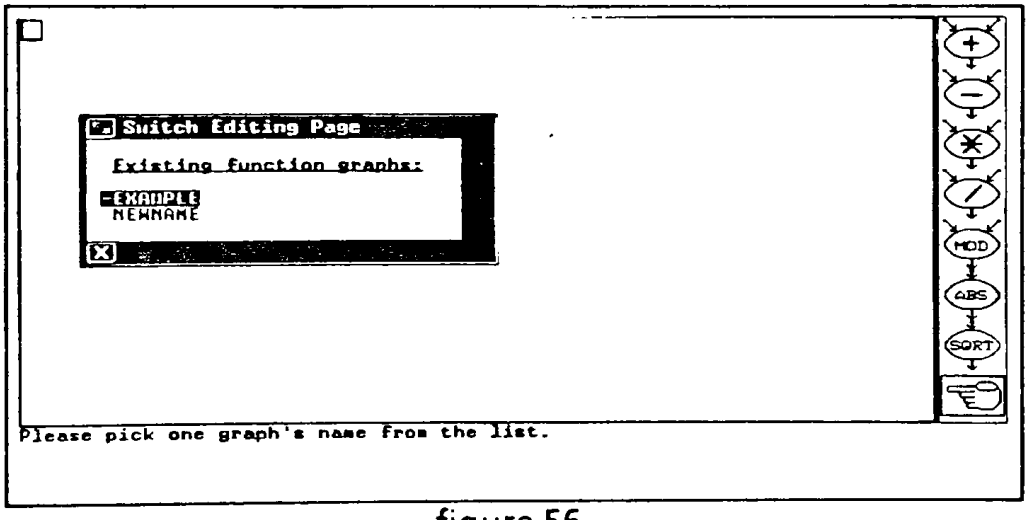


figure 56

You have tried all the commands provided in the graphical programming system except the on-line help. The on-line help function supplies brief explanation to all the operator displayed on the menu panel and all the commands you can get from the command-request-box except "quit" command. If you need to review the help text for specific operator or command, just select the operator or command then choose "help" command. A help text window will be opened with some helpful text, and will be erased after you finish reading.

Try to choose the "quit" command twice to get out of the graphical editor. We can discover some other functions supported by the system.

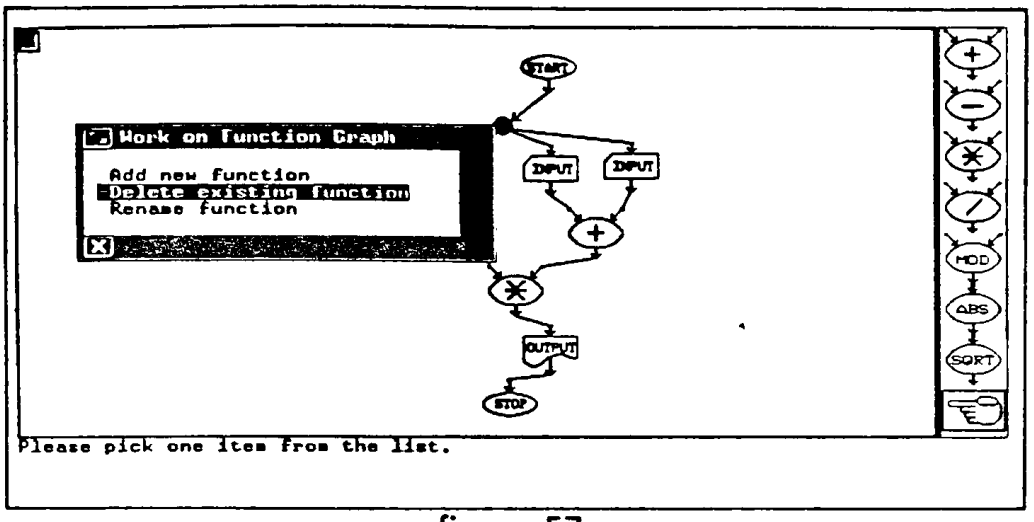


figure 57

In figure 58, we choose the second option which is to edit operator icons. A 48 by 37

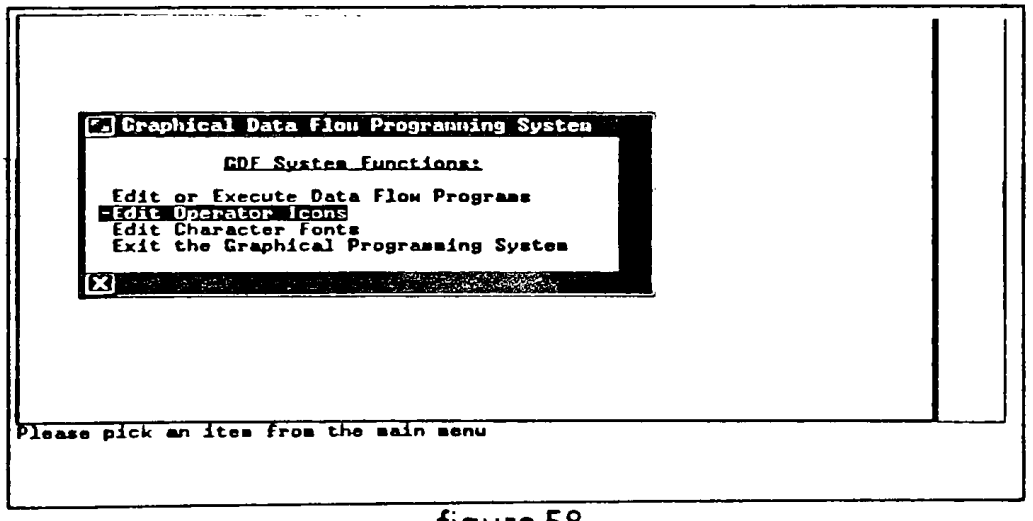


figure 58

grid is presented on the screen. Pick one operator from the list, you can see the system loads the bitmap information for the operator you picked and displays it in large scaled on the grid. Then you can turn ON and OFF for each cell on the grid by clicking <B1> on the cell. When you finish the modifications, click <B3>, the new bitmap is restored into the opicon file and used for the later work. See figure 59, 60, and 61 for the above operations.

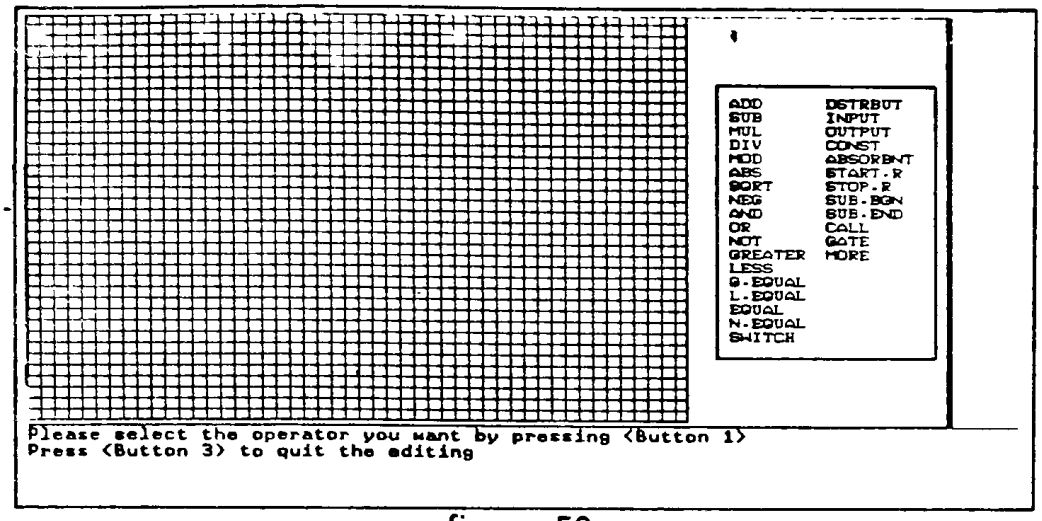


figure 59

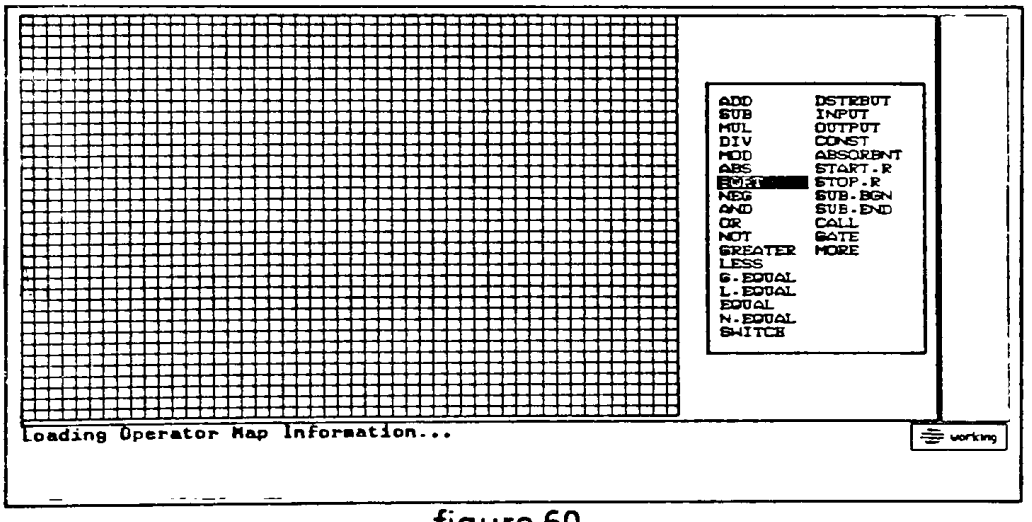


figure 60

Choose the third option from the main function list, which is to edit character fonts. Similar to the way of editing operator icons, you can pick the character you want to modify from the character list, then click <B1> to turn ON and OFF on the cells in the 8 by 8 grid. Click <B3> at the end, then the modification is restored to the font file for later use. See figure 62, 63, 64, and 65.

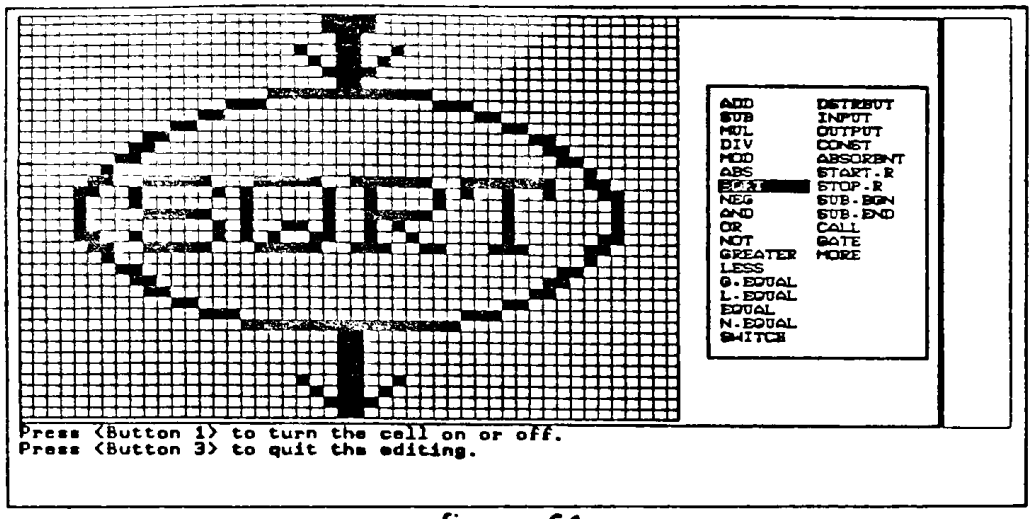


figure 61

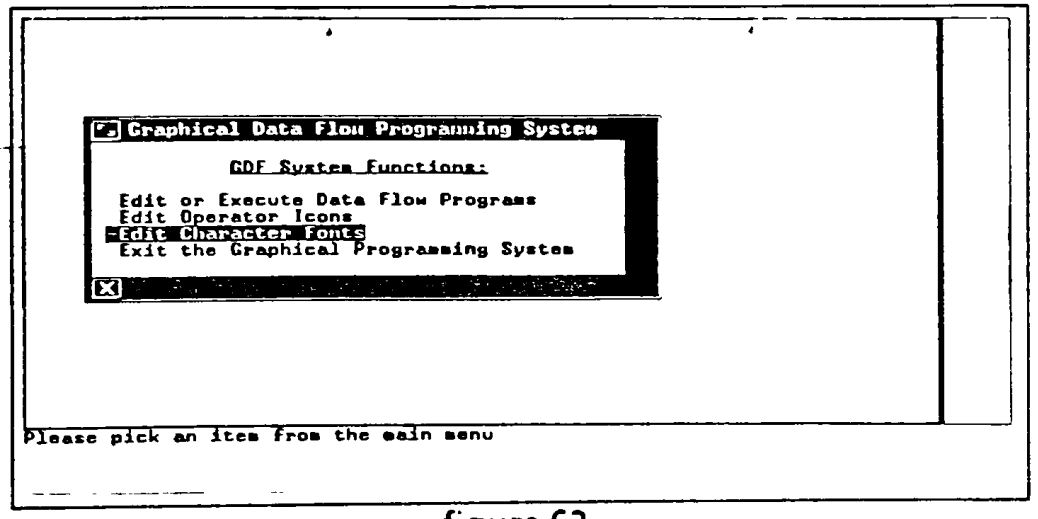


figure 62

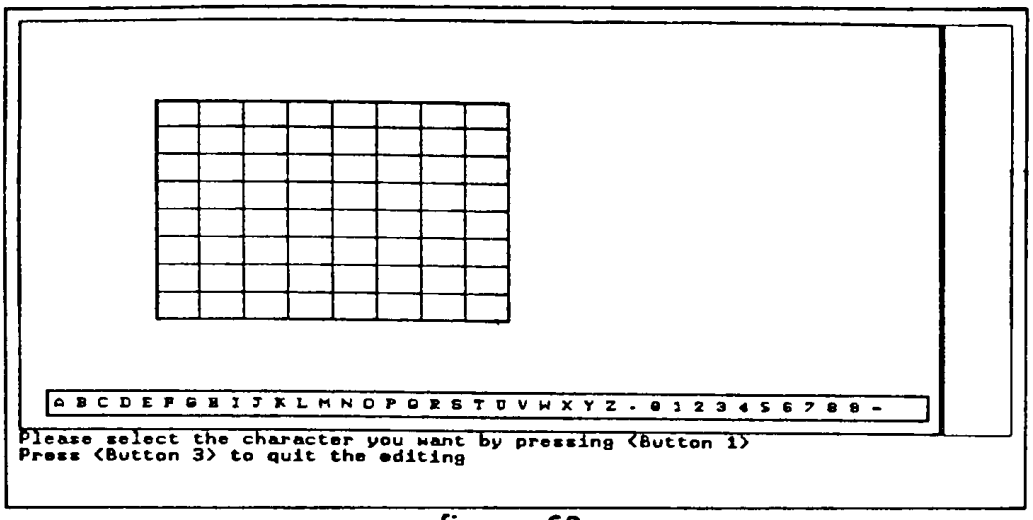


figure 63

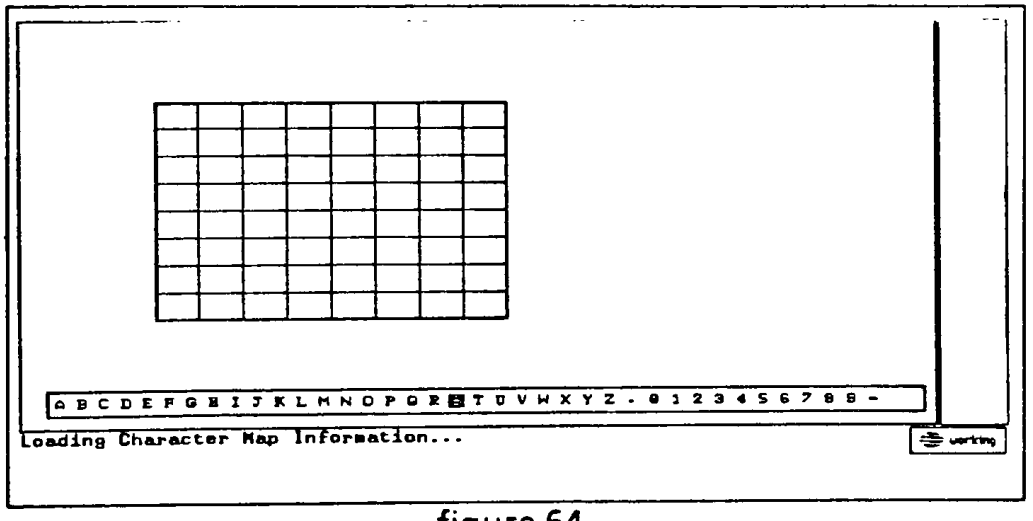


figure 64

Note: The restoration of operator icons and character fonts is not allowed in the current release. Because improper modifications may cause the system running weird or making confusion. A user must understand how the system was implemented before he starts to modify the bitmap informations.

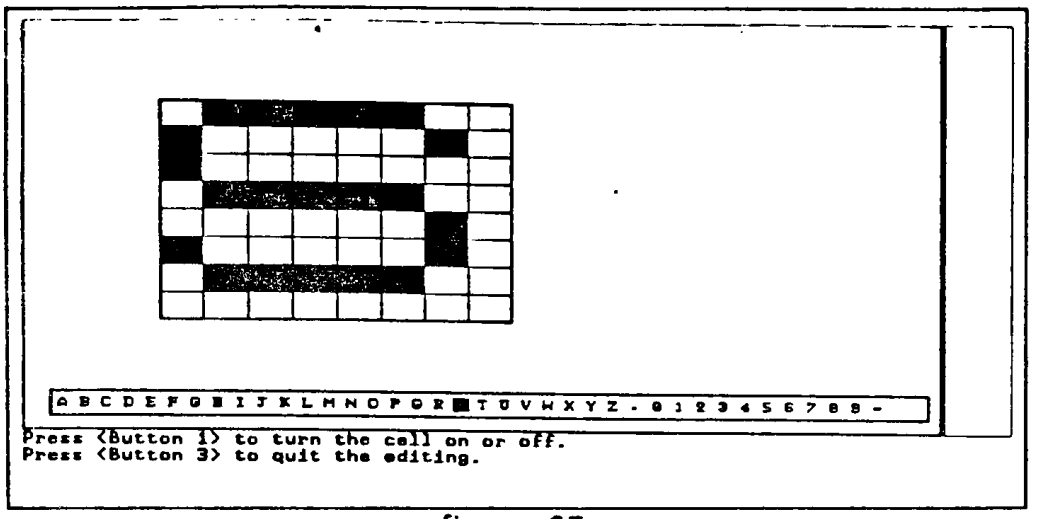


figure 65

Choose the forth option from the main function list to exit the graphical programming system.

You have already had the experience in using the graphical programming system. For the later work, you should practice how to draw dataflow graphs to carry out what you need. It is not easy, but it is the main idea for using such a tool.

6. Graphical Editor Commands

In this section, a more detail description about the system commands is provided. The commands include *copy_locate*, *select*, *move*, *delete*, *connect*, *property*, *redraw*, *new*, *functions*, *switch_pages*, *save*, *retrieve*, *execute*, *help*, and *quit*.

copy_locate

When this command is issued, the user can copy one of the operators which are shown on the current menu panel, and locate it on the scratchpad. The menu panel template can be changed if the user points the mouse cursor to the "more" icon and clicks <B1> or <B3>. Clicking <B1> causes the next template to be displayed; and clicking <B3> causes the previous template to be displayed again. When an operator is selected, the mouse cursor is changed to a "carrying" icon, and the name of the operator is printed on the system message board. The "copy_locate" command is the default command when an existing graph is retrieved or the current edited graph is changed.

select

If the current command is "select", the user can point the mouse cursor to any node or any arc in the scratchpad area and click <B1> to highlight the selected node or arc. This command is the default command for most of the editing session and it must be chosen before the user can choose "move", "delete", or "property" command.

move

The "move" command enables the user to move any of the operators in the scratchpad. The operator must be selected before it can be moved. When the "move" command effects, the cursor's shape is changed to a "moving" icon with a bright highlighted square. Select a new place and click mouse to locate the moved operator. All the arcs which have been connected to the operator will be erased and redrawn according to the new location of the operator. The chosen command will be set to "select" after the move operation is completed.

delete

A user can use the "delete" command to delete any of the nodes or arcs in the scratchpad. The node or the arc which is to be deleted must have been selected and highlighted in advanced. If a node is deleted, all the arcs which have been connected to it will be erased automatically. The deleted object can not be recovered. The current chosen command will be set to "select" after the deletion is completed.

connect

The "connect" command enables the user to construct the arcs of a dataflow graph. After the "connect" command is selected, a user can connect as many arcs as he wants; each arc is formed by connecting an output port of a node to an input port of a node. The node with the output port should be selected first, then the node with the input port can be selected.

property

This command is useful in checking or changing the property for INPUT node, OUTPUT node, CONST node, and CALL node. The property of an INPUT node is the type of the token that the INPUT node will generate and the prompt string that will be used when the INPUT node is executed. The property of an OUTPUT node is the prompt string that will be used when the OUTPUT node is executed. The property of a CONST node is the type and value of the token that will be generated when the CONST node is executed. The property of the CALL node is the name of the function graph which is to be called when the CALL node is executed. One of the INPUT, OUTPUT, CONST, and CALL nodes should be selected before the "property" command is invoked. Nothing will happen if the user tries to apply this command to the other operators, the current chosen command will be set to "select" after the property command is completed.

redraw

This command is used to redraw the screen. The reason for providing this command is because that improper mouse operations might cause some garbage or unexpected objects left on the screen. When the "redraw" command is invoked, the screen is cleared to blank, then the actual nodes and arcs information contained in the current edited graph are refreshed on the screen, the user can

use this command to make sure how many nodes and arcs have been constructed in the current graph exactly.

new

This command can be used when the user wants to start to draw a new dataflow program. The user will be prompted to enter a name for the new program. This name is also used as the name of the main function graph name. A program can have several sub-functions or subroutines which will be called by main function or other subroutines.

functions

This command is used when a program is currently edited. It allows the user to add new function graph into the current edited program, so the user can draw as many new function graphs as he wants. The system will request the user to give a name to the new graph, and will clear the screen, and set the current chosen command to "copy_locate". The "functions" command also allows a user to delete an existing function graph from the program, or rename an existing function graph of the program.

switch_pages

If the current edited program contains more than one function graph, this command allows the user to switch the current edited graph among all the graphs. When the "switch_pages" command is invoked, the system will open a temporary window, displaying the names of all the graphs contained in the current edited program, and waiting for the user to pick one of them by clicking the mouse on the name. If the command is canceled, current edited graph will remain the original one. If a new function graph name is selected, the current edited graph on the screen will be replaced by the new graph.

save

This command enables a user to save a program which may consist of any number of function graphs. The user can save the program after editing it, or before leaving the GDF programming system, or any time he wants to save the program. The name of the main graph is used as the default name of the program. The user can use the default name or create a new name for the program. The user also has the options to choose either quitting the current session or resuming after the "save" operation is completed.

retrieve

The "retrieve" command allows the user to retrieve a GDF language program form the current directory. The main graph of the program is then displayed on the screen. If the user wants to retrieve an non-existing program, proper error message will be given.

execute

This command can be issued when the user wants to execute the current edited program. If the program is a new edited one, or an old one but has not been executed since last modification, the dataflow simulator will check the arc connections for all the graphs before the program is executed. Seven execution simulation options are provided.

help

The "help" command is provided to invoke the on-line help function. A brief explanation for the previous chosen event will be displayed in a newly created temporary window. The previous chosen event might be an operator or a command that the user just selected before the "help" command.

quit

Issuing the "quit" command will stop the current editing session. If there is a program in editing, the system will prompt a confirmation message to the user. The user may cancel the "quit" command, save the program then try quitting again, or can just quit without concerning the modification or saving program.

7. *Error Messages and Warning*

The system messages provided in the graphical programming system include error message which will be given when an error is encountered during the execution of a program, and warning which is given while a program graph is edited. These messages are listed in this section.

Warning

-- error in display output function

-- location is not appropriate

-- illegal action, use a GATE or try re-connecting

-- error - first point must be an OUTput port

-- must be an INput port, connection fails

-- function name duplicated, please try another

-- function name duplicated, please try some other names

-- save the current graph before retrieving other graph

-- open failure, or file non-existing

-- program execution terminated

-- graph connections are not completed

-- line segment already exists

-- error in searching input connecting port

-- error in searching output connecting port

-- error in searching port

-- input string contains invalid characters

-- program is aborted as you requested

-- program is terminated because of an error

-- type error in CONST operator

-- error in finding token display location

Error messages

memory overflow! system will be aborted

-- program or function is not ended with STOP or END

-- maybe the STOP or END is not positioned properly

-- only one START or BEGIN is allowed

-- a function graph needs a BEGIN or START node

-- type error in ADD operator

-- type error in SUB operator

-- type error in MUL operator

-- type error in DIV operator

-- type error in MOD operator

type error in ABS operator

-- type error in NEG operator

-- type error in SQRT operator

-- type error in AND operator

-- type error in OR operator

-- type error in NOT operator

-- type error in GREATER operator

type error in LESS operator

-- type error in G__EQUAL operator

-- type error in L__EQUAL operator

-- type error in EQUAL operator

-- type error in N__EQUAL operator

-- error in finding a CALL node

-- error in function name, or function non-existing

-- error in finding the beginning of a function

-- error - divided by zero

8. Special Notes ...

At the end of this manual, several things which are worth for reminding. Remember to save your program graphs after you have made changes on any of your graphs. The changes will be discarded without performing the "save" operation. Make sure to copy your dataflow programs back to floppy disk after you leave the graphical programming system. Any program left in the hard disk drive might be deleted and cannot get recovery.

The functions of the graphical programming system have been tested for most of the possible circumstance. If you have any questions, or encounter any strange response or errors to cause the system to be aborted or core dump, please make a copy of your program which caused the system error, put down the operation procedures you did, and prepare a short but detailed descriptions about the problems. Mail or contact the following persons:

Professor Alan R. Kaminsky, or Professor Donald L. Kreher
Computer Science Graduate Department
Rochester Institute of Technology
Rochester, NY 14623

or the author: Roland Jehng
84 Country Corner Lane
Fairport, NY 14450

-- THE END --