A software architecture for a mature design automation system

by Richard L. Taylor

Design automation systems are groups of programs used to aid the design of the electronic portions of computers. IBM has used such systems for over twenty-five years, and has a large number of experienced users who place severe requirements upon their design automation system. The essential requirement is that design programs must be easy to use in the way that a particular development location wishes to use them. Design programs which cannot be changed to meet local requirements are not acceptable. The software structure portion of an architecture for a design automation system which meets these requirements is described; interactive and foreground design applications are stressed. The structure involves a uniform set of services, such as command languages and terminal access methods, needed to support the design application, and a formal, two-part partitioning of the design application itself. Modularity and good interfaces are important parts of the software structure which permit a development laboratory to change the application enough for it to be easily used. These changes are the rule, not the exception. Examples of such changes are given, based upon early user experience. The applicability of this architecture to future system needs such as distributed data and distributed processing is discussed.

Mature design automation systems

The phrase *design automation system*, as used in this paper, means those design application programs which are used to design the electronic portions of computers such as chips, cards, modules, and boards.

The IBM Corporation has used design automation systems for many years, now having over a quarter of a century of experience in implementing and using such systems. A history of these systems is contained in [1–3]. The present version of the IBM design automation system, the Engineering Design System (EDS), is almost thirteen years old. Thus, the IBM design automation system has longevity, but other characteristics also make it a mature system.

It is widely used by a large number of people at IBM locations around the world. These people are experienced users of design automation systems and place severe demands upon them. The initial euphoria about any program which replaced manual design labor has long since gone away.

The IBM design automation system contains a large number of different design applications, capable of supporting many different technologies, including those used in the newest IBM processors [4, 5].

IBM development laboratories are autonomous units, each being free to employ design methodologies that best apply to its local conditions and to the products under development. Each laboratory is free to employ the IBM design automation system as it sees fit, replacing that system with locally written design applications or purchased programs. A number of people at each development site are engaged in the implementation and maintenance of such local methodologies.

It was this combination of factors—many experienced users with many different applications, and the use of diverse local design methodologies—that led to the system requirements.

Requirements

The requirements apply to interactive and foreground applications only, that is, applications which are initiated from a terminal and which execute synchronously with the user at the terminal. While there are still many batch applications in the EDS, they are gradually being replaced by interactive or foreground jobs.

The evolution of the requirements for the software portion of the system architecture occurred over a number of years, and was slow and painful. Mistakes were made, both by the developers and the users of the architecture. The evolution of requirements can be divided into three phases.

First came the experimental or prototype phase. There have been experimental efforts to perform interactive design applications since the early 1970s with the the advent of the Time-Sharing Option (TSO) for the MVS operating system. A small number of interactive applications were developed, each using its own architectural constructs. There was little or no resemblance among these early applications, and each one had to be addressed on a case-by-case basis in regard to installation, usage, and potential local modification.

The second phase was dominated by the application developers. Interest in interactive applications grew slowly for a number of years, but then started a period of rapid growth due to the availability of adequate computing power and adequate numbers of terminals. The majority of these terminals were alphanumeric, members of the 3270 Information Display System family of terminals. A limited number of terminals were high-function vector graphic, such as the IBM 2250 or IBM 3250.

As the number of interactive applications grew, the outlines of a system architecture for those applications evolved. Applications were written to a common architecture. In many ways, this phase was a step forward, but it was a very rigid architecture, with little or no flexibility. There was insufficient understanding of the requirements of the various development locations which would use the design system. This lack of understanding is not completely the fault of the application designers or the system designers; the users, in general, did not understand their requirements.

The third phase of requirements evolution can be called the mature phase. The users have now had enough experience with interactive applications to have a good understanding of their requirements. The system and application designers have tried a number of alternative designs, and have a much better idea of what will work and what will not. Both groups are looking to a future with distributed data and professional work stations.

The basic requirements from the users can be simply stated:

- Deliver a system of design applications that will work as delivered.
- Deliver interactive applications in such a system environment so that the applications can be easily used at a development site.
- Design the system to allow easy transition into future computing environments, such as distributed data bases and professional work stations, even if the exact details of such environments are not known.

Many detailed steps are needed to implement these requirements, but a common theme is the notion of having modular design at all levels of the system, with well-defined, stable interfaces between the modular parts.

The system architecture

The software portion of the system architecture is concerned with two major interdependent parts: the application environment and the interactive application itself.

The application environment is the computing environment that the application sees external to itself. The major element of this environment is the architecture of the computing system on which the application executes. The EDS applications assume a System/370 architecture, although no overt recognition of that architecture is made. Conversely, no attempt has been made to enforce a "machine-independent" mode of coding by using only certain constructs from high-level languages. Some thought has been given to this approach, but it has not proved successful.

This major portion of the applications environment is defined by the services that an application can call upon for its use. These services are called application support programs.

The application must conform to certain guidelines or standards. Use of the application support programs is one such standard, but many other more difficult guidelines must be followed.

- *Application support programs*

Application support programs provide various specific services to the applications. The software architecture specifies the following application support programs.

Terminal monitor program

The terminal monitor program manages the communication protocols between the application and the terminal.

Terminal access methods

Terminal access methods are the means by which an application program communicates with a terminal. This communication can be as simple as "write a line" or "read a line," or considerably more complex, such as "write a full screen of information at once, doing field substitutions in the process."

Parameter bases

Parameter bases are means of saving certain data from one terminal session to another, and supplying that information so that the terminal user does not have to re-enter it.

Parametric data are communicated to the application by means of a parameter interface area (PIA).

Command languages

A command language is similar to a TSO CLIST or a VM/CMS EXEC. The command language is used for certain specific parts of every application. This use is explained in the next section.

An overview of the application support programs is given in **Figure 1**. Each of the application support programs is described in detail, but there are several important architecture goals which apply to all of them. These goals are described as follows:

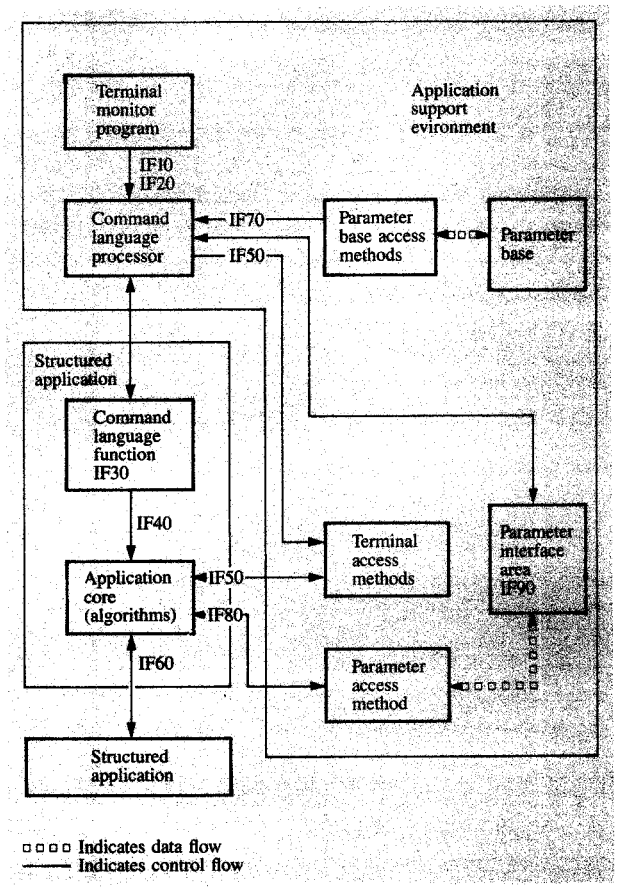

Figure 1

Application support programs.

- *Adequate functionality.* The support program must provide enough function to be useful to the design application. For example, a terminal access method for graphic terminals must permit convenient construction of any display. This requirement may appear to be obvious, but many of the interactive applications are quite complex and place severe functional demands upon the support programs.
- *Modularity.* Modularity means that the services provided by the support programs are grouped into packages of related independent services. Typically, each package of services is one program load module.

  Putting related services into one package means that a particular group of services can be used without any dead code. The parameter base is not packaged with a terminal access method.

  Independent services means that it is generally not necessary to load one service package as a prerequisite to another.
- *Replaceability.* It must be possible to replace an application support program with another functionally equivalent program. A necessary condition to allow replaceability is to have well-defined interfaces for the support programs.

The interface definitions must include calling sequences, data structures, and precise descriptions of the actions that result from a call. In other words, the interface definition must be precise enough so that a user of the system could write a functionally equivalent program. Further, the interfaces must remain stable long enough so that replacement programs written by users will be useful for a long time.

EDS guarantees that the interfaces for its application support programs are functionally complete and will have long-term stability, thus making it practical for other groups to write programs which use EDS interfaces. The locations of the major interfaces are shown in Fig. 1 as IF10 through IF90.

The replaceability requirement also dictates that the service packages be loaded at time of use, rather than permanently linked into the application.

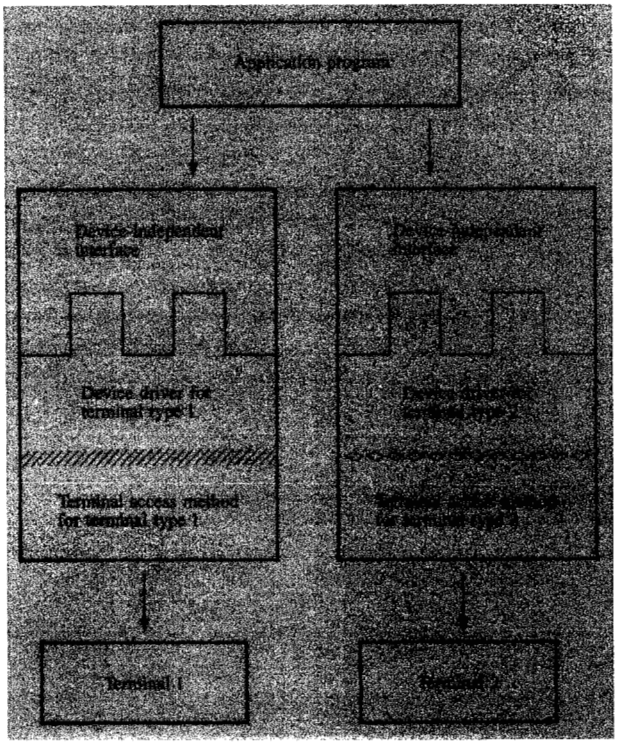

Figure 2

Device-independent access method.

- *Efficiency.* There are a number of aspects to program efficiency, but the ones of chief concern are execution time and virtual storage occupancy.

  Studies have shown a direct correlation between rapid terminal response and productivity in interactive applications. Many EDS interactive applications try to respond to trivial actions in a fraction of a second. Any application support program which is invoked in the process of providing the response must therefore operate in a fraction of a second.

  The amount of virtual storage occupied by service programs is of great concern in today's LSI-VLSI design environment. The size of parts being designed seems to grow faster than the amount of available storage! Much attention was paid to the size of the support programs and to various strategies to minimize their storage occupancy. Modularity helps greatly in management of storage. It is necessary to load only the services that are needed by an application. In some cases the modularity was carried to a further level. Some support programs were partitioned into two or more load modules, guided by an estimate of frequency of use of particular services. In many cases, only one module from the set is loaded into virtual storage.

Terminal monitor program

The terminal monitor program (TMP) controls the two-way communication between the application program and a terminal. Such a program is often supplied as part of an operating system, or it may be necessary to write a special terminal monitor program for a particular terminal. In our case, we found the IBM-supplied terminal monitor program adequate for the 3270 IDS family of terminals.

We found it necessary, however, to write our own terminal monitor program for the 2250 and 3250 terminals. Standard IBM product offerings were not adequate for our application set. One of the great challenges in the design of the terminal monitor program was the exact amount of function to put in it. Since this program will always be present when high-function graphic terminals are in use, there was a tendency to include other functions, such as dump management, which had nothing to do with terminal communication. Inclusion of these other functions clearly violates the modularity requirement. A great deal of design effort was spent on this program, and we now have a terminal monitor program that meets the modularity and efficiency requirements.

Terminal access methods

Two terminal access methods are available, one for alphanumeric terminals such as the IBM 3277 or 3278, and one for vector-graphic terminals such as the IBM 3250. Each terminal access method appears to the application programmer as a number of CALL statements, each of which performs a specific service.

Alphanumeric terminal access method The alphanumeric access method consists of a few dozens of CALLS. Major functions are the construction and presentation of full-screen displays, and the return of data from such displays to the application program. The displays may be initially stored in a data set, or they may be constructed dynamically by the application program. Full support of input devices such as a keyboard, program function keys, and a selector pen is also provided in this program package. This access method depends upon the presence of the standard IBM terminal monitor program, supplied as part of the operating system.

Displays constructed by using the alphanumeric terminal access method may also be presented on a high-function graphic terminal. In this case, the screen buffers for the alphanumeric terminal are translated into graphic orders and passed to the graphic terminal access method. Thus, the user of a graphic terminal can often run an alphanumeric application upon that terminal, avoiding the need to LOGOFF the graphic terminal and find an alphanumeric terminal.

Graphic terminal access method The graphic terminal access method consists of several hundred CALLS. A

comprehensive group of services is provided, allowing the rapid construction and presentation of complex vector-graphic displays. Full support of input devices such as keyboards, program function keys, and light pens is provided. This access method depends upon the presence of the EDS terminal monitor program.

Major design effort has been spent to ensure a high measure of device independence for this access method. That is, the graphic terminal access method is designed to run with a number of different types of graphic terminal, not just one. The design approach is to separate the access method into two main parts, one of which is independent of device characteristics, and the other of which is tailored to a specific device. This second part is called a device driver. When it becomes necessary to support a new graphic device, it may only be necessary to write a new device driver. This design is shown conceptually in **Figure 2**. This aspect of the terminal access method design, and many others, is consonant with the SIGGRAPH CORE proposal, a proposed standard for graphic access methods [6].

This design concept has been very effective. Support for a number of new terminals has been rapidly and economically implemented by writing new device drivers. Additionally, plotters can be regarded as a special type of terminal, and a number of them have been easily supported with this design.

Parameter base

The parameter base and the parameter access services provide a view of parameter storage to the user.

Parameters are used for two distinct functions in this architecture.

Parameters provide user convenience, by placing values in displays for interactive applications. These values may have been saved from a previous terminal session, and hence the terminal user avoids unnecessary key strokes. This is a conventional use of a parameter base, similar to the way parameters are used in the IBM Interactive System Productivity Facility (ISPF).

Parameters also provide a very important control function, by forcing certain actions or specific operand values in certain commands. Values of these control parameters *cannot* be changed by the individual terminal user. This is a less conventional use of a parameter base.

Each parameter has four attributes: value, control character, standard question, and syntax model:

- The value of the parameter is the character string that appears at the terminal.
- The control character determines whether the parameter may be changed, whether it may be displayed but not changed, or whether it may be changed but not displayed (as, for example, a password). The control character is necessary to implement the control function of the parameter base.
- The standard question is a character string that is used when prompting for the parameter value. Thus, every prompt for a given parameter is made in the same words.
- The syntax model is a description of the allowable syntax for the parameter, and is used for checking purposes when the parameter is entered at the terminal.

Programs access parameters by means of access services which search an in-storage data structure. This data structure, the parameter interface area (PIA), is one of the stable system interfaces. Since application programs access this structure, various means could be used to physically store parameters and place them in the parameter interface area. In actual use, the PIA is one of the most important interfaces.

EDS supplies a parameter base on a data set, and access methods. At any instant in time, the parameter base appears to be a tree structure, with parameters stored at the nodes of the tree. (The actual structure of the parameter base is a directed graph, as explained later.)

The user is "connected" to a leaf of the tree, and has access to all the parameters from the leaf to the root of the tree.

The same parameter may exist at multiple nodes in the tree structure. In this case, the value of the parameter closest to the leaf of the tree is accessible to the program.

The tree structure makes it possible to place parameters which all terminal users require near the root of the tree, and to have parameters which are unique to each user at leaves of the tree. The tree structure also supports the control function of the parameter base. Control parameters which pertain to many terminal users are placed near the root of the tree. While it is technically possible to have unique control parameters for each user, this is not typical usage.

Figure 3 shows a simple example of a parameter base which supports site, department, project, and user parameters.

In this example, the site parameter MSGCLASS is set to the value A. This could be a control parameter, and therefore all users of the parameter base use MSGCLASS A.

Each department node has specified a different value for the ACCOUNT parameter, another control parameter which cannot be changed by the normal user of the parameter base. Each department node has one or more project nodes, containing a part number parameter (PART #) and a TIME parameter pertinent to processing that part number. Users are connected to various projects, and hence to the department and site nodes.

Users Bob, Mary, and Henry have defined no parameters at their nodes. Users Mark, Bill, Iris, Rhonda, and Marina have found it necessary to override the standard value of the TIME parameter. User Lou has specified a value for a new parameter, MSG.

The actual structure of the parameter base is a directed graph, and the connection between nodes is selected (or

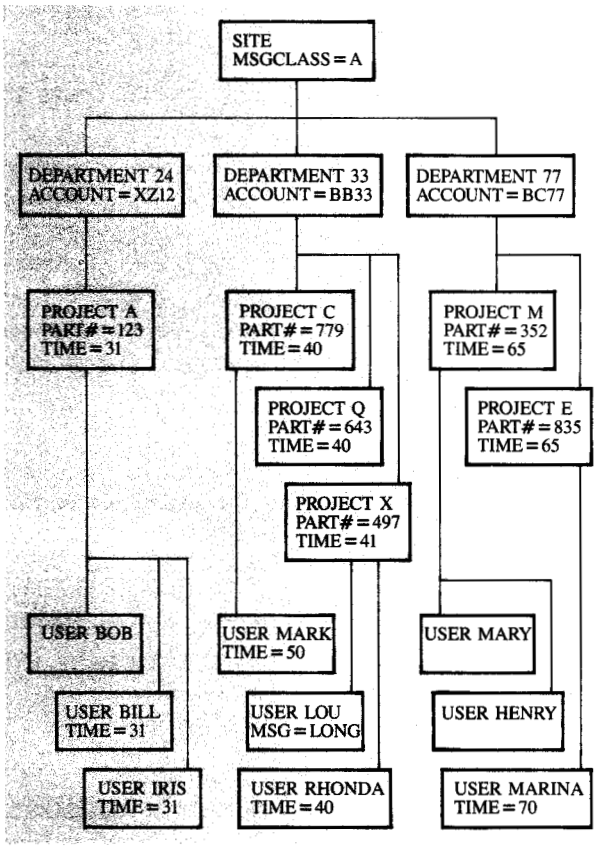

Figure 3

Example node structure.

defaulted) when the parameter base is opened for a given user. This more complex data structure allows a number of different meaningful paths to be defined. Two popular strategies are to define parameter access paths based upon the release level of the design programs being used, or upon a particular technology being employed. In the example above, it allows the user nodes to be connected to different project nodes.

In order to support its control function, each node in the parameter base has a set of permissible operations associated with it. These operations are the ability to define a parameter, the ability to define connections between parameter base nodes, the ability to define new nodes or delete existing ones, and maintenance services, the ability to list and change many different fields in the parameter base. A user of the parameter base can perform those operations if he can LOGON to a node. Typically, only a few people have maintenance authority, and they set up the structure of the parameter base of the control parameters for the rest of the users.

The basic design of this parameter base has been quite satisfactory. It permits a few people to define the directed graph and the parameters at the nodes of that graph so that a large number of people may use the parameter base with little or no knowledge of its internals. These control capabilities have been extensively used as the means of enforcing local design methodologies at a number of IBM laboratories.

Command language

The requirements state that a significant part of every application should be written in some command language. A description of that portion is contained in the section on the structure of applications.

A consequence of this requirement is that the command language must be able to

- Communicate with alphanumeric and graphic terminals.
- Access a physical parameter base, and use values found there as command language variables. This access capability must be general, not just to a specific parameter base.
- Build the parameter interface area from command language variables (whose source may have been the parameter base.)
- Allocate resources, primarily data sets.
- Display static menus held in data sets, and dynamic menus constructed in a command language program.
- Conditionally execute statements by means of constructs such as IF, CASE, or SELECT statements.
- Support command language subroutines.
- Control the scope of symbols within and among subroutines.
- Provide powerful editing capabilities in several forms. The command language must provide an editor so that terminal users can edit external data in data sets. The editor should also be capable of editing the command language program itself at the time of execution.
- Embed TSO or CMS commands for capabilities otherwise not supported.

It must be possible for technicians who are not trained programmers to use the command language. The general type of language required is similar to TSO commands as used in CLISTS, or VM/CMS commands as used in EXECS, but neither of these languages has adequate functional capability [7, 8]. Almost every laboratory that uses EDS has made extensive use of some command language to control and sequence design applications. Several different command languages have been used. There has been a limited amount of use, thus far, of command language as an integral part of the design application itself.

All evidence points to the conclusion that a command language will be extensively used to implement and control

local design methodologies. Covergence upon one powerful command language is a strong probability.

• *Structure of applications*

The second major element in the software architecture is the structure of a design application. An intensive effort has gone into examining design applications to determine what structure they must have to meet the requirements of the user.

It is clear that applications must use the application support programs. There is little point in having a superb terminal access method if it is not used. However, use of the application support programs is the easiest of all the questions concerning the structure of applications. More difficult questions are the following:

1. What is an "application"?
   This apparently trivial question is very difficult to answer. It is important because there is a requirement to have applications implemented as modular units of work, so that various sequences of these units can easily be used in different design methodologies.

   However, there is a strong tendency upon the part of application developers to group applications together under the control of an executive program which controls some common services, perhaps data access, required by the individual applications. These groupings of applications are generally called subsystems. Thus, one might have a chip design subsystem or a module design subsystem. Such subsystems may provide fast transitions between their individual applications, thus leading to greater user productivity.

   What is wrong with the concept of subsystem? Nothing, if every user can agree that a particular subsystem embodies exactly the methodology that his location requires. Since such agreement is almost impossible to obtain, the concept of rigid subsystems is not acceptable.

   Another attempt to define an "application" is based upon the data objects that are created or destroyed. This effort was not successful due to the application designers' attempts to use the same data object for many applications in order to reduce transition time from one application to another.

   The best definition we have achieved is that of a *design task*, which is some unit of work with a determinate start and a determinate stop. Note that this definition is in user terms, not in computer science terms. The requirement, then, is that design tasks be modular entities, capable of being rearranged in time sequence by the using development laboratories.

   Design tasks are not incompatible with the notion of a subsystem, but the subsystem must be carefully designed. In general terms, each design task must determine whether the subsystem environment has been established, and if not, invoke the subsystem executive before continuing with the design task. This approach gives the necessary amount of modularity to design tasks which are parts of a subsystem.

2. How is the user environment for an application established?
   The user environment for an application differs from the system environment for an application. The system environment is determined by the operating system and the application support programs. The user environment determines how the terminal user sees and interacts with the design applications. The user environment is determined by local machine configurations and by the dictates of local design methodologies. Different user environments can be supported by the same system environment, or it may be necessary to change the system environment to achieve the desired user environment.

   A simple example of the user environment is the allocation of data sets. Users are not free to allocate data sets in an arbitrary fashion. Each location wishes to control the allocation of data sets in a manner that is consistent with site practices.

   A less obvious example is the desire to control some of the terminology in the dialogue between the application and the user. One development laboratory may wish its designers to think in terms of chips, cards, and boards. Another laboratory may wish to use a part number terminology. How can the same application conduct a dialogue with the terminal in both terminologies?

3. How is the transition made from one application to another?
   What is the mechanism for moving from one application to another in any sequence that is needed by a development laboratory? Rigid sequences which cannot be altered are not acceptable. Sequences which require many levels of display to make the transition are not acceptable. It must be possible to go directly from any application to any other.

The architecture to satisfy these requirements is the *structured application.*

Application program structure

Applications have been given a two-part structure. One part of the application is written in the command language, described previously. This part of the application establishes the user environment for the application. The command language portion is further divided into three logical portions: task selection, parameter gathering, and resource allocation and task invocation.

It is assumed that the command language portion of an application is subject to change at every location which uses

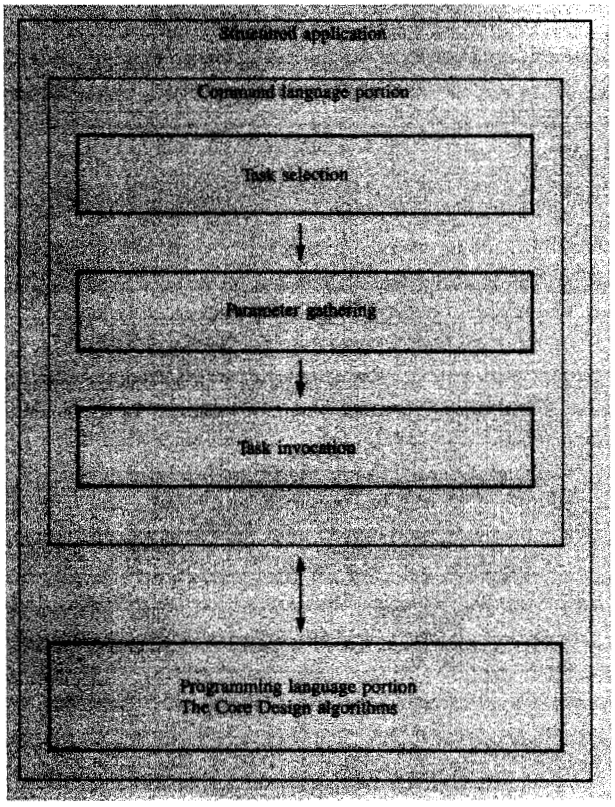

Figure 4

Application program structure.

the application. The purpose of these changes is to satisfy one or more of the requirements described above. This part of the application is executed first.

The other part of the application is called the "core" of the application. It is written in a programming language, such as PL/I, and contains the design algorithms. This part of the application is not subject to local changes.

These concepts are shown in **Figure 4**.

Task selection Task selection is the first part of the command language portion of a structured application. Task selection is the process by which a terminal user chooses the design task which he wishes to perform. For a number of practical reasons, task selection must be controllable by the site. It may be desirable to present only a subset of all possible design tasks, depending upon the progress of a part through the design cycle. Thus, a particular design methodology can be enforced by presenting only the appropriate design task at any given time.

It may also be desirable to limit access to certain design tasks to specific people. In theory, every user of the design system could have different menus of design tasks presented.

Note that the mechanics of task selection are very closely connected with the definitions of applications and subsystems. The design tasks may be applications or subsystems.

Task selection is done by means of menus, presented by the command language. Part of the contents of the menus may be parameter values. The using location can then change task invocation by changing parameter values, by changing the menu definitions, by changing the command language programs which present the menus, or by any combination of these factors. Great freedom exists to make useful changes to task invocation.

Parameter gathering The next logical division of the command language program is parameter gathering. The parameters may control resource allocation. The principal source of the parameters is the parameter base. Values from there are considered the initial defaults. The terminal user is normally given the opportunity to confirm the default values or to change them.

A particular site can change this scenario to meet its local requirements, by setting appropriate values for the control of parameters. Thus, a site may lock the values of certain parameters and perhaps not even display them to the terminal user.

Task invocation The next function performed in the command language program is resource allocation. Such resources include all the data and external temporary storage.

No data set allocation is to be done in the core of the application, that is, in the programming language. Data set allocation *must* be done in the command language portion of the application.

There are some necessary exceptions of this "pure" view of allocation, and they generally pertain to unpredictable situations. Error conditions which require additional data set allocation may arise. The terminal user may choose an application option which requires additional allocations. It is generally not practical to pre-allocate all the resources which might be conditionally needed. Therefore, data sets can be allocated from the core of an application, but *only* by means of returning to the command language and performing the allocation there. Hence, all allocations, both predictable and nonpredictable, are treated uniformly.

This is the place in the application where many of the terminology problems can be resolved. The command language function can transform external data names, such as CHIPA, into actual data set names, and perform the necessary allocation. Catalogue searches and other forms of name resolution can take place here. The goal is that this

part of the command language function should be the *only* place in the application where the association between user names for data and actual data set names is known. The core of the application accesses the data only by means of a fixed symbolic name. This architecture avoids many problems inherent in the common practice of coding data set naming conventions directly into the core of the application.

If a site wishes to change the allocations, they can readily be found, understood, and changed. If a site wishes to use a particular data set naming convention, the proper code can easily be put into the command language program.

After all resources have been allocated, the core of the application is invoked. The core may run to completion and then return to the task invocation program, or there may be two-way communication between the core and the task invocation program before the core completes execution.

Upon final return of control to the task invocation program, resources are freed. Note that the terminal user can be involved in this process, perhaps specifying the disposition of one or more data sets.

The full value of application structuring cannot yet be evaluated. All EDS interactive and foreground applications have been converted to have a command language portion and a programming language portion, and to access parameters only in the command language portion. However, only a few applications have been fully structured in the sense that all the functions discussed in this section are properly placed in the command language portion and programming language portion of the application.

We have seen much user benefit from initial structuring. This first step allows great freedom in replacing command languages and parameter bases. However, more applications must be fully structured before we can completely assess this concept.

Assessment

The architecture for interactive applications described here is not revolutionary; it is generally an example of sound design, with careful partition of function into modular parts. The most novel feature is the use of a command language portion for each application.

While our experience with this architecture is still limited, our initial impressions have been favorable. We have been able to use the modularity to our advantage, upgrading certain system services with little or no impact upon the applications.

This software structure has been available at IBM development laboratories for a relatively short time, but a few trends about its use can be observed.

The smaller laboratories use the system as shipped by EDS and will continue to do so. They do not have the resource to spend in making changes, even though the changes are now fairly simple.

Many laboratories have changed the menus that present the selection of design tasks. Some EDS tasks have been removed, and locally written design tasks added. Menu sequences and hierarchies have been changed. This type of local alteration will probably continue to increase as the dependence on local tools or vendor tools increases.

A more significant change is the replacement of the command language and the parameter base at a number of laboratories. Modularity and good interfaces make these replacements feasible, and a number of laboratories have done so. In many cases, the replacement parameter base has very little similarity to the EDS parameter base. The decision and control aspects of the EDS parameter base, supported by the graph structure and controlled parameters, have been replaced by programmed logic in the command language. This approach is fundamentally different from that planned by EDS, but it works just as well! The applications see only the parameter interface area (PIA) and have no dependence upon the source of those parameters.

So far as we know, no laboratory has replaced either of the EDS terminal access methods. Such replacement is theoretically possible, but the labor involved would be very great. However, other terminal access methods are used during the same terminal session that uses EDS terminal access methods. Such use is more practical because the EDS terminal access methods are deleted when not in use, thus making more storage available.

We feel this software structure provides a good environment for interactive and foreground applications. Initial reports from the IBM development laboratories involved indicate that this structure is meeting their requirements.

Future implications

There are a number of follow-on activities in progress to complete or extend this system architecture. Several of these activities are briefly mentioned in this section.

- *Batch jobs*

The focus of this paper has been upon interactive and foreground applications, because that is the focus of much of the new application development. However, any design automation system still contains a number of batch applications, which use a great deal of computer resource. Board wiring or test generation are examples of such applications.

A good way to apply much of this architecture to batch jobs is to insist that every batch job be submitted from an interactive terminal session, whose purpose is to gather parameter values and generate the JCL needed to execute that job at a particular location.

This architecture provides the tools needed to implement this approach, and most of the users of EDS have already employed them.

• *Data management*

There is one important part of the system architecture that has not been described—data management. The lack of uniform data management services for all applications has been recognized, and studies of data management are an ongoing effort which will occupy considerable time. Convergence of many different applications to common data management services will be a particularly difficult effort. Standard IBM database products such as DB2 and SQL/DS are being examined for their relevancy to the requirements of design automation systems.

• *Distributed processing*

One of the goals of this system architecture is to create an environment in which recent advances in computer science, such as distributed databases or professional work stations, can be easily incorporated. Let us briefly consider how this architecture fits with several views of distributed processing.

One implementation of distributed processing is called by some people the department computer. In this implementation, one small computer services the terminals of a department of people, perhaps four to ten. The computer itself is architecturally equivalent to a much larger processor. Examples of such small computers are the IBM 4300 series, the 4331, 4341, 4361, and 4381. These computers offer outstanding price/performance, are physically small in size, and can be installed in normal office environments. These computers can run the same operating systems as do the larger processors. Therefore, the system architecture described in this paper applies very well to the department computer.

More interesting questions can be asked about the professional work station, roughly defined as a small general-purpose computer which drives one or more terminals and which is used by one person at a time. An example might be an IBM Personal Computer with suitable terminals. The professional work station is generally connected to a larger "host" computer. Several host computers may be interconnected. In crude terms, the computing system is a network of professional work stations and host computers.

There is a conceptually attractive way to map the software architecture described in this paper onto this distributed processing system. Think of the application support programs as a set of processes. Replace the CALL interface to the application support programs with a more general method of invocation, one which provides interprocess communication. This communication must be able to link two processes which execute in the same computer, and two processes which execute on different computers. The interprocess communication programs must know where the process executes, but the design applications must be insensitive to such considerations. The communication programs are the key element in this design. Finally, assume the existence of a command language program that will cause processes to be started in any computer in the network.

A software architecture of this type provides a number of useful capabilities for distributed processing. Some application support programs can reside at the host computer, rather than at the professional work station. The exact residence of the support programs can be controlled by an IBM laboratory by editing the command language function that starts the processes. For example, the parameter base and the parameter base access methods could execute upon the host computer, and the resulting parameter interface area (PIA) be sent to the professional work station. Thus, one parameter base can serve the whole network and difficult problems relative to synchronization of distributed parameter bases can be avoided. Another attractive possibility is to regard the command language portion and the programming language portion of a structured application as two cooperating processes. Then the command language portion of some design applications could run on the professional work station and the programming language portion, which might require a great deal of computing power and large data resources, could run on a host computer.

Data distribution must also be considered. The current software architecture can accommodate many aspects of data distribution rather easily. Data allocation is isolated into the command language portion of the application, a portion that is easily changed or extended. It should be easy to have the allocation phase invoke a data server, a process which would typically reside on a host computer and would transmit data to other computers in the network, including the professional work stations. There is no need to change the core of the application, which can remain unaware of the location of the data obtained and allocated in the command language portion of the application. Data storage can be centralized in a few large databases, thus avoiding problems in synchronizing a number of distributed databases.

This brief description is incomplete and has ignored a number of very difficult questions. Should every interface in the software system be converted into interprocess communication? Are there not some application support programs, such as terminal access methods and the terminal monitor program, that must inherently execute in the computer that is driving the terminal? What is the efficiency of the more complex form of interface when measured against the simple CALL interface? Is the additional complexity justified when both processes are in the same computer? Regardless of a number of questions, the distributed processing architecture described here appears to be a good solution. The present software architecture maps onto it easily. It seems amenable to fairly easy implementation. Migration of existing design applications should be relatively easy. It is an architecture advocated by a number of different groups, because it applies to many kinds

of application system, not just design automation. EDS is not currently committed to this architecture, but it is the favorite among several contending architectures.

Acknowledgments

The work and ideas reported here are the product of many people, both EDS users and EDS developers. Special recognition must be given to the technical contributions of L. P. Biskup for early design of the terminal manager program and for further insights into application structuring, to Harry Halliwell for innovative work on the parameter base and the command language, and to W. M. Amaro for much of the current implementation. We are also grateful for the managerial encouragement of A. J. Emma and R. A. Rasmussen.

References

1. P. W. Case, H. H. Graff, and M. Kloomok, "The Recording, Checking and Printing of Logic Diagrams," *Proceedings of the Eastern Joint Computer Conference,* Philadelphia, PA, 1958, pp. 108–118.
2. P. W. Case, H. H. Graff, L. E. Griffith, A. R. Leclerq, W. B. Murley, and T. M. Spence, "Solid Logic Design Automation," *IBM J. Res. Develop.* **8,** 127–140 (1964).
3. P. W. Case, M. Correia, W. Gianopulos, W. R. Heller, H. Ofek, T. C. Raymond, R. L. Simek, and C. B. Stieglitz, "Design Automation in IBM," *IBM J. Res. Develop.* **25,** 631–646 (1981).
4. A. J. Blodgett and D. R. Barbour, "Thermal Conduction Module: A High-Performance Multilayer Ceramic Package," *IBM J. Res. Develop.* **26,** 30–36 (1982).
5. Donald P. Seraphim, "A New Set of Printed-Circuit Technologies for the IBM 3081 Processor Unit," *IBM J. Res. Develop.* **26,** 37–44 (1982).
6. "General Methodology and the Proposed Standard," *Computer Graphics* **11,** No. 3, II-1–II-117 (Fall 1977).
7. *OS/VS TSO Command Language Reference,* Order No. GC28-0646, available through IBM branch offices.
8. *IBM Virtual Machine Facility 370: CMS Command and Macro Reference,* Order No. GC20-1818, available through IBM branch offices.

Received December 1, 1983; revised April 19, 1984

Richard L. Taylor *IBM General Technology Division, East Fishkill facility, Hopewell Junction, New York 12533.* Mr. Taylor is a senior engineer working on design and implementation of system services for design automation applications. Since joining IBM in 1957 at Poughkeepsie, New York, he has held numerous technical and managerial assignments in design automation and machine development. Mr. Taylor received his S. B. in 1953 and his S. M. in 1955, both in electrical engineering, from the Massachusetts Institute of Technology, Cambridge. Mr. Taylor is a member of Eta Kappa Nu.