PROGRAMMING ENVIRONMENTS

by

J. A. N. Lee*
Professor of Computer Science
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

Technical Report - CS83027

*The work reported here was supported by the IBM Santa Teresa Laboratory, San Jose, CA.

Abstract

This report examines in detail the contents of the environments which are established during the progression of design, check-out and execution of a program. The level of support given by vendors to the elements of these environments is then examined with the conclusion as to where additional effort needs to be expended to achieve some improvements in program development support.

INTRODUCTION

An examination of the trends in programming languages today or an attempt to either influence or affect those trends cannot be completed without consideration of the environments in which those languages exist. In fact, I would contend that there are no trends in programming languages which are not first influenced by trends in programming. As it were, the worm has turned. In the early days of programming what one could or could not do was directly a function of the available programming language and, in consonance with the "Whorfian Hypothesis", programming thought processes (methodologies) were strictly bounded by those languages. The programmer saw the programming environment through the tinted glass of programming languages. Added to this, the programming languages systems were "stand-alone" and had total control of the computer. Thus the programming environment was an extremely sparse environment, the pre-operations being off-line, and the post-operations depending on the crumbs left over in the cupboard otherwise known as memory and identified as the "octal dump".

Gradually, starting with the advent of supervisory systems and monitors (more attuned to improving through-put and to protecting the system from the user than improving his lot) the programming language environment has been fleshed out by the addition of new features. For example, most of these were initially intended for independent usage and only peripherally have had any direct connection with the programming languages. Thus we have developed sets of disjoint processes each well-meaning in their intentions but lacking coordination and being almost impervious to changes in companion (supporting) systems. Thus we find that we have:

(1) programming languages that have not changed radically since 1960,

(2) "support" systems which were originally intended for off-line usage,

(3) non-standard system interfaces even when different manifestations of the same elements reside in a common environment, such as the relationships between debuggers and programming language processors, and

(4) widely differing representations of common data elements.

Before attempting to answer any questions about the individual elements of these programming language environments, it is necessary to examine the components of these environments and to attempt to identify common elements, information flows and interdependencies.

THE PROBLEM SOLVING ENVIRONMENT

The typical problem solving environment espoused by many teachers includes five to seven subprocesses each of which contain several tasks. For the purposes of discussion here, we want to restrict ourselves to those problem solving environments that include the intention of using a computational system and expressing solution processes by means of a programming language.

Following the problem solving task of "developing a solution", there exist three subsequent tasks which we can express as:

* implement the solution
* verify the implementation (against sample data)
* run the implementation (against the actual data)

These three tasks are performed in three apparently differing environments though, as will be seen, there exist commonalities between the subprocesses in each environment. We shall name the three environments as follows:

*Program Design Environment
- containing program creation, generation (as a load module) and (because of the cyclic nature of the problem solving environment) updating processes.

*Program Check-out Environment
- the anticipated environment of actual problem solution with the addition of a conglomeration of testing devices.

*Run-time Environment
- the actual environment of problem solution in the wider environment of the user.

These environments are each connected by a decision mechanism which permits the cycling of the solution space onto itself or to any of the other environments, and by an inheritance process by which elements generated in one environment can be passed to a succeeding environment. Of course, the programming methodology goals, which form a super environment to these three environments, would expect a perfectly sequential activation of these three environments without any cyclic activity.

We can further divide these environments into three concentric components:

- the activities supported in each environment and their associated processors,
- the data elements applicable to these activities and,
- the data flow between these data elements (and the commonality of data).

COMMON ENVIRONMENTAL ACTIVITY CLASSES

The keystone of each of the environments is the execution of a process in the sequence of the environments starting with that containing the language compiler, and continuing through those containing the prototype programs through to the production program. This process of execution is no longer initiated by the user providing a "one-card loader" and putting the library of run-time routines at the back of the source deck, nor is it dependent on the user making sure that the punch is turned on and the correct tape is mounted on a magnetic tape device. Rather, supporting processors are selected and installed through a link/loading activity, the processors being provided by an archival system and a set of libraries. The configuration of the environment is managed in two modes - by attaching (accessing) the appropriate archives and libraries as well as physical resources and then providing the appropriate communicators to those facilities.

These three activities, execution, link/loading and configuration (embedded in which is the necessary communication between the elements of the configuration) constitute the fundamental activities of an operating system and are common to every environment.

The processes of design, check-out and execution are supported in the first place by documentation concerned with the processes being run. At the design level the documentation may be language manuals while in the run-time environment the user needs program operation instructions. At each level the user (of that environment) may need information on such items as error message interpretation or data preparation. Let us name this activity the "help" process.

Four other activities can be grouped together under the general classification "debug", though it would be preferable to find another term for this class. That is, while all the activities have a debugging usage, their total scope is somewhat wider. Data

display for example, is a very important debugging facility, but display also is applicable to deferred output operations as well as query systems. Similarly, data manipulation is essential to debugging if stored data is to be modified at a break point to enforce a peculiar condition, but data manipulation is also an essential part of program creation as an editing facility. Analysis of the state of a system including the hardware facilities to recognize error conditions must exist during both check-out and run phases in order to protect the user. During design, however, the analysis of the program can show faulty implementation even before the check-out phase is entered. Verification is a process which is generally thought of as being restricted to the design and check-out environments. However, the leaving of assertion statements in a program at run-time can improve program reliability and extend verification to more than simply the sample run-time environment which is modelled at check-out.

Let us represent these core activities as show in Figure 1. The outer segments of this figure indicate the data elements pertinent to each environment, and the surrounding space will represent the data associations.*

In summary, the common environmental activity classes are (Figure 1):

A. Operating System Support
   - resource configuration and communication
   - linking and loading
   - initiation and execution

B. Information dissemination (Help)
   - documentation access
   - status interpretation

C. Debug Facilitation
   - data manipulation
   - data display
   - analysis
   - verification

----

*Note that data communication as an actual physical process must take place through the core system; this outer space is intended to be used to show the data flow between the processors which create it and those which utilize it.

Other groupings of activities can be formulated under these headings but these clearly support the problems identified here. For example, it would be possible to combine information dissemination with data display since these two activities perform similar tasks. However, this separation is maintained so as to clearly show the different styles of support provided at each environmental level.

In the next section we shall review each of these activities in the different environments.

DETAILED EXAMINATION OF EACH ENVIRONMENT

Each of the activities mentioned above can be viewed as the uniformly defined operations in the core of each environment. The choice of the term "uniformly defined" is explicit so as to indicate that the types of activities are similar though in each environment the objectives may be different and thus each implementation may vary slightly.

In some systems, such as JOSS and APL, these activities are in fact constant and common throughout the three environments. In these cases each environment is derived as an adaptation of the prior environment, thus showing a continuum in which the data is updated but never discarded. In other cases, such as a PL/I or COBOL system, the environments represent discrete, disjoint steps. For the purposes of this discussion we shall use the discrete model though the modification to a continuous model will be obvious.

PROGRAM DESIGN ENVIRONMENT

Figures 2(a-d) show the data elements in the environment of Program Design (or Generation). Realizing that the purpose of the activities in this environment is to develop a program that is the implementation of a previously developed solution, there is an external activity (the thinking task of the programmer) to modify the specifications into a program. Although both specifications and program text occur in this diagram, there is no processor in the environment which performs this transformation. Only the tools to assist in that transformation exist. Thus the two major inputs into the environments are the program text (being edited into the suitable machine readable form) and the specifications (as an element of the verification activity). The program text

can be displayed at any time and eventually is transmitted to the language processor for compilation and to the help segment for documentation purposes. The execution of the processor then returns the program listing to the display activity (see figure 2a).

The activity of compilation (figure 2b) is dependent on a number of factors which, depending on the particular environment, may be user specified or automatically supplied. For example, the program text may be tagged with some programmer supplied attributes which indicate which processor is to be used (COBOL, PL/I, etc.) and INCLUDE statements may specify other library elements which are to be added to the text. The selection of a particular processor will in turn make system requests which are transmitted to the programmer. Further the programmer may specify, through a command language, or programming languages pragmas special activities to be undertaken by the compiler; optimization, preparation for debugging, etc.

Output from the language processor goes in several different directions, some of which are maintained in this environment and others which are passed to the check-out environment. The fundamental output is the object code which is generated and this is transmitted to the succeeding environment. For the purposes of facilitating the debugging activities at the check-out level and assuming the (obvious) need to relate debug information to the source code (program text), the processor also passes the symbol table to the next environment. Program listings are returned to the design environment and are retained for use in the check-out environment. Assuming that such listings also contain error messages, the help activity uses the data dictionaries from the library system to interpret those error messages (see figure 2c).

Most modern compilers perform optimization over the generated code by the use of analyses of the logical flow and the data flow of the program. These analyses are rarely made available to the remainder of the design environment but they can represent a very valuable tool and should not be lost (see figure 2d).

The details of the actual flow of data between activities is shown in figures 3. Assuming that there exists some operating environment containing the archives and library of processors, then the two other inputs into the environment are the problem/solution specifications and the program text developed therefrom. Every other data path in the communications environment consists of either copying or accessing data between activities. These data flows can easily be classified into two groups - those that take place prior to compilation (figure 3a) and those which result from the compilation (figure 3b). In the

class of pre-compilation data movements is the copying of data from the editing activity (the preparation of the program text) to the compilation activity. After compilation, one typical data movement within the design environment is that required to display the program listing and associated information while the generated code from the compilation is inherited by the check-out environment.

In this model we are assuming that an environment has a latent storage capability (besides those specified in libraries and archives) which constitutes a basic residue for the initiation of the next phase. This concept enhances our vision of the various environments as being continuous rather than as discrete elements.

It is possible that the next phase is not the check-out phase shown in the logical ordering here, but is in fact the run-time environment. However the residues are similar.

CHECK-OUT ENVIRONMENT

Figures 4(a-d) show the intermediate environment which we have termed the "check-out" environment. Several elements of the data set of this environment are or can be inherited from the preliminary environment which contains the process of compilation. These elements are (* indicating "can be" inherited rather than "must be"):

- generated object code (program)
- program text and listing
- specifications
- symbol table*
- flow analyses*
- documentation

In the same manner as the execution of a language compiler is preceded by the establishment of communications with its required resources and supporting modules, so the analogous activity must take place to initiate the execution of the object program. In this environment (figure 4a) the scrutiny under which the program executes may be much more severe, though it is not clear whether this scrutiny must be anticipated at the design level. Most current check-out environments are defined at compile-time, the object code being explicitly modified to permit the debugging activities and unmodified by optimization that may blur the effects of individual statements at the high order language level. Thus a program can be executed in either the free running mode or with the expectation of having the run suspended as the result of pre-established breakpoints or by manual interrupts. Whether the

establishment of breakpoints can be achieved at the check-out level is highly dependent on the support services that are provided. Ideally it would be appropriate to be able to set breakpoints at the check-out time at any point in the program and with reference to the source text of the program. This requires extensive linkages between the object code, the program text (or line numbered listing) and the symbol table. Similarly it would be ideal to be able to specify which identifiers were critical and were to be subject to monitoring in the check-out environment.

These facilities imply a patchwork of interconnections between the object code, the program text/listing, the symbol table and the activity drivers. For example, the monitoring of a particular variable may require an interface between the executor and the data display activity, while the invocation of error detection requires the analyzer to access the symbol table, the program listing and the error interpretation information. Display of the pertinent data on error situations may also need the support of the data display facilities.

Data for testing is an element which is first introduced at this environmental level and can be classified into three categories:

- typical data
- sample data
- purposely bad data

and the sources of this data may be fourfold:

- specially prepared files
- an institutional data base
- direct entry from the debugger
- data generation programs

Let us consider the sources of data primarily. Specially prepared files are those data sets which are prepared with this particular program in mind and conforming to the specifications of the program. Data from an institutional data base does not have the "benefit" of having been prepared in the program specific form; thus it will be necessary to condition this data prior to its presentation to the program. If the environment provides an activity to process data from one format to another in a manner which is transparent to the user, then we shall term this activity "auto-conditioning".* On the other hand, if this transliteration must be undertaken by the program then we shall consider this to be part of the "specially prepared files" category. Hence "auto-conditioned" data occurs in the check-out environment or the succeeding run-time environment. Most activities at the check-out level (figure 4) are dependent on either the residue from the

----

*In OSI model this activity is implemented with the Presentation layer.

design phase or the contents of the typical environments provided as part of that phase.

Data display (figure 4c) in our very general definition, includes classical program driven output, but this form of communication, particularly during check-out, can be supplemented by what might be termed "deferred" output. That is, that information pertinent to the desired results is stored in some form that may then be inspected by the user on-line or dumped (say) to a printer or a graphics processor. This is expected in the debug activity of the check-out environment to include the inspection of the work area of the program to access critical data relevant to the generation of the results but which are part of the user's output data set. This requires the use of both the residual work area and the symbol table, so as to provide referencing at the user's level of identification.

The analysis activity (figure 4d) includes the classical error interrupt processing (augmented by interpretation through the help activity) as well as collecting program execution statistics. The latter may require the installation of pre-determined stubs during the compilation phase onto which the statistics gathering procedures can be hung.

Verification at the check-out level involves the use of the original specifications, the three data sets described above and the programmer built-in assertions. The latter (ultimately) may be merely the set of assertions that could not be resolved (confirmed) at compile-time.

Examining the data flow at the check-out level (figures 5a and b), it is interesting to note the comparative lack of complexity at this level compared to the previous (design) environment. Much of the flow is hidden; it is inherited from the environment which included the compilation activity. Three external inputs are discernible:

- any separately compiled units
- the run-time environment model
- test data

The separately compiled units may be simply collected from a library or archive in the previous level. The process of compilation generates the list of additional routines to be included and provides either that list or the actual routines to the check-out level. Provision of a typical run-time environment and set of data sets which will adequately test the program is necessary. Test data sets may be developed through automatic data set generators but the provision of a typical environment other than the current working environment is difficult to simulate.

Test data is communicated through the environment to two other activities; the data manipulator and the running program. The data manipulator stage intervenes between the entry of the test data and the program running activity in order to "massage" the data into the appropriate form. For many cases the data may be generated so as to conform to the formats and types specified by the program but in other cases the data may need to be "conditioned" to satisfy these requirements. This latter activity may be particularly prevalent when test data is provided from a data base system where data is stored in some "neutral" form.

RUN TIME ENVIRONMENT

Finally at the user's level (figures 6a-6d), that is a runtime environment, many of the same data sets exist as in the check-out level. However, the data sets now contain the actual data to be used and the test facilities now remain only to verify the correct operation of the activities. Data manipulation and data display are almost entirely concerned with conventional I/O while the help segment is chiefly concerned with assisting the user to interface with the running program rather than with interfacing the program with the system.

Remnants of the check-out environment exist to provide maintenance support and analytical facilities have been reduced to error detection based on programmer defined on-conditions and hardware interrupts.

Data flow in the user's environment (figure 7) is substantially reduced, the majority of data sets being inherited from a previous environment.

CURRENT VENDOR SUPPORT

The next three figures indicate the activities which are commonly supported by the total system. In the design environment (figure 8) the "OS" segments is generally supported (the cross-hatched segment 1) in each system which also boasts an operating system, the facilities provided being the primary purposes of those systems. Where interactive compilation is also provided, the activities of help, data manipulation and data display (the lined section 2) are immediately available to the user/programmer,

though the degree to which these tools are supported varies significantly over different systems. The facilities of validation and verification (the shaded section 3) are uncommon.

At the check-out level, the support (figure 9) is extended to cover additional activities both in the common areas of support (area 1) and those provided under interactive systems (area 2). The lack of support in the identified areas (area 3) of monitored runs, data conditioning (between (say) a data base system and the running program) and the debugging elements of assertions, traces and frequency counts, is significant to providing support to the programmer in this special intermediate environment.

In the user's environment (figure 10) the amount of support has increased significantly to the point where only two items are uncommon (sections 3); system conditioned data and assertive verification. The latter is highly dependent on the design of the program and the skills with which the programmer identifies the critical stages of the program. The former is an outstanding problem that is currently overcome by "manually" transforming the data from its storage form into that desired by the program.

CONCLUSIONS

It is very difficult today to regard the environments of programming and program usage as being "program centered". That is, we are at a stage which might be likened to the age of Gallileo where it was necessary to move from the concepts of an earth centered universe to one in which the sun was the focal point. So in program environments, we have left the "stand-alone" environment of the early 1960's and now find the program to be merely a segment of a much larger problem solving environment. The pressure today is to make the programmer more productive even at the cost of using more computer time and more system resources. Some of the facilities ultimately provided to the user of a program are dependent on the good will of the programmer (e.g. adequate documentation in a help environment) but in the same manner the programmer depends on the good auspices of the systems provider to acquire supportive tools for his task.

The fact that the area of uncommon support (shaded, area 3) in figures 8 through 10 reduces in the progression through the environments, is an indication of past efforts to support running programs. The fact that programmers are beginning to recognize their responsibilities towards the user is also improving this situation. The greatest areas of lack of support occur in the design and check-out environments.

Some of the support needed will require the addition of new facilities in the language processors, more education for programmers and lastly (but more easily) the overcoming of our tendency to throw good information away. Many of the results of activities which take place during compilation could be reported to the programmers and added to his repertoir of tools. These especially include data and logical flow analyses.

One final comment. In the same way as Gallileo's sun centered universe was superceded by the "big bang" theory, so the view of programming environments is rapidly changing. Whereas we formerly we thought of programs as the more permanent part of a data processing system, now the development of information systems views the data as having the permanence and the processes to be fleeting transitions passing over that data. This complete reversal of roles may shed further light on the concept of environments.

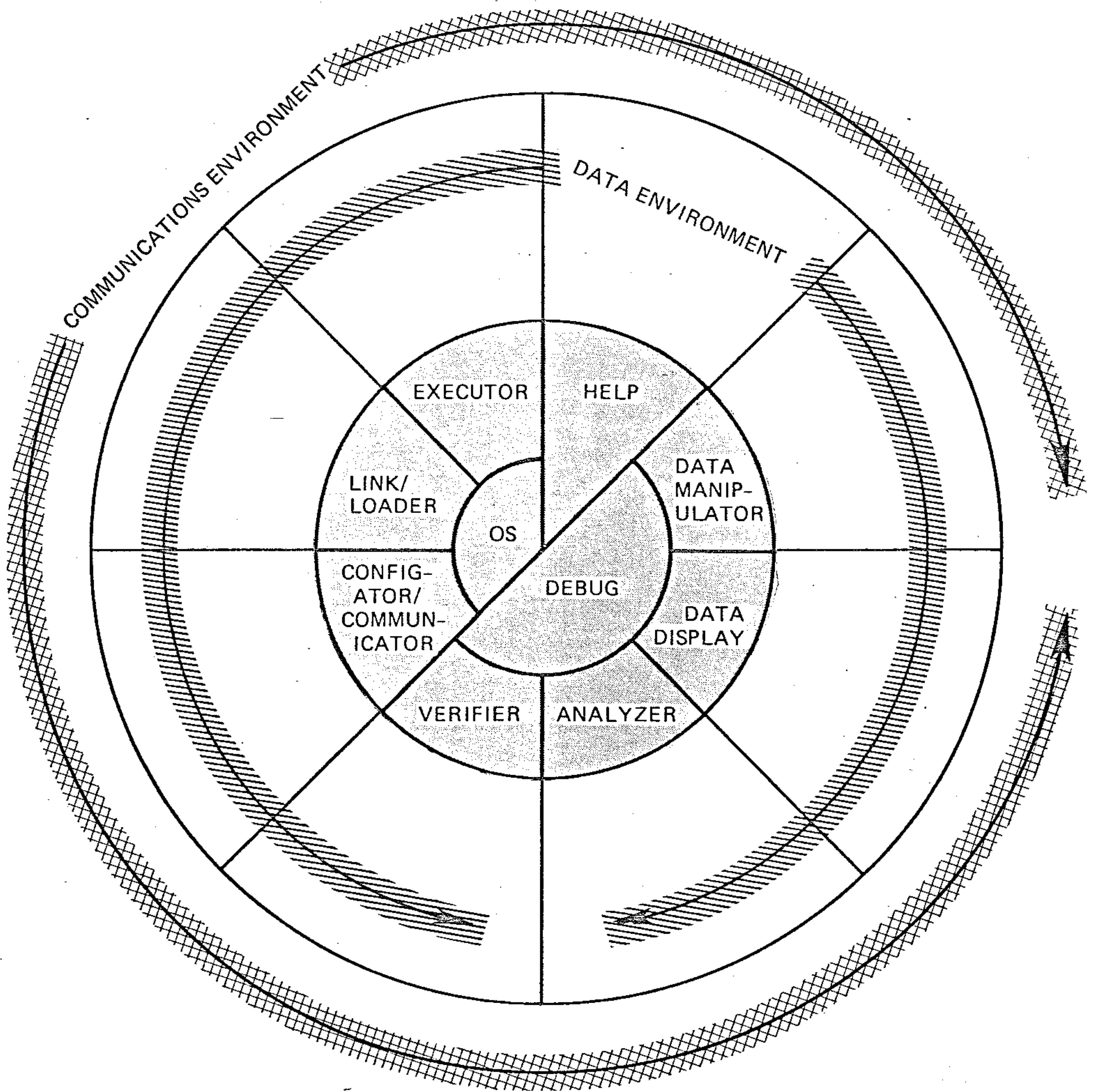

FIGURE 1. GENERALIZED ENVIRONMENT

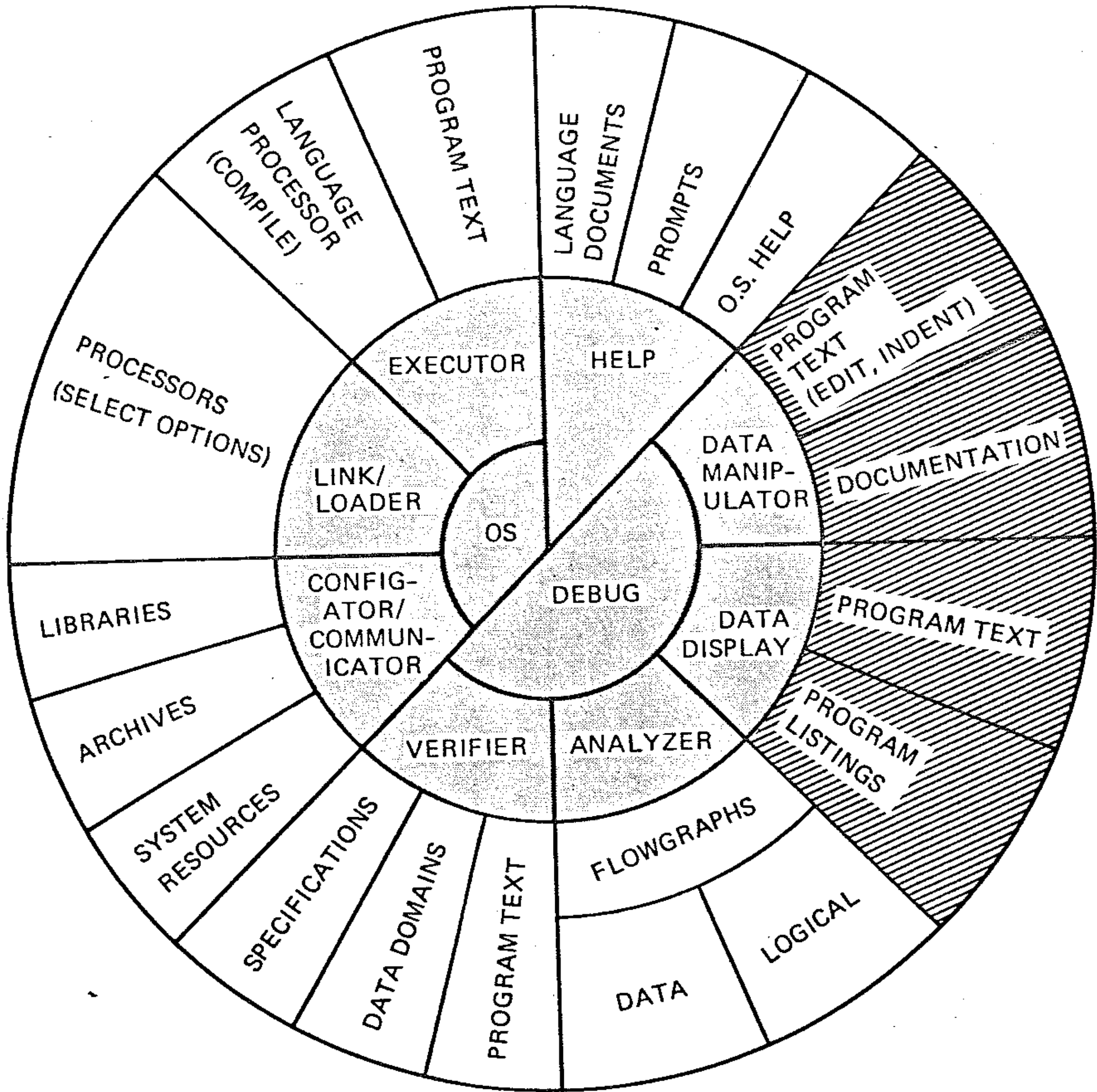

FIGURE 2a: PROGRAM DESIGN

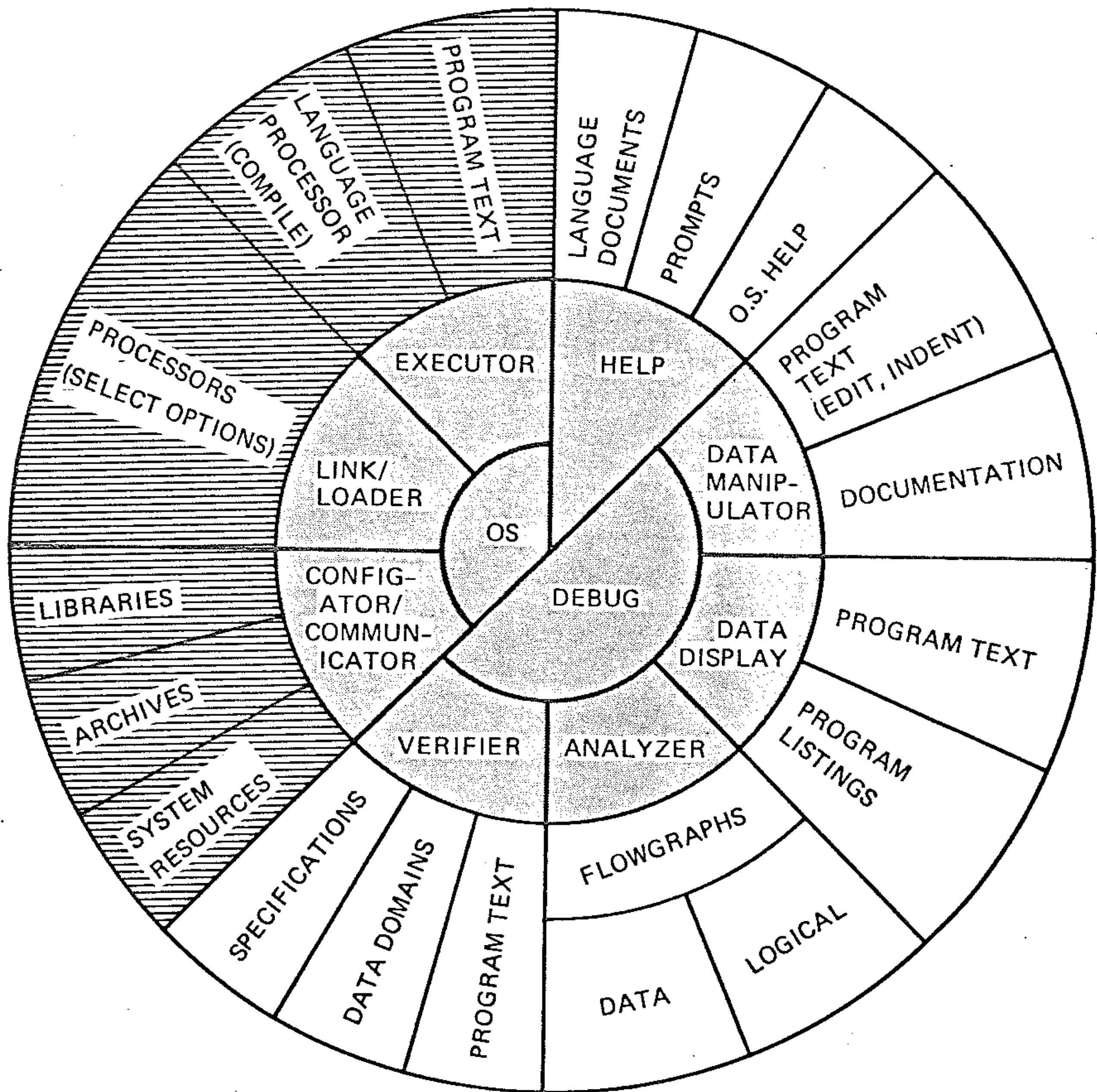

FIGURE 2b: PROGRAM DESIGN

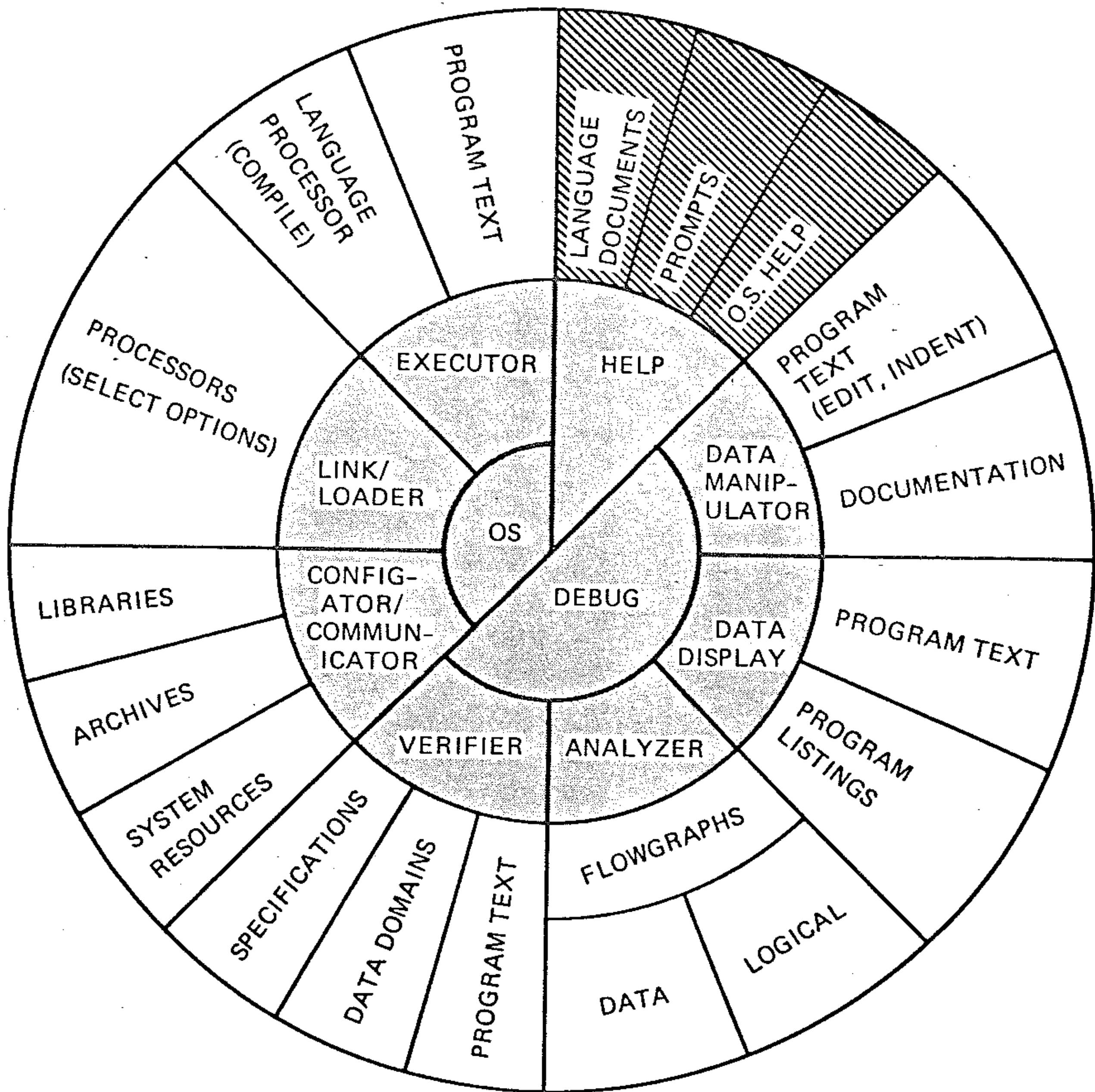

FIGURE 2c: PROGRAM DESIGN

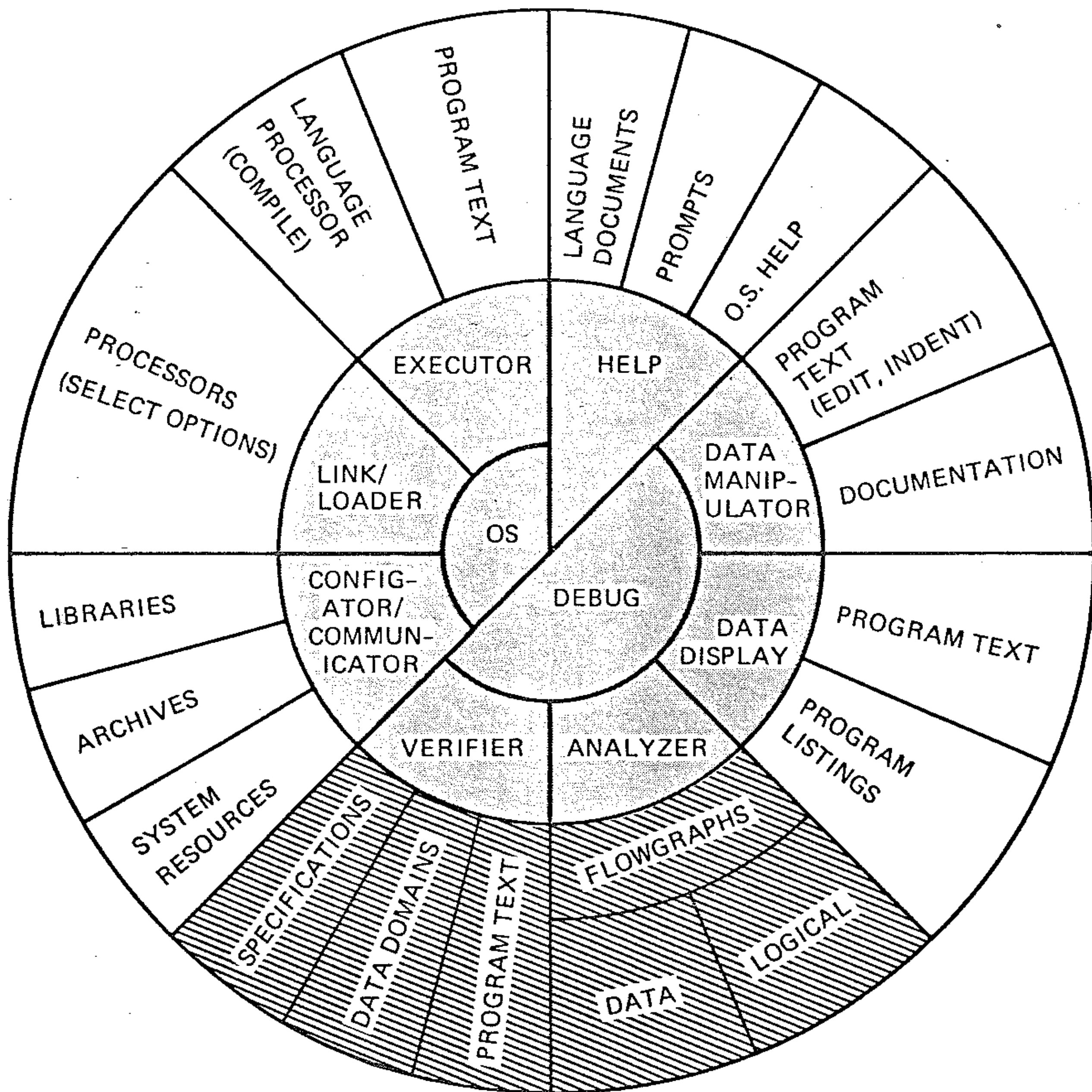

FIGURE 2d: PROGRAM DESIGN

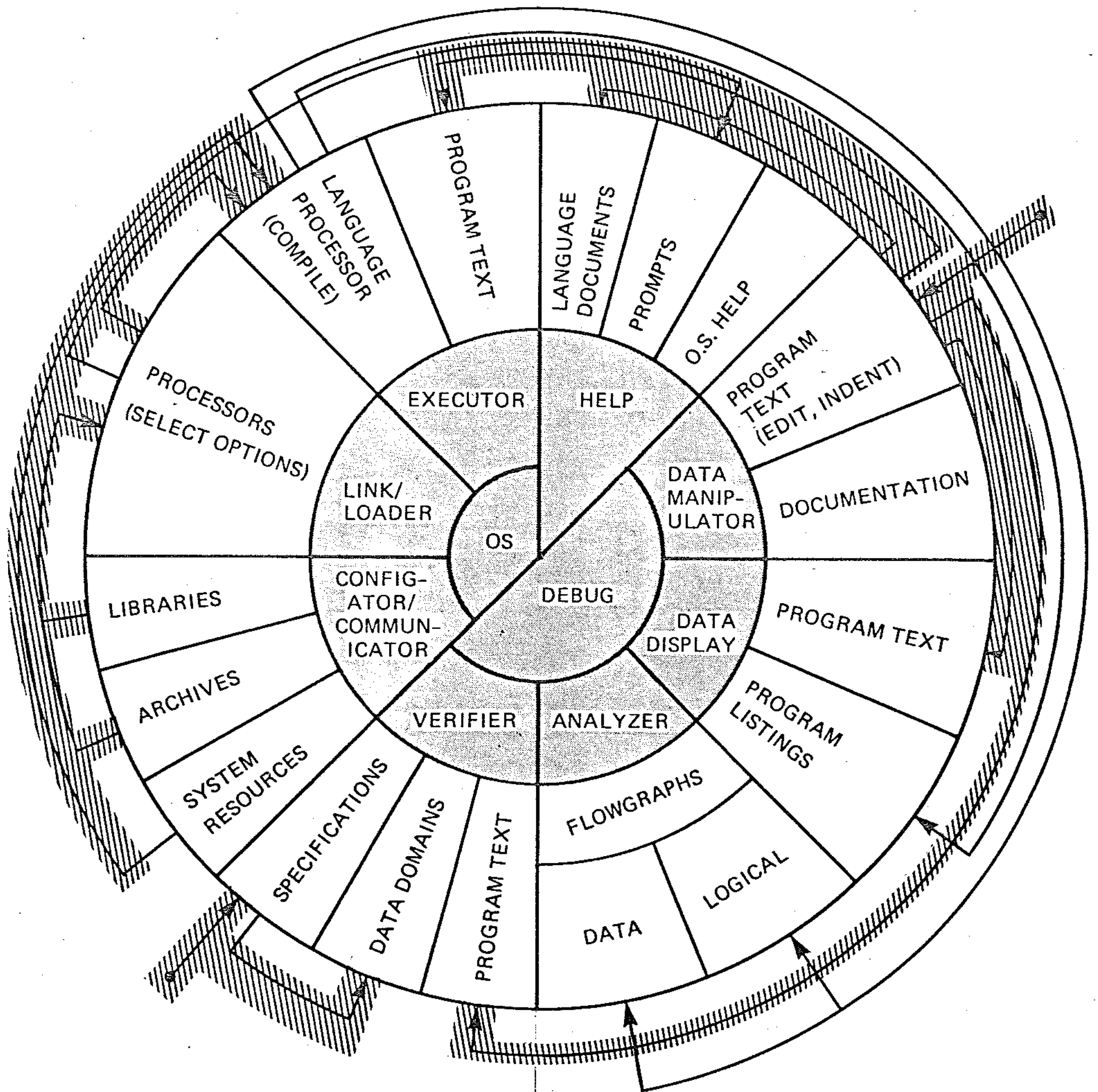

FIGURE 3a: PRE-COMPILE DATA FLOW

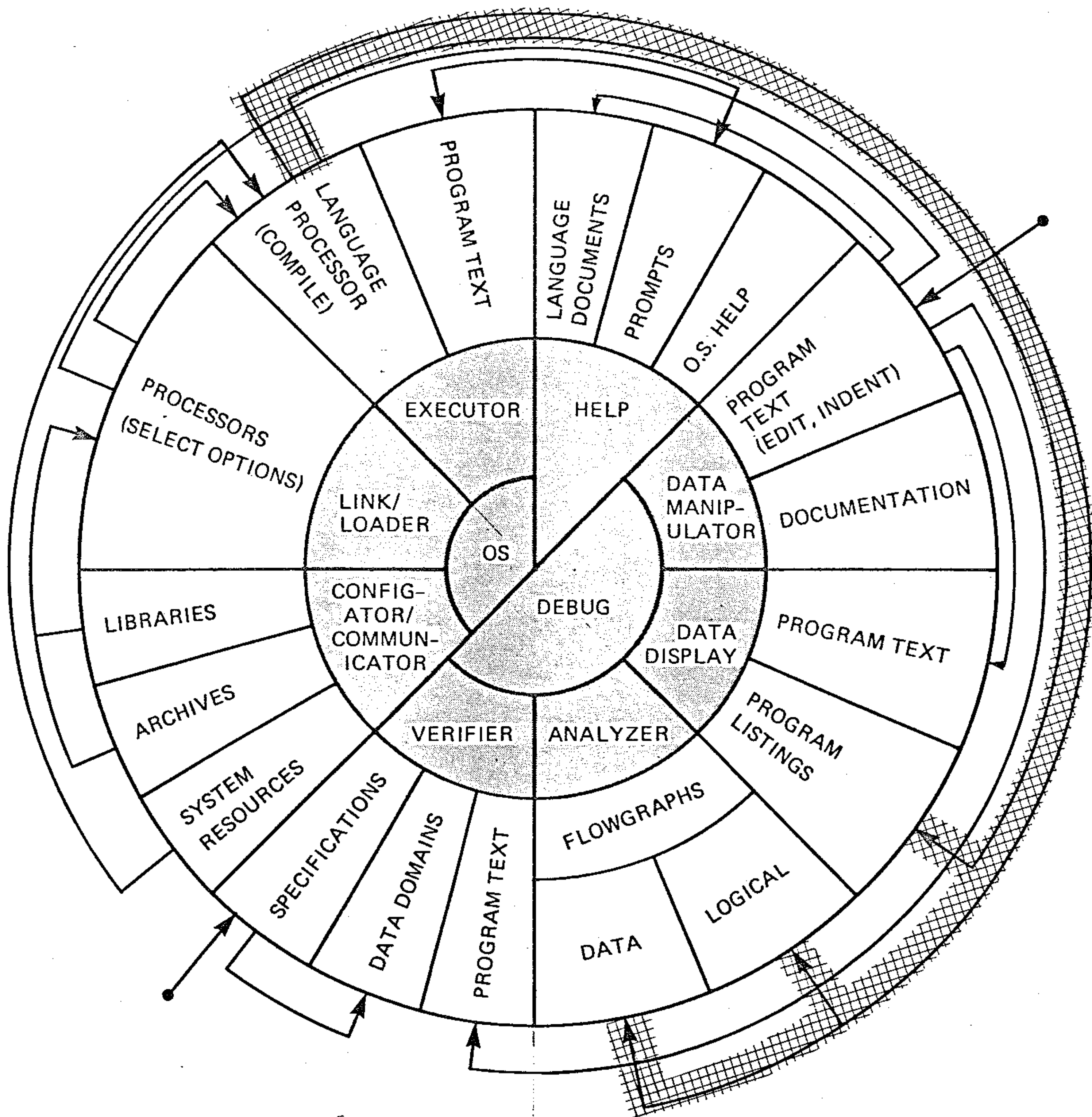

FIGURE 3b: POST-COMPILE DATA FLOW

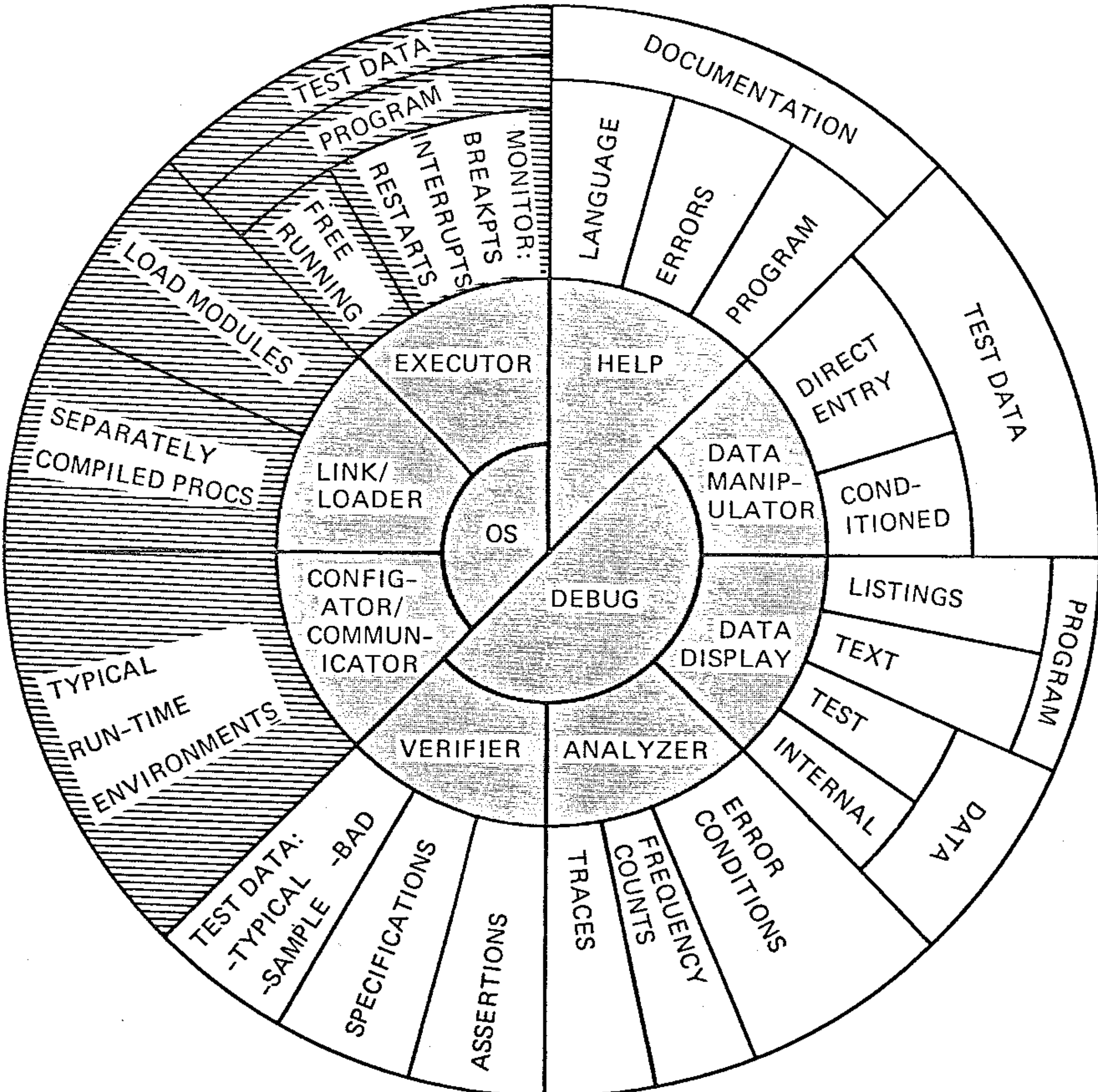

FIGURE 4a: CHECK-OUT ENVIRONMENT

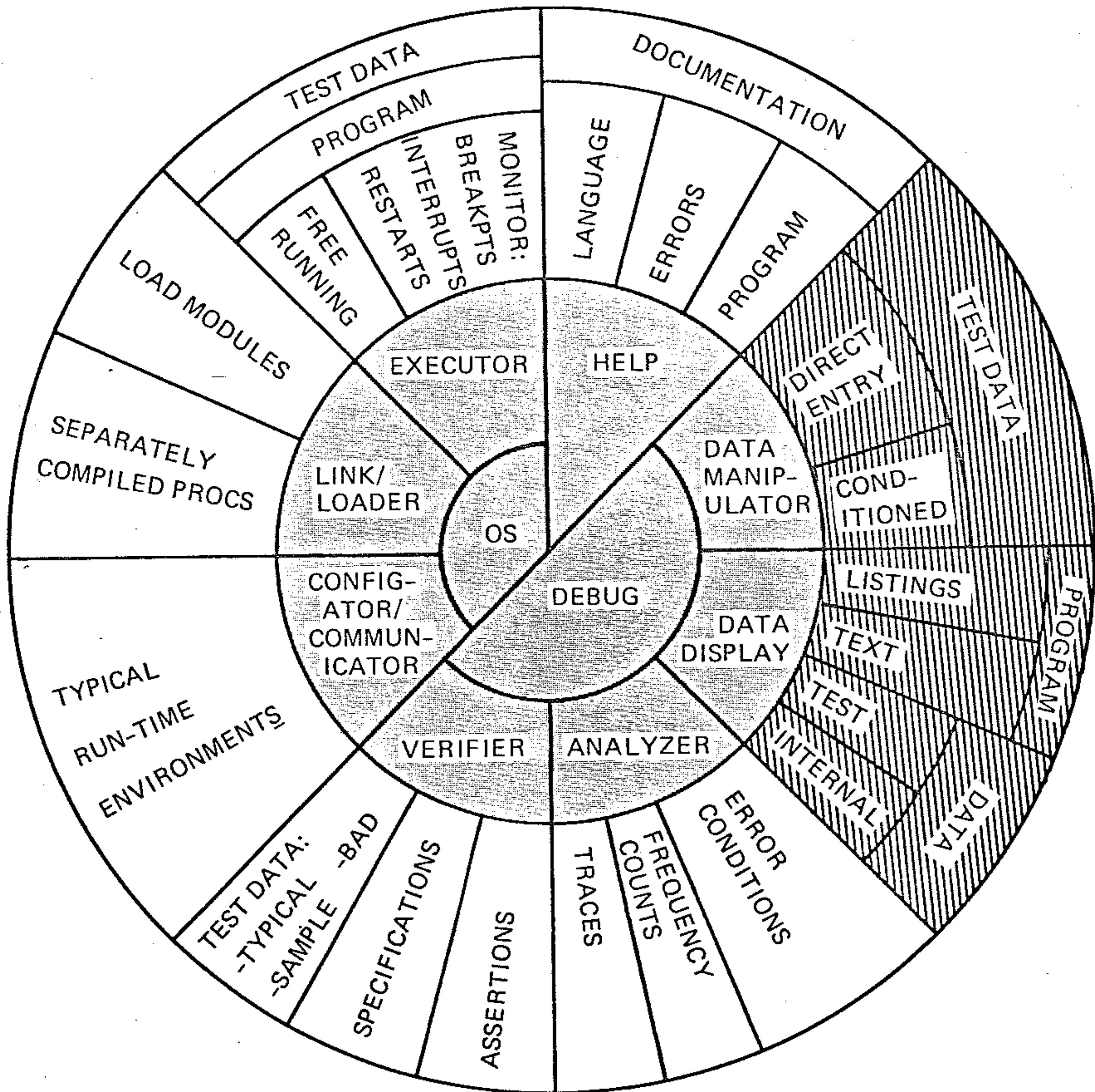

FIGURE 4c: CHECK-OUT ENVIRONMENT

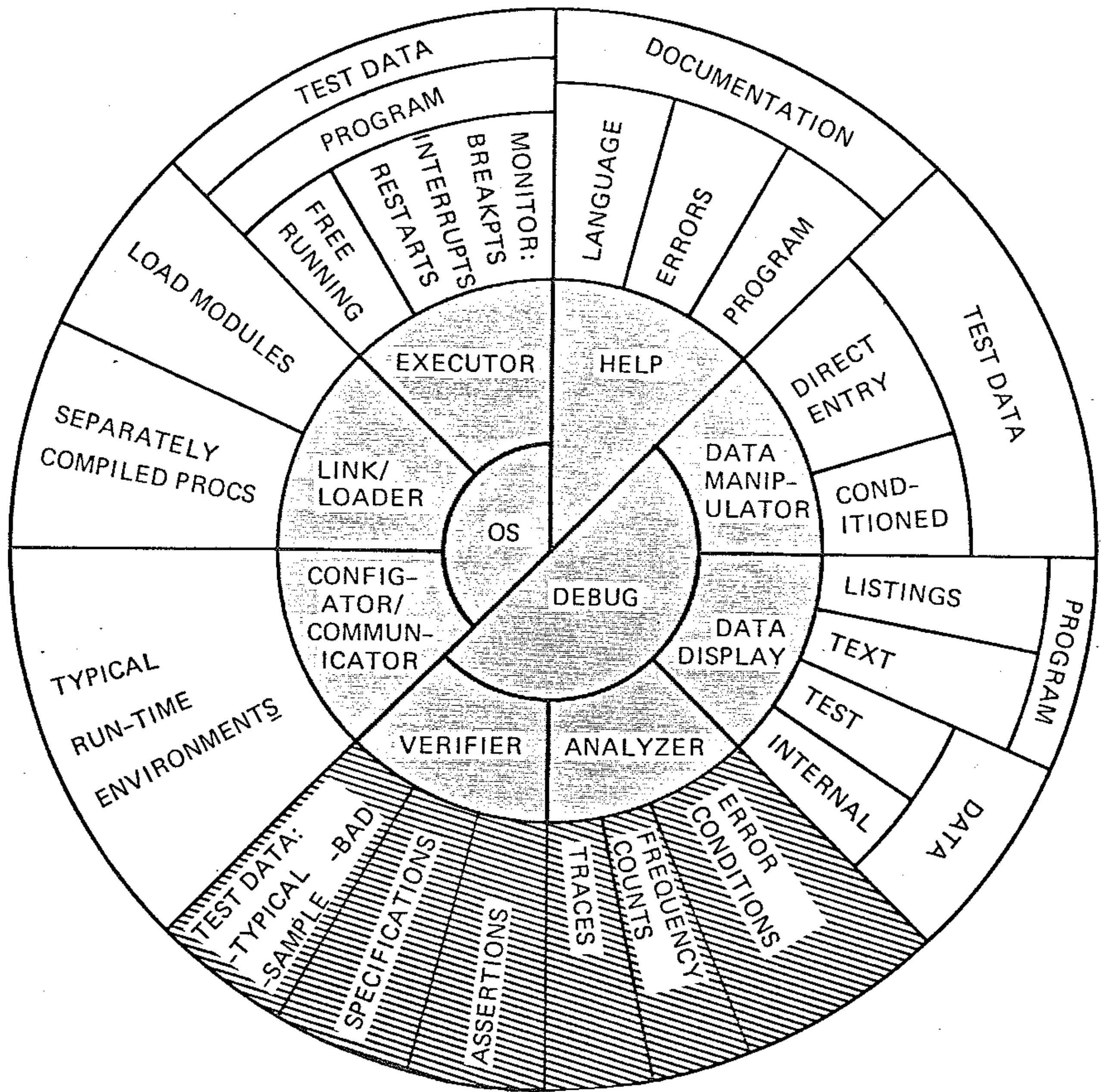

FIGURE 4d: CHECK-OUT TOOLS

TEST DATA
PROGRAM
DOCUMENTATION
MONITOR: BREAKPTS INTERRUPTS RESTARTS
FREE RUNNING
LOAD MODULES
SEPARATELY COMPILED PROCS
LANGUAGE
ERRORS
PROGRAM
TEST DATA
DIRECT ENTRY
CONDITIONED
EXECUTOR
HELP
LINK/ LOADER
OS
DATA MANIPULATOR
DEBUG
CONFIGATOR/ COMMUNICATOR
DATA DISPLAY
VERIFIER
ANALYZER
LISTINGS
TEXT
TEST
INTERNAL
PROGRAM
DATA
TYPICAL RUN-TIME ENVIRONMENTS
TEST DATA: -TYPICAL -SAMPLE -BAD
SPECIFICATIONS
ASSERTIONS
TRACES
FREQUENCY COUNTS
ERROR CONDITIONS

FIGURE 5a: TEST DATA FLOW

TEST DATA
PROGRAM
MONITOR: BREAKPTS INTERRUPTS RESTARTS
FREE RUNNING
DOCUMENTATION
LANGUAGE
ERRORS
PROGRAM
TEST DATA
LOAD MODULES
EXECUTOR
HELP
DIRECT ENTRY
SEPARATELY COMPILED PROCS
LINK/ LOADER
OS
DATA MANIP- ULATOR
COND- ITIONED
DEBUG
LISTINGS
CONFIG- ATOR/ COMMUN- ICATOR
DATA DISPLAY
TEXT
PROGRAM
TYPICAL RUN-TIME ENVIRONMENTS
TEST
VERIFIER
ANALYZER
INTERNAL
DATA
TEST DATA: -TYPICAL -BAD -SAMPLE
SPECIFICATIONS
ASSERTIONS
TRACES
FREQUENCY COUNTS
ERROR CONDITIONS

FIGURE 5b: LOGICAL FLOW

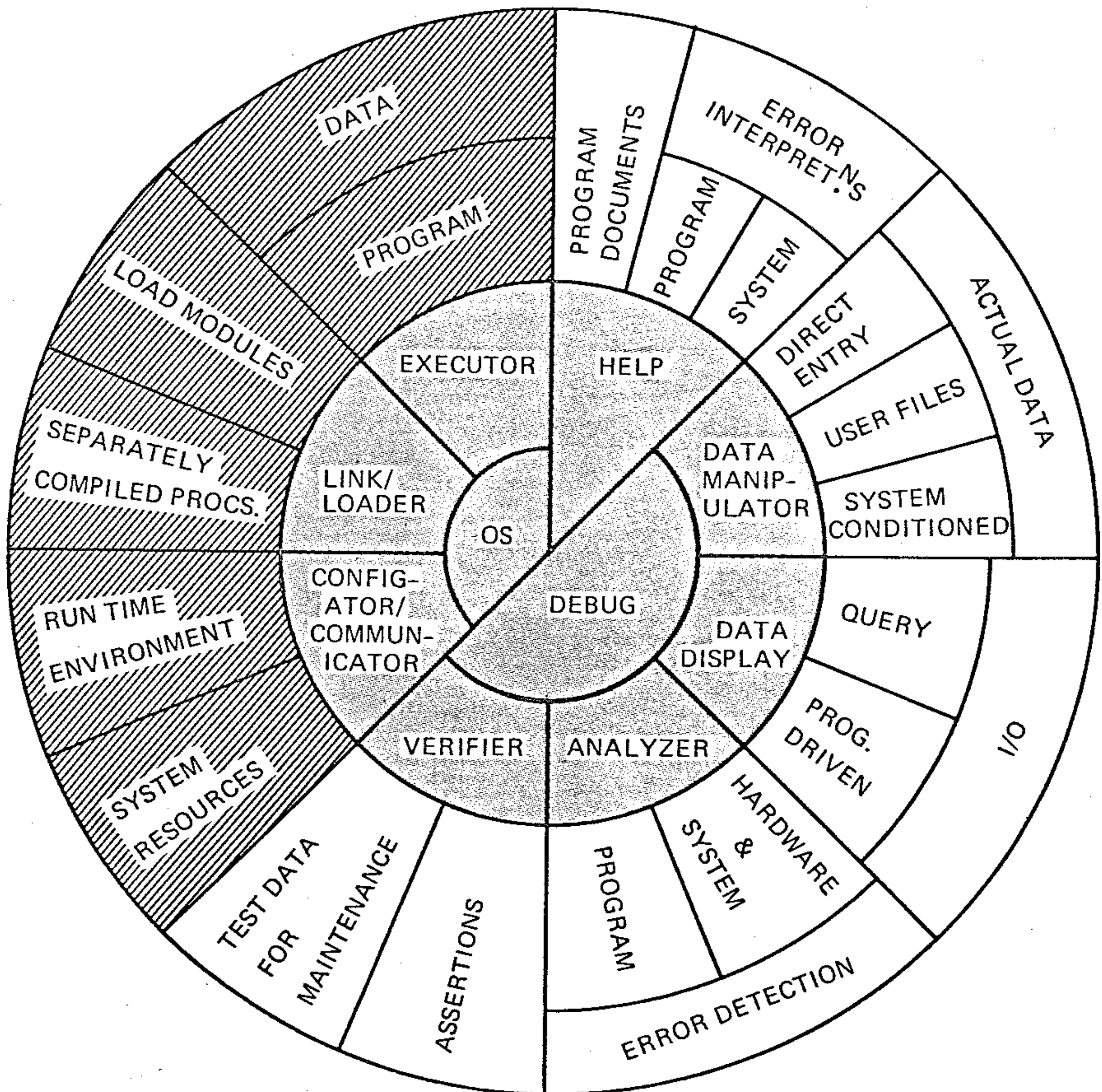

FIGURE 6a: RUN-TIME ENVIRONMENT

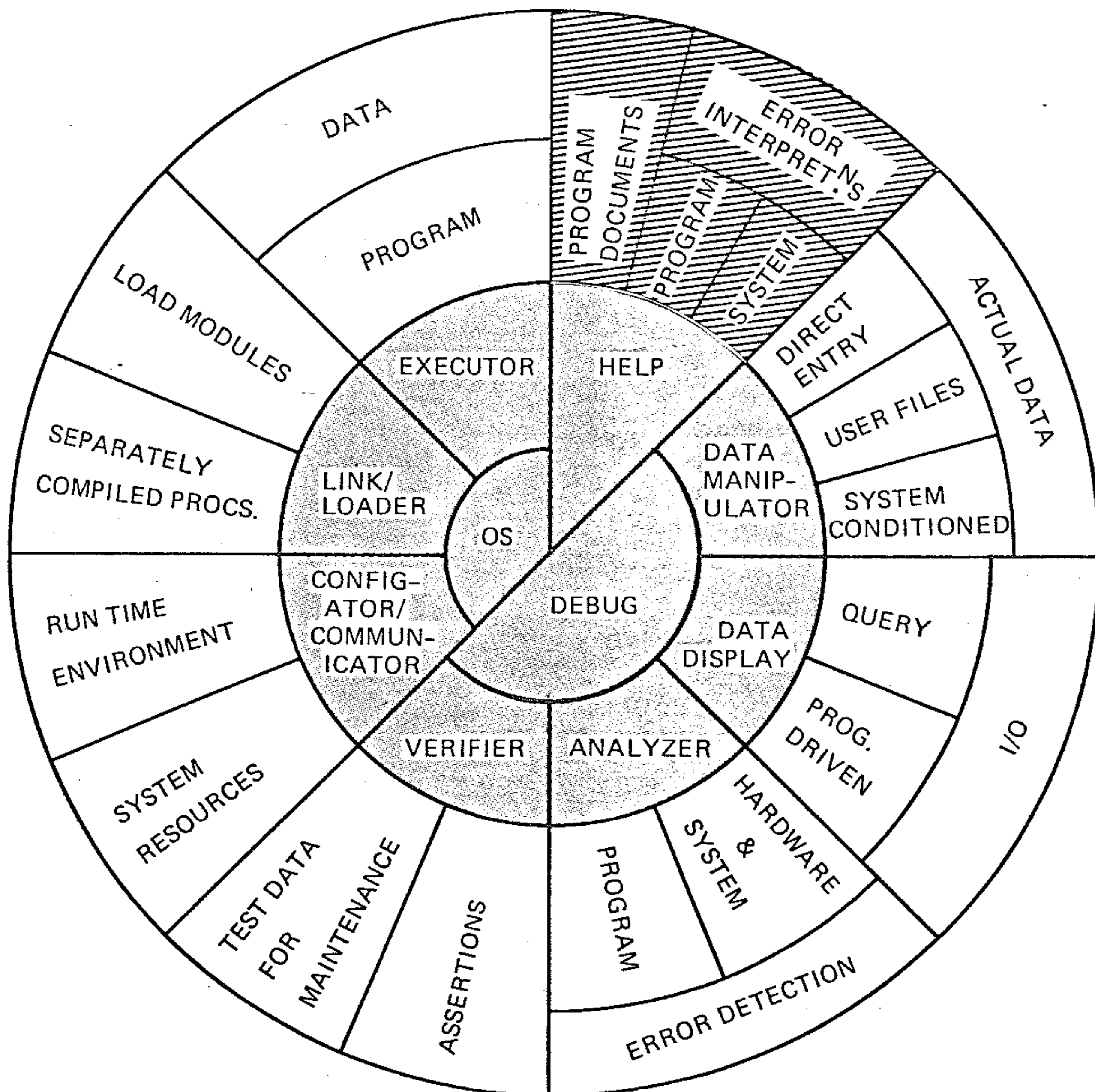

FIGURE 6b: RUN-TIME HELP

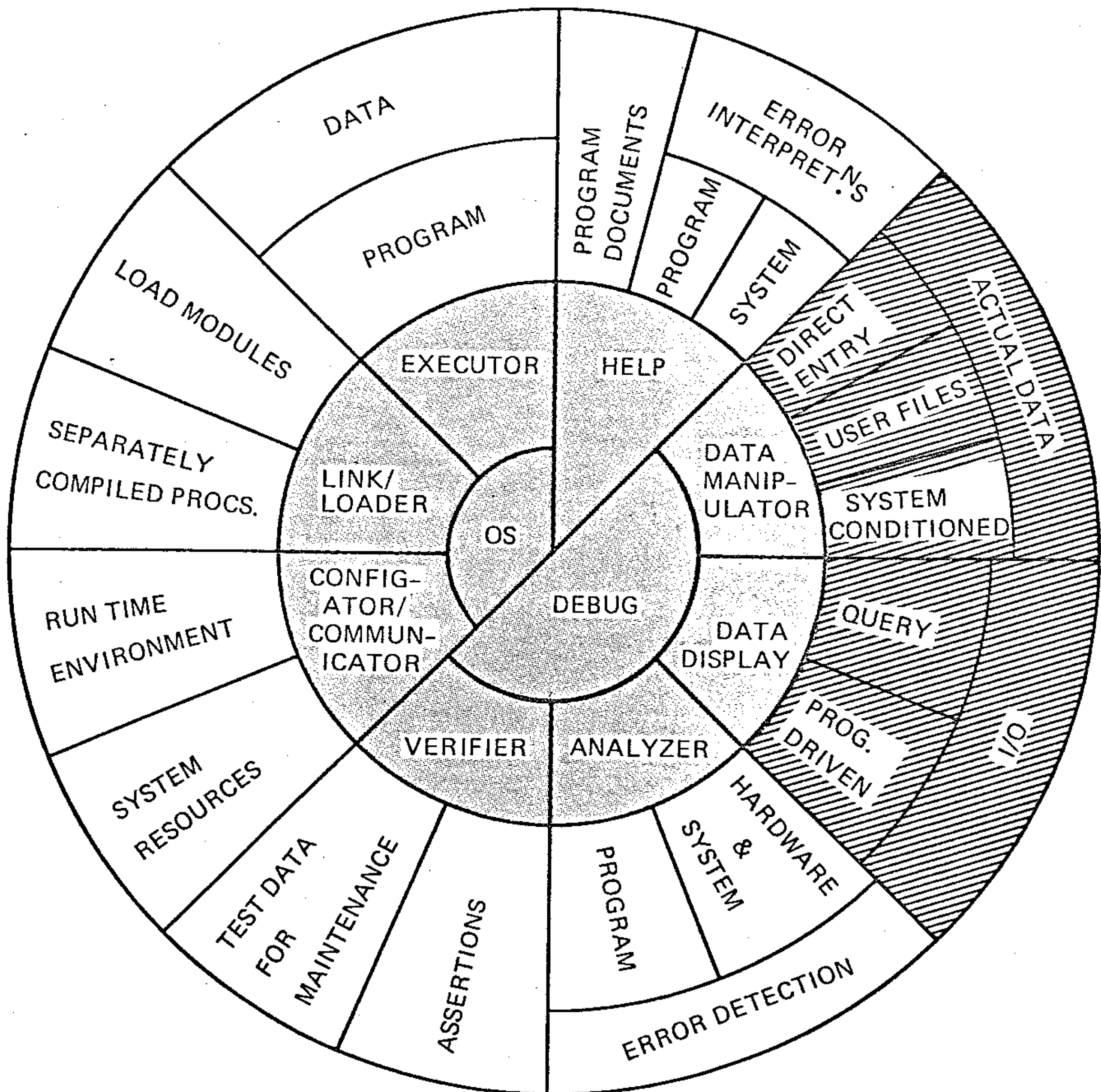

FIGURE 6c: RUN-TIME DATA

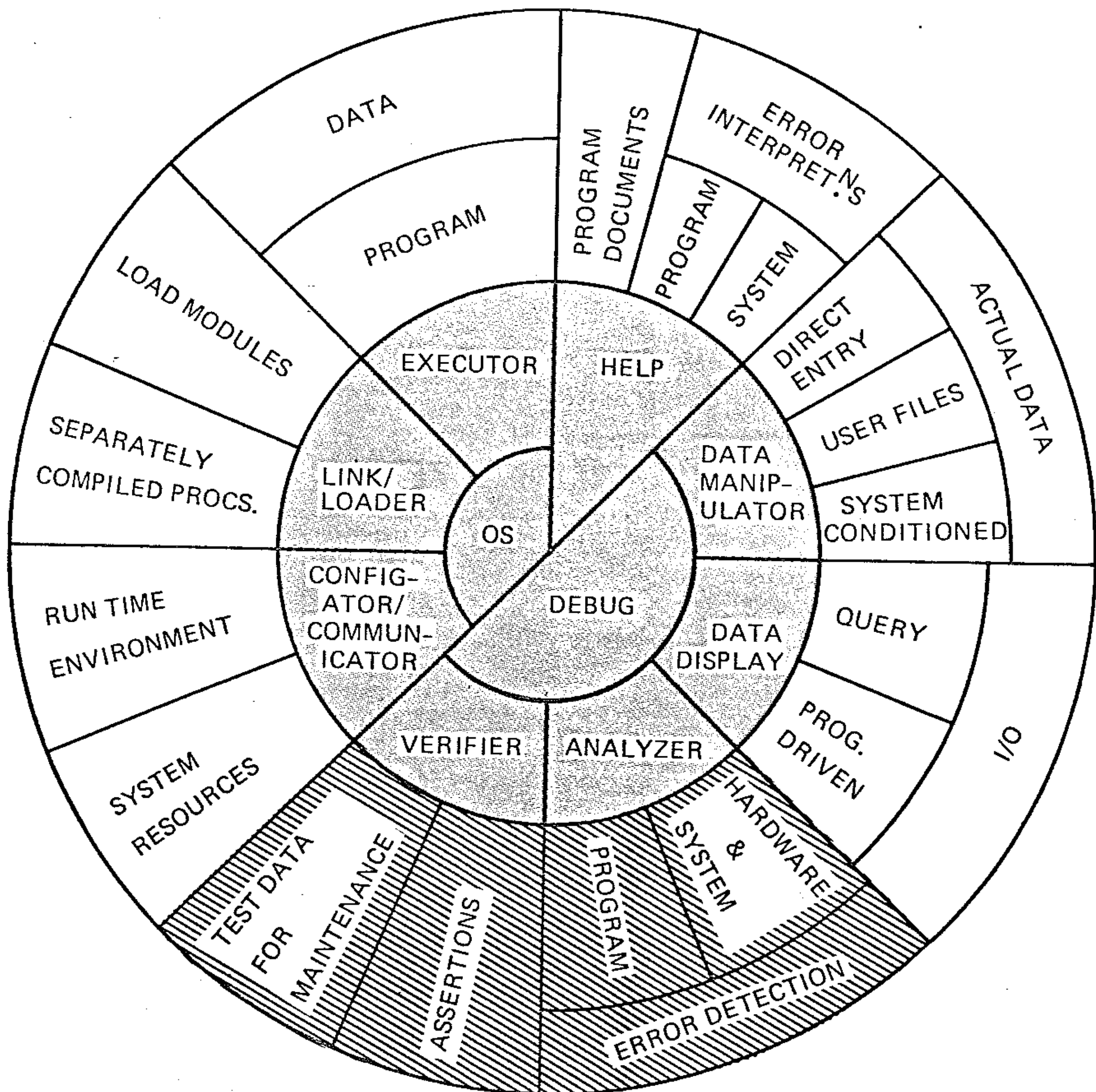

FIGURE 6d: RUN-TIME RELIABILITY

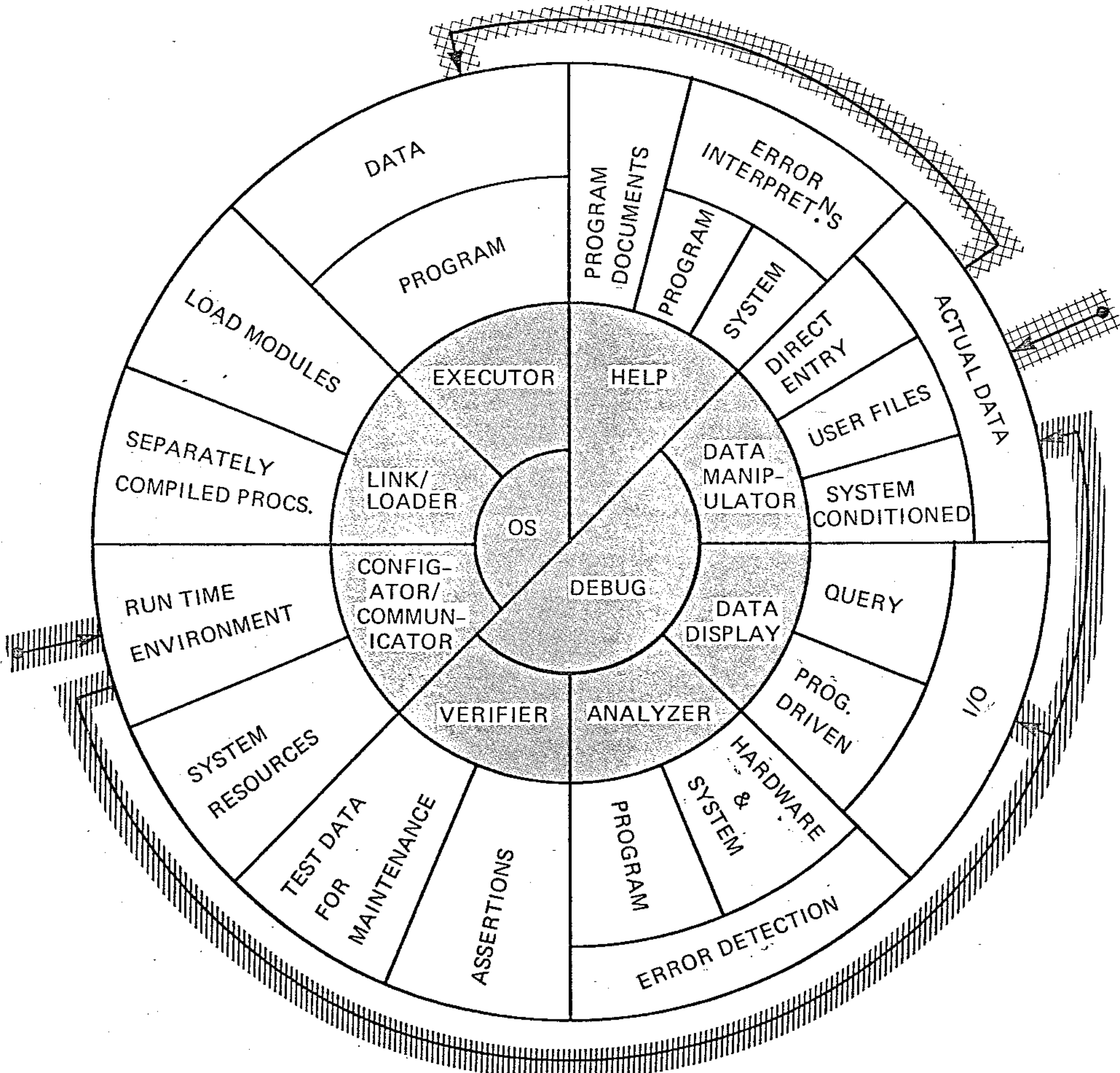

FIGURE 7: RUN-TIME DATA FLOW

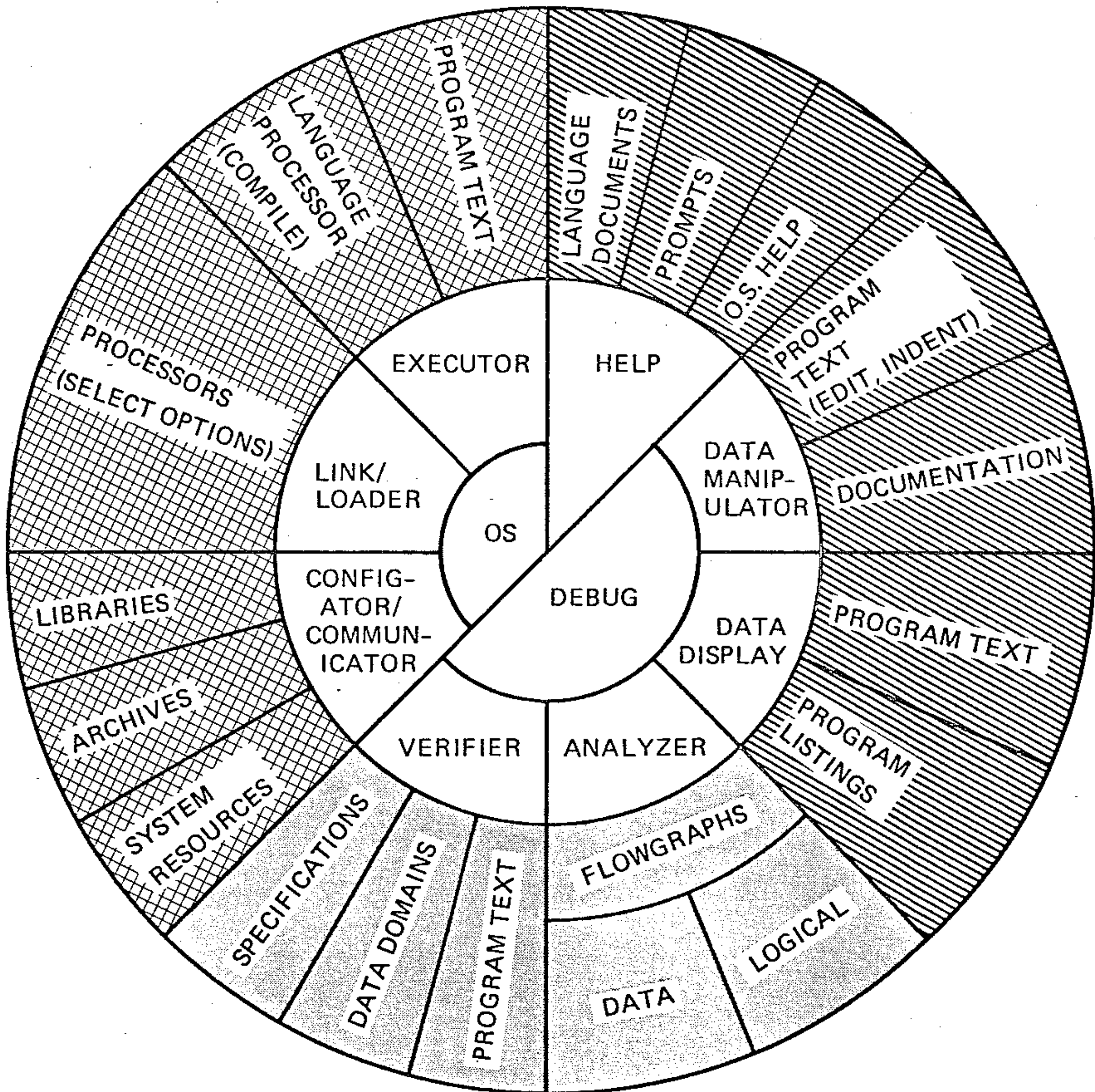

FIGURE 8: DESIGN SUPPORT

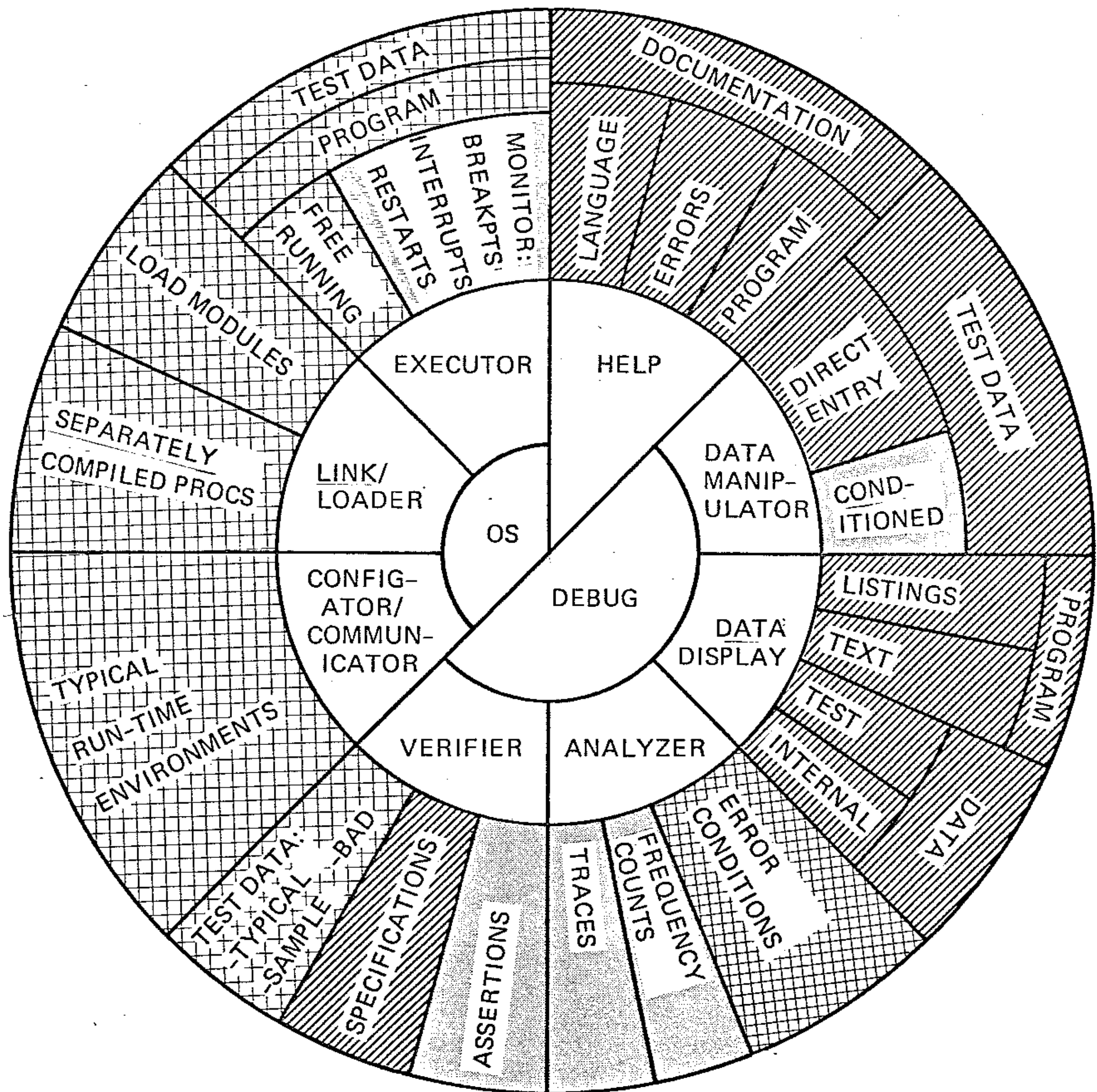

FIGURE 9: CHECK-OUT SUPPORT

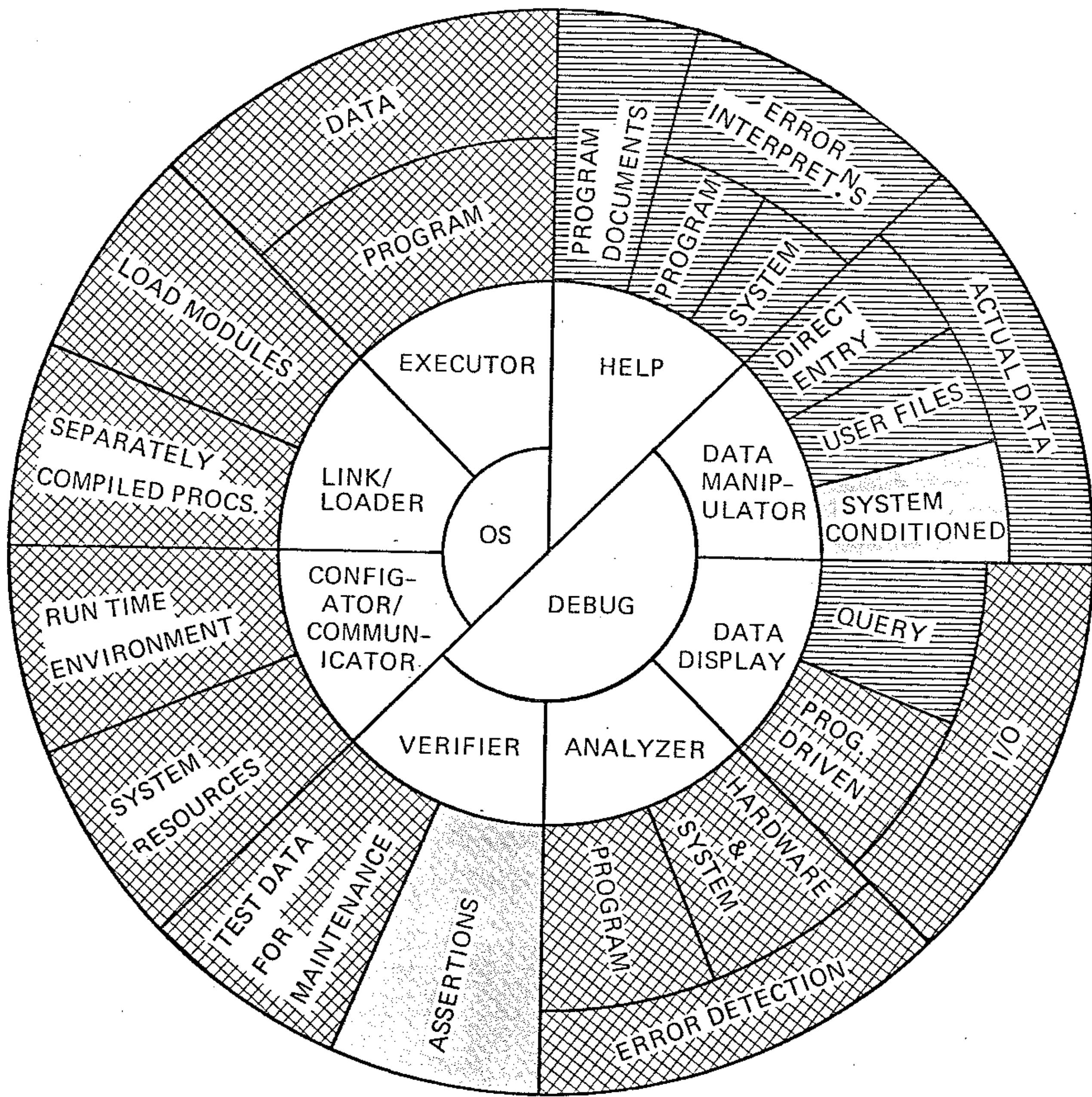

FIGURE 10: RUN-TIME SUPPORT