THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 59, No. 9, November 1980
Printed in U.S.A.

Designer's Workbench:

The Programmer Environment

By P. H. McDONALD and T. J. THOMPSON

(Manuscript received April 16, 1980)

Designer's Workbench, based on the PWB/UNIX operating system, is a software system that provides a high level interface to the circuit design aids used within Bell Laboratories. As well as providing a convenient environment for the use of these applications, a primary goal in the design of DWB was to provide a convenient and versatile environment in which programmers could work. This paper describes the objectives that were defined to reach this goal. The approach was to simplify the addition of new capabilities to the system, realizing that a viable design-aids system must be able to expand with the increased demand for and complexity of the applications it supports. We discuss in detail the methods and mechanisms used in the development and implementation of the system. Although some of these are specific to the functions required by DWB, the general philosophies that are emphasized can be applied to provide a programming environment for any software system.*

I. INTRODUCTION

Designer's Workbench is an interactive front end, using the PWB/UNIX operating system,[1,2] to a variety of circuit design-aid application programs. The system is designed to be convenient to use at all levels of user expertise. This is accomplished primarily through a consistent user interface and extensive on-line tutorials. A discussion of the user-oriented design philosophy and features upon which the system is based can be found in Ref. 3. As well as being user-oriented, however, DWB is also *programmer*-oriented. This means that programming simplicity, consistency, and versatility are as important in the overall

* UNIX is a trademark of Bell Laboratories.

design as are issues of user convenience. The special considerations given to the programming environment of DWB are the subject of this paper. We first discuss basic objectives for the environment and the properties they require. The methods and mechanisms used in fulfilling these objectives are then discussed in detail.

II. OBJECTIVES

Designer's Workbench was developed to coordinate and enhance the use of circuit design aids. Since we realized that the number of design aids to which DWB could interface was virtually unbounded, a primary goal in the design and implementation of our programming environment was to simplify the addition of new applications; many of the objectives discussed below are directly related to this goal. First, this emphasis simplifies our own programming in improving and expanding the system. Second, it allows other people to take the framework provided by DWB and tailor its use and application to their own particular requirements. With this capability, DWB can become a generalized interface to almost any set of applications programs.

2.1 Control framework

The way in which applications are executed must be easily expanded and modified. For example, when adding a new capability similar to existing ones, it is often tempting to copy existing code, make the control and data changes necessary, and install that as the implementation of the new capability. This, however, wreaks havoc with the maintainability and modifiability of the system. In a good control framework, the addition of a new application should require only the coding of information and programs that are unique to that application, and should not require changes that have to be reflected in a number of modules that use similar code.

2.2 Distributed data base

Each circuit stored under DWB has its own data base, including the interconnectivity of its components and the input vectors for simulation and testing, which is used to generate the input for running applications. Almost every application run from DWB also makes use of a *library* of data that exists on another computer. To eliminate the need for storing and maintaining copies of these libraries on DWB, to reduce the amount of information which must be sent from DWB to the remote machines, and to reduce the intricacies and interdependencies of the data required by a complete data base, we have chosen to leave most of these libraries on the remote machines. We therefore have a distributed data base in which the circuit data base kept on DWB refers to other data bases on other machines. However, to reduce

the number of (usually expensive) jobs which must be submitted to the computer network, it is desirable for DWB to have some knowledge of these remote libraries so that simple consistency checking can be done on the circuit data bases.

2.3 Data framework

The structure and format of data in our programming environment should be easily accessed, modified, and expanded. To simplify the provision of data required for new applications, interdependent and redundant data should be avoided. The requirements of a distributed data base indicated that the amount of data that is stored and maintained on DWB and must be transmitted to other computers should be kept to a minimum.

2.4 Data translation

To provide the input necessary to run applications on other machines without modifying the application programs, DWB must translate its data to the required format. This was necessitated by the large number of design-aids application programs that use either completely different circuit description languages or require special dialects and restrictive coding conventions of a supposedly "common" language. To eliminate redundant data and the duplication of programming effort, the implementation of these translators should be coordinated to take advantage of common data and program structures.

2.5 Customization

Each application to which DWB provides an interface has a number of user-dependent and circuit-dependent variations that require some customization of the jobs submitted by DWB. For example, a typical batch application would need variations in line and time limits, depending on the size of the circuit being processed. The implementation of this customization mechanism should allow the addition of new variations and the manipulation of existing ones without reprogramming. To simplify their addition and use, these variations should only have to be specified once, to be used automatically therafter. To minimize the effort required to enter them, the most typical variations should be automatically provided by a defaulting mechanism. Naturally, there must be a mechanism to override these defaults for individual user or circuit dependent customization.

2.6 Software development

Since the development of DWB involved up to 10 people at two locations, it was extremely important to control the organization and maintenance of the software. We did not want to hinder the ability to

experiment with and create new software; however, early production use required careful control of the testing, installation, and documentation of our software. Our operational procedures should provide all these capabilities in a simple framework.

III. METHODS

To achieve the objectives discussed above, DWB was designed in a top-down modular approach that emphasized expandable control and data frameworks. The UNIX operating system, known for its simple and consistent organization, was chosen as the basis for our programming environment. The large number of PWB/UNIX software utilities significantly reduced the programming effort and contributed to modularity. The various methods and mechanisms used in specific implementations of the general objectives discussed above are described below.

3.1 Shell procedures

In implementing the "first-cut" of DWB, we wanted to demonstrate the feasibility of the overall system with a minimum of programming effort. This was accomplished through extensive use of the UNIX *shell*,[4] a command language that can be used either on-line as the normal interface for time-sharing users or as an interpretive programming language via shell *procedures*. DWB programmers have found many advantages in using the shell as a vehicle for software development. Its syntax is simple and concise, and procedures are easy to create and maintain. Debugging shell procedures is facilitated by the capability of on-line testing as well as a "verbose" mode that traces their execution. The interpretive nature of the shell which provides these features also results in substantially slower execution times. However, there is no syntactic or operational difference between the use of a shell procedure and a C program.[5] ("C" is the standard compiled programming language of the UNIX operating system.) This feature has been of crucial importance in the continuing development and improvement of DWB. Programs initially implemented in the shell for feasibility and debugging purposes can be easily rewritten in C to obtain the increased speed and efficiency of a compiled language. The programs which required such rewriting were identified through timing and statistical data gathered using utility shell procedures.

3.2 Version control

One of the most distinctive software features of DWB is its implementation and use of a "version control" mechanism that has proved invaluable in resolving the conflicting needs of production use and development work on DWB. This mechanism provides multiple versions

of the entire DWB system which are independently and simultaneously available, so that programmers can do software development on one version without disturbing production users on another.

Under the "version control" mechanism, five separate versions of DWB are named DEBUG, TEST, RELEASE, CURRENT, and OLD. The DEBUG version is used by programmers for any debugging and program development. The TEST version contains new features and modifications that have been debugged but are not reliable enough to be put into production use. "Friendly" users can use the TEST version to try out these new features and provide valuable feedback and further testing. The RELEASE version is used for final regression testing just before a new release or "issue" of DWB is made. When a release is made, the changes which have accumulated in the TEST version are transferred to the CURRENT version, which is the default system for most production users. The OLD version is a copy of the CURRENT version before the latest release, and is always available for users as a backup in case any problems develop with a new release. In this way, we provide a dynamic development environment (the DEBUG and TEST versions) as well as a reliable production environment (the CURRENT and OLD versions). The PWB/UNIX Source Code Control System (SCCS)[6] is used to control all formal releases of DWB so that we can reconstruct and maintain any prior release of the system.

Debugging and development are greatly facilitated by the existence of the DEBUG version, a copy of the entire DWB system in which programmers are free to experiment. This eliminates the need for writing software drivers or other testing devices, since new programs can be tested within the complete system. Naturally, good communication and coordination are required to prevent any testing conflicts between programmers.

The versions of DWB are under the strict control of a "program librarian." Shell procedures are used to insure the consistency and reliability of the operations performed by the librarian. To work on a particular module in the DEBUG version, for example, a programmer must notify the librarian of this desire. The librarian, through the use of a shell procedure, then performs the manipulations necessary to create a separate DEBUG version of that particular module. Similarly, the librarian is notified when the module is ready to be installed in the TEST version, and a shell procedure is used to perform the operation.

It may seem to be a waste of storage to keep five complete copies of an entire software system. Actually, storage is kept to a minimum by using UNIX "links" to link common programs between versions. For example, if program edit is identical in all five versions, it is stored only once with "links" to the directories representing the five versions. Both source and executable files are managed in this way.

A more complete discussion of the "version control" mechanism and its implementation can be found in Ref. 7.

3.3 Data organization

UNIX data files are normally ascii text files, and most utilities rely on and take advantage of this fact. DWB data files follow the same convention, even though binary storage of data might reduce storage requirements and increase execution speed slightly. These disadvantages are far outweighed by the improved consistency and modularity which text files provide.

3.3.1 File and directory structure

The organization of data on DWB is based, at both the user and system levels, on the UNIX file system. DWB user files are organized according to projects and circuits, each project containing several circuits. Under the DWB root directory, there is one directory for each project; its name identifies the project. Under each project directory, there is one directory for each circuit; its name identifies the circuit and it contains all the data files for that circuit.

System files are organized in the sys directory under the DWB root. Under sys are subdirectories containing various categories of system files, illustrated in Fig. 1.

3.3.2 Data access

The programming access to various files within the system is simplified by the structure described above. Equally important is the requirement that DWB users, when executing the programs of DWB, are in a circuit directory. When DWB is initiated, the user is asked for the project and circuit on which work is to be done. After checking that the user has permission to work on that particular project, DWB places the user in the appropriate circuit directory via the chdir (change directory) function.

The fact that a user is in a circuit directory makes it very easy for programmers to access user files without worrying about full UNIX path names. For example, each circuit's description is kept in the form of Logic Simulation Language (LSL)[8,9] statements in a file called source.lsl under the circuit directory. Any program that wants to access it need only specify source.lsl, since the current directory will determine the circuit.

Executing from a circuit directory also makes program access to system files consistent. Relying on the fact that circuit directories are two levels down from the DWB root, system files can be accessed via a *relative* path name. For example, files under the sys/data directory

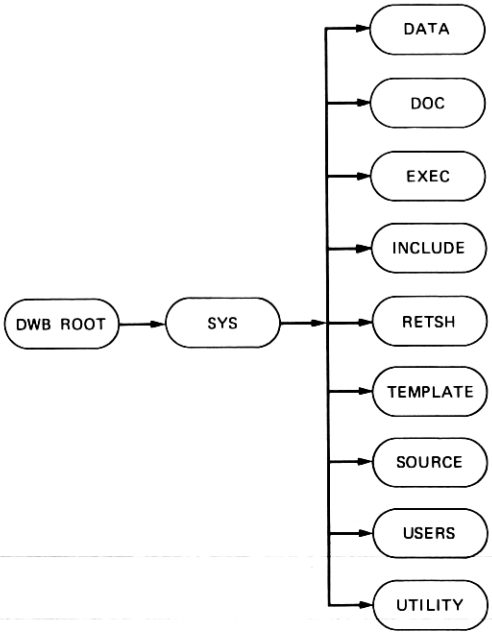

Fig. 1—Subdirectories: data contains miscellaneous system data files, doc contains on-line documentation, exec contains all executable C and shell programs, include contains C program including files, retsh and template contain files which drive the batch submission capability, source contains C program source files, users contains files for user information, and utility contains C and shell programs required for system operation and maintenance.

can be accessed from any circuit directory by using the relative path name ../../sys/data. (The notation ".." in a UNIX file name signifies moving up one directory level.) This is illustrated by Fig. 2, where proj.1 and ckt.1 represent arbitrary project and circuit directories. As you can see, the name ../../sys/data/edit.table will access the file edit.table from *any* circuit directory.

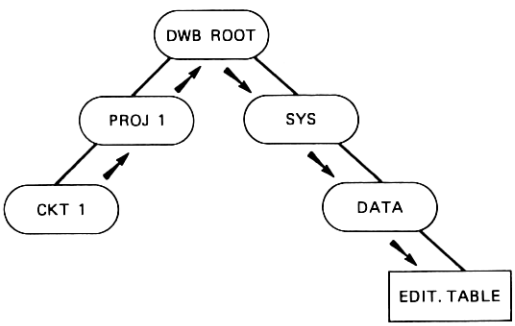

Fig. 2—System file access.

The directory structure and method of accessing files described above have a very distinct advantage in that the DWB root directory can be changed without having to change or recompile *any* DWB software. This was crucial in our initial development programming environment on the "public" UNIX machines in the Holmdel Computer Center. On these machines, user directories (of which DWB's root directory was one) were subject to periodic change as users were moved between computers and file systems for purposes of load balancing. Our implementation rendered a programming environment immune to such changes.

3.4 Data-driven control and processing

To provide a flexible control framework, DWB uses data-driven control and processing. This means that the operation of many DWB programs is defined and can be altered by the contents of system data files. For example, the edit program allows the user to edit specific files in a circuit directory by specifying file *keywords*. The mapping of file keywords to the actual UNIX file names is provided by a data file called edit.table in the sys/data directory (see data access example above). Each line of this table contains two words, the file keyword and the actual UNIX file name. This table is a text file that can be manually edited if additional keywords, either as aliases for existing files or as new files, need to be added to the capabilities of the edit program. In this way, the most typical additions to the edit program are handled quickly, easily, and without program modifications or recompilations.

The major sections of DWB which have been implemented using this data-driven approach are discussed below.

3.4.1 Table-driven monitor

DWB is a single-word, command-oriented system with automatic prompting for any additional information required. For example, to edit the LSL (circuit description) file, the user merely types the command edit. DWB will come back with the question What do you want to edit? to which the user will answer lsl. DWB then asks What editor do you want to use? to which the user can answer either unix (for the UNIX text editor) or easy (for a simplified line-oriented editor). The user is then put into the desired editor.

Each single word command corresponds to a particular DWB program. This mapping is provided by a "monitor table" stored in the sys/data directory which serves to drive the monitor program. The section of this table which specifies the edit command is shown below:

```
.cm     edit 2
.al     qed
.fc     edit.x
```

The format of the monitor table is based on the philosophy of the UNIX text formatters TROFF and NROFF,[10] where each line begins with a "directive." The .cm directive in the first line above indicates the definition of a command, which in this case is the edit command. The 2 which follows is the minimum number of letters required to recognize the command. Within this limit, unambiguous abbreviations of commands will be recognized. The .al directive defines an alias for the command which has just been defined. The .fc directive specifies the C program (in this case, edit.x) which should be executed for that command. Shell procedures are specified with the .fs directive. For procedures which must be executed within the monitor program itself, such as logging off or changing circuits, the .fi directive is used.

If the command typed by the user is not found in the monitor table, a prompt (specified in the table with a .pr directive) is printed. A typical example would be:

```
.pr "Command not recognized, please re-enter →"
```

If the user's next try is unsuccessful, a list of all available commands is printed. On the third consecutive error, the user is advised to use the help command. More information on the help or "tutorial" mechanism follows.

Using a table-driven monitor, the command repertoire of DWB can be expanded by merely adding lines to the monitor table. Programmers responsible for writing new commands need not worry about linking, loading, or recompiling procedures when interfacing their programs with DWB. The monitor contains no "hard coding" for specific commands except those which must use the .fi directive. For example, a list of current commands is automatically derived from the monitor table.

Additional directives can be added to the monitor table to increase its capabilities for expansion. Currently being developed is the concept of "sub-monitors." Using the single monitor table approach above, all new commands are added to the same table and are accessible to all users. Most users, however, use only a subset of the available commands which pertains to their application of DWB. Inundating users with a list and on-line documentation of every available command makes it very difficult (especially for new users) to find those commands which are useful for a particular application. Therefore, we wish to implement submonitors which contain subsets of commands specific to each application. Each submonitor will have its own table of commands, and users will be able to switch between submonitors as they work on different applications.

3.4.2 Textfile-driven tutorials

New users of a complex software system must normally sift through a detailed user's manual to discover what facilities are available,

learning the command names and the various arguments associated with each. DWB simplifies this process for users by providing immediate help for every question, as well as complete on-line documentation. Whenever a user types help, DWB will first respond with a short message specific to the question at hand. If the user wants more information, DWB will then enter a "tutorial" mode in which information is available in a keyword-oriented fashion (e.g., by typing commands, the user gets a list of commands). Through this mechanism, users can be guided through a tutorial on the complete operation and use of DWB, obtaining as much or as little information as is desired.

To make sure that the "help" mechanism is always active, it must be included in every program that requests input from the user. This is provided by a C function (also available from the shell) called ask, which accepts all user input and automatically catches and processes special keywords such as help. Arguments provided by the programmer to this function specify the text of the question to be asked and the immediate help message. With a minimum of programming effort, this single input function provides a consistent user interface, including a look-ahead feature that enables a user to enter more than one command or response at a time.

To help insure that on-line documentation is up to date, it must be convenient for programmers to enter and modify its text. The tutorial mechanism is therefore controlled by a series of "textfiles," which contain the on-line documentation and which are easily edited and modified by programmers. Each "textfile" contains a list of messages, each preceded by the keyword to which it applies. For example, the following excerpt from a textfile defines the message for the keyword "circuit":

```
~31~circuit~
The "circuit" command allows you to switch to another circuit
(within the same project) without logging off of DWB. After typing
"circuit", you will be asked for the name of the circuit on which
you want to work.

See also:    project
```

Every time the user enters a keyword in the "tutorial" mode, the message corresponding to it is printed. To eliminate repeated searching through the textfiles, an internal "directory" is constructed first, and is used thereafter to locate each message.

In addition to the general textfile which contains basic information on the commands and capabilities of DWB, each application, such as LAMP,[8,9] has its own textfile which contains more specific information on that application. This information can be accessed by using the keyword more after obtaining information on a particular application. DWB will respond by allowing the perusal of that application's textfile.

3.4.3 Template-driven batch submission

DWB provides access to many applications through the submission of batch jobs to the Bell Labs interlocation computer network. The implementation of this capability is driven by a series of *template* files which define the Job Control Language (JCL) required for each application. By using these template files to control the operation of as many DWB programs as possible, we have significantly reduced the programming effort required to add a new batch-oriented application. In many cases, a new application can be added merely by adding a single JCL template file. This implementation again illustrates the value of data-driven programming.

A batch-oriented application program can often be run in several different ways, each of which requires different JCL. DWB refers to these as *options*, and each of these has its own template. Therefore, to determine the particular template to use for a batch run, the user must specify an *application* and *option*. These are concatenated to construct the specific file name of the JCL template. Through the UNIX "linking" mechanism, we can specify two different names for the same file, hence providing a simple aliasing mechanism for application and option names. For example, consider a template file named lampcompile which contains the JCL to compile a circuit description for the LAMP application. A user can access this template by specifying application lamp and option compile. However, by creating a link to this template file called lampc, a user can also access the template by specifying application lamp and option c. The first line of each template file contains an internal code used by DWB to uniquely identify that template. Notice that in this implementation we are using both the template file names and the contents of the files to drive our software.

The existence of the template files provides a number of DWB programs with information about the applications currently available. For example, programs which require the user to enter an "application-option" pair can check the validity of those names by looking for the appropriate template file. The internal code contained on the first line of the template can then be used for all internal references and processing for that "application-option."

3.5 Parameter substitution

The JCL template files described above contain various *parameters* that allow for detailed customization of the batch jobs. The substitution of specific values for these parameters is driven entirely by the contents of the template files, and new parameters can therefore be specified by programmers merely by including them in the JCL templates. Several features are automatically provided by the implementation described below. These include: automatic query for and storage of missing parameter values, and a defaulting and override mechanism.

The values used to fill in the template parameters come from several sources, depending on the type of parameter. Some values, such as the project and circuit names, are computed and included by DWB at the time of submission. Other values come from parameter files, which are divided into two categories: *user* and *circuit. User* parameters, such as job numbers and output bin numbers, are specific to the user who requests the batch run. These values are stored in user files under the sys/users directory (see figure above). The name of the file is the same as the user's last name, hence easy programming access via the relative path name ../../sys/users/{lastname}. *Circuit* parameters, such as component library names, are specific to the individual circuits, and are stored in a file called parms under the circuit directory. The parameter fields in the JCL templates contain flags which indicate the parameter type (*user*, *circuit*, or computed), and hence determine where the value can be found.

A defaulting and override facility for *circuit* parameter values is provided by using the project-circuit directory structure. As in circuit directories, parms files are stored under each project directory and in the DWB root directory. This provides a three-level hierarchy of *circuit* parameters: a circuit level (the parms file in the circuit directory), a project level (the parms file in the project directory), and a global level (the parms file in the DWB root directory). Figure 3 illustrates this structure.

When looking for parameter values, the parms file under the circuit directory is searched first. If any values are not found, the parms file under the project directory is searched, and if any values are still not found, the parms file in the DWB root directory is searched. In this way, values not specified at the circuit level can default to values stored in the project or global level. Also circuit-level values can override those in the upper two levels. The implementation of this capability is accomplished by using the DWB directory structure and the knowledge that execution occurs in a circuit directory. Using the

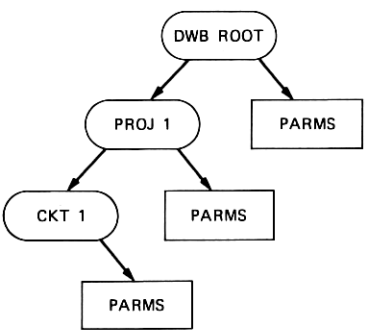

Fig. 3—Project-circuit directory structure.

relative path notation, the three files in which to search for *circuit* parameters are merely parms, ../parms, and ../../parms, no matter which circuit is being used.

When a particular parameter value is not found, the user is requested to enter its value. This value is then stored in either the circuit-level parms file or the user file, depending on the parameter type indicated by the template. In this way, parameter values need only be entered once, and are used automatically thereafter. If any values need to be modified, the change command can be used.

3.6 Libraries

DWB maintains several libraries of information. The component library contains LSL language descriptions of various components (e.g., integrated circuits) that can be referenced and automatically included in users' circuit descriptions. There are also libraries of component references. These are essentially lists of the components that exist in libraries on other computers. Using these lists, along with some utilities for their interrogation, programmers can easily perform consistency checks, verifying that components referenced by users in their circuit descriptions exist in remote libraries. To provide a framework for easy access of these libraries, as well as provide a defaulting and override capability, we again use the UNIX directory structure. Just as with *circuit* parameters, library information in DWB exists on three levels, circuit, project, and global, corresponding to the circuit, project, and DWB root directories. Under each of these directories is a directory called *libraries* which contains all the library data for the associated level. For example, the directory *libraries* under the DWB root directory contains all global library information, which can be used as the default by everyone. Under a particular project directory, a *libraries* directory contains the library data specific to that project. Project library elements override equivalent global library elements, that is, library elements with the same name. Similarly, each circuit directory contains a *libraries* directory whose contents can override equivalent project or global data. The entry and modification of library elements is handled through several utilities and simple file manipulation.

3.7 Generalized translation

The implementation of the data translations required by DWB is done in a generalized fashion which takes advantage of current compiler techniques and available software tools. The motivation for this implementation was to simplify the addition of new translators, and thereby new applications, to DWB. We have found that individual translation programs can take from six months to a year for planning and implementation. Using the techniques described below, we can add new translations in two to four weeks.

A typical compiler can be divided into two major sections. The "front end" takes an input language, parses its statements, and constructs an internal representation of its contents. The "back end" takes the internal representation and produces its equivalent in the machine code of the computer for which the compilation is intended. A "portable" compiler allows the compilation of a common language for execution on several different computers by using a single front end and several back ends, one for each target computer. The input language is always parsed by the same program, producing the same internal representation. The back end programs take this representation and produce the machine code required for a particular computer. DWB uses this "portable" compiler technique to produce different input languages for application programs from a single input language.

The circuit description language used as input to DWB is a dialect of LSL (Logic Simulation Language) known as HIWIRE.[11] The front end of the DWB translation process consists of a "table generator," which takes the HIWIRE, parses it, and constructs a relational data base. This data base contains all the information expressed by the original HIWIRE input. In addition, a postprocessor derives additional information about the circuit (e.g., determining which components are resistors according to a naming convention) which is added to the tables. The back end of the translation process consists of a variety of "formatters," which take the information from the data base and produce equivalent circuit descriptions in other languages. Some of these languages serve as input to application systems such as LASAR,[12] CAPABLE,[13] and TMEAS.[14] Figure 4 illustrates the overall process.

This system, then, provides translation from the HIWIRE language to a variety of other languages, so that a single circuit description can serve as the input to a number of application programs. Notice that, once the data base is built, the formatters can all be run off it without rerunning the "front end," thereby saving execution time.

To add a new output language to the capability of DWB now requires only the writing of a new formatter, assuming, of course, that all the information required by the new language is already present in the data base. The most dramatic expansion of capabilities is produced by writing a new front end to the process. Consider, for example, the writing of a program which translates a new language into the data base. This would allow the translation of the new language into all the languages represented by existing formatters, a considerable capability to be gained by the writing of a single program. We are currently adding a new input language which will handle analog data in just this way, using several UNIX utilities, described in the next section, to significantly reduce the programming effort required.

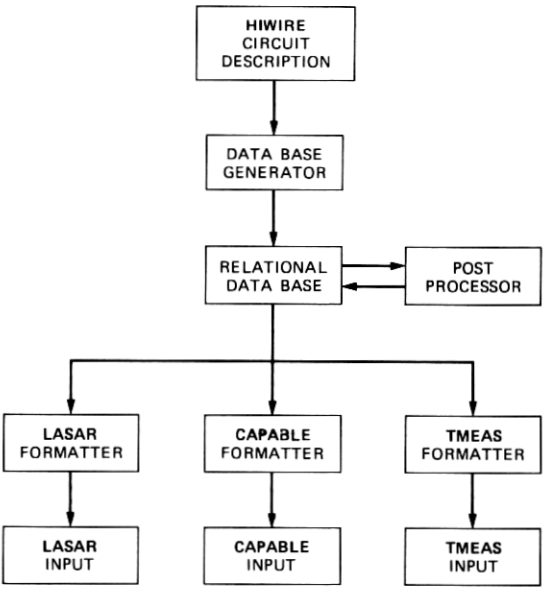

Fig. 4—Translation process.

3.7.1 Lex and Yacc

The PWB/UNIX utilities Lex and Yacc[15] are a tremendous aid in writing software which must parse and interpret complex input, such as HIWIRE and many other circuit description languages. In DWB, they have been used to write the "table generator" described above.

Lex generates a program to do *lexical analysis*, breaking its input into "tokens" (names, numbers, and punctuation). Yacc generates a program to do *grammatical analysis*, the separation of tokens into meaningful "statements." Both Lex and Yacc are controlled by sets of rules supplied by the programmer. In the DWB table generator, these rules describe the format of the HIWIRE language along with the actions necessary to build the data base.

The use of Lex and Yacc is especially important in a constantly changing development system such as DWB because changes can be made quickly and easily. For example, if the syntax of the HIWIRE input must be changed, perhaps by the addition of new statements to express new types of circuit information, we need only change the Yacc rules and recompile the program.

3.7.2 Data base manager

To create and manipulate a relational data base containing the circuit description, we needed some sort of data base manager. We

experimented with the INGRES[16] system, but found it too slow for the large amount of data and queries required. We therefore chose to write a small data base manager of our own, incorporating the features we desired and making it easy to access, change, and expand.

One of the main advantages in using relational data bases is to eliminate the storing of redundant information by arranging the data base in *third normal form*. To illustrate, consider the following table:

GATENAME	ICNAME	ICTYPE	ICLOC
GATEA	IC1	TI7400	A-10
BATEB	IC1	TI7400	A-10
GATEC	IC2	TI7474	B-10

This table may seem like a reasonable way of storing information about the gates and IC's of a circuit. However, the redundant information can be eliminated by using the following two tables, which have been reorganized in *third normal form*:

ICNAME	ICTYPE	ICLOC
IC1	TI7400	A-10
IC2	TI7474	B-10

GATENAME	ICINDX
GATEA	1
GATEB	1
GATEC	2

Since the ICTYPE and ICLOC are directly dependent upon ICNAME, they need only be stored once, in the separate table on the left. The GATENAME is then related to these values in the table on the right through the ICINDX field, which indicates the row number in the first table of the IC on which the gate lies. In addition to eliminating redundant information, *third normal form* also facilitates changes. For example, to change the location of IC1 from "A-10" to "C-10" would require two changes in the first table above, while requiring only one change in the *third normal form* tables.

An important feature of the DWB data base manager is the fact that access to the tables is independent of the order of its fields or the existence of additional fields. For example, consider the table on the left above. If, in accessing that table, a program specified the fields ICNAME, ICLOC, and ICTYPE, the data base manager would return the values from the table in that order, and *not* the order in which they were actually stored. Also, a program does not have to access all the fields in the table. For example, a program could specify that it wanted to access the ICNAME and ICLOC fields, without even knowing that the table contained a ICTYPE field. This means that data-base tables can be reordered and expanded as necessary, without having to change existing programs which access them. This built-in upward compatibility has been extremely useful in the expanding development of DWB, as new information has been added to the circuit description data base. For example, several applications have required that special

information about analog components (e.g., resistors and capacitors) be derived from and added to the data base. This is accomplished by a postprocessor (see Fig. 4) which adds extra fields and tables to the data base, and requires absolutely no changes to the existing formatters.

IV. SUMMARY

This paper has discussed the objectives and methods used in the implementation of DWB which have created an effective programming environment. The overall objective has been to create a software system that is easily expandable and maintainable. This philosophy was emphasized even in the early design and feasibility implementation through modular, data-driven, and generalized programming techniques. As a result, we have gained considerable control and flexibility in a *programmer*-oriented development effort.

REFERENCES

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," B.S.T.J., *57*, No. 6, Part 2 (July–August 1978), pp. 1905–1930.
2. T. A. Dolotta and J. R. Mashey, "An Introduction to the Programmer's Workbench," Proc. 2nd Int. Conf. on Software Engineering (October 13–15, 1976), pp. 164–168.
3. J. R. Breiland and R. A. Friedenson, "Designer's Workbench—The User Environment," B.S.T.J., this issue, pp. 1765–1790.
4. S. R. Bourne, "The UNIX Shell," B.S.T.J., *57*, No. 6, Part 2 (July–August 1978), pp. 1971–1990.
5. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, N.J.: Prentice-Hall, 1978.
6. M. J. Rochkind, "The Source Code Control System," IEEE Trans. on Software Engineering, *SE-1* (December 1975), pp. 364–370.
7. T. J. Thompson, "Designer's Workbench—The Production Environment," B.S.T.J., this issue, pp. 1809–1824.
8. H. Y. Chang, G. W. Smith, Jr., and R. B. Walford, "LAMP: Logic Analyzer for Maintenance Planning, System Description," B.S.T.J., *53*, No. 8 (October 1974), pp. 1431–1450.
9. S. G. Chappell et al., "LAMP: Logic Analyzer for Maintenance Planning, Logic-Circuit Simulators," B.S.T.J., *53*, No. 8 (October 1974), pp. 1451–1476.
10. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, Jr., UNIX Time-Sharing System: "Document Preparation," B.S.T.J., *57*, No. 6, Part 2 (July–August 1978), pp. 2115–2135.
11. D. S. Evans and L. A. O'Neill, "An Integrated System for the Design of Printed Wiring Boards," ELECTRO '76 Professional Program, Session 26 (1976), Boston, Mass.
12. *The Users Manual for the Teradyne P400 Automatic Test Plan Generation System*, Boston, Mass.: Terdyne Inc., 183 Essex St.
13. *CAPABLE 4000 Logic Simulation System User Manual*, Irvine, California: Computer Automation Industrial Products Division, 2181 Dupont Drive.
14. J. Grason, "TMEAS, A Testability Measurement Program," Proc. 16th Design Automation Conference (June 25–27, 1979), pp. 156–161.
15. S. C. Johnson and M. E. Lesk, UNIX Time-Sharing System: "Language Development Tools," B.S.T.J., *57*, No. 6, Part 2 (July–August 1978), pp. 2155–2175.
16. John Woodfill et al., "INGRES Version 6.2 Reference Manual," Electronics Research Laboratory, University of California, Berkeley, California, Memorandum No. UCB/ERL M79/43, May 1979.