N. J. Denil

A Business Language

The paper describes a language and supporting interactive system for use by the small businessman. To help the businessman-user understand and apply an application program expressed in the language, he can watch the application run in a single-step mode. If tailoring of the application program is necessary, the system guides the user by diagnosing inconsistencies in the modified program. The user controls production processing from the same user interface. In the first part of the paper, the language is described. Next some example user sessions are outlined. Finally the prototype implementation and some design issues are discussed.

Introduction

This paper describes a language for expressing business application programs. This unnamed language, which we call β in this paper, is intended primarily for expression of relatively low-volume, transaction-oriented applications encountered in small businesses. The businessman—the primary user—is to be furnished with β application programs that perform complete business functions, such as billing or accounts receivable.

The system intended for this businessman-user must provide sufficient capability to express all the application programs expected to be encountered. It must also be simple, easy to learn, and easy to operate. To meet these goals, β provides a language in which applications are expressed and also a computer-based supporting system through which the user views his application programs.

The programming community has been concerned with developing languages and techniques for making programs that are better organized and more readable. Improvements in program organization and readability have, in turn, made it possible to improve program function and to remove defects.

New high-level programming languages not only allow improved program organization and readability; they also provide data and program structures closely suited to their particular problem sets. Examples of general-purpose languages of this type are Pascal [1], Euclid [2], Alphard [3], and CLU [4]. BDL [5] is an example of a specialized high-level language.

However, because the end user is not prepared to read or to understand the programming languages in which application programs are expressed, these programs are still communicated to the end user by text and diagrams separate from the program itself. This separate description causes a number of problems. First, the description is not always effective in communicating to the user what the program does and how he may run it. Second, this documentation is expensive to produce and to maintain. Finally, familiarity with the use of the program and with its documentation does not develop in the user that expertise which would allow him to modify the application program. Instead, he must communicate his needs to a professional programmer who can perform the modifications.

Thus, improvements which have been made in the program-production process and in programming languages for use by professional programmers have not been effective in improving communications between the program and the end user. This lack of communicativeness in programs is further described by Winograd [6].

In this paper, some of the techniques for organizing and producing programs within the programming community are applied to the program-to-end-user interface. In the β language are structured program and data forms, a set of simple, user-familiar data objects, and strict data access control. The language is supported by a computing system which the user employs to examine, run, and modify β application programs.

β—the language and the supporting interactive computer system—exists as a design and in a running prototype system. However, user experience is still required to determine the extent to which β meets its objectives.

- *System outline*

The end user is to be furnished with some completed, running application programs. He will want to examine them, run them, and modify them to suit his own requirements.

A system to support the application end user must do the following:

1. Communicate what the application software does,
2. Help the user to modify the programs by calling his attention to inconsistencies caused by his changes, and
3. Handle errors resulting from improper data or program modifications.

These facilities are provided by:

1. The β language. It consists of a small number of simple, familiar (to the user) notions. The application programs are expressed in this language.
2. A machine interface through which the user views his β application program, modifies it, runs it, and examines or modifies his data.
3. An error checker, which examines each program change for consistency with the remainder of the program, flagging inconsistent items. This makes it possible for the user who is unfamiliar with the entire application program to modify it.
4. An execution and testing environment, wherein error effects are localized and within which error recovery is possible.

β and these supporting facilities build upon many notions and facilities in BDL [5]; β provides for the end user what BDL furnished the application analyst-programmer. In particular, β uses similar (but different) primitive notions, a different execution rule (describing when and how individual programs are executed), and a different splitting of graphical and textual expressions.

The execution error handling strategy embodies some of the ideas described by Goodenough [7].

- *β programs*

β programs have the following characteristics:

- They are hierarchical in nature; each process is small and simple.
- They are intended to be "examined" and "modified" by use of an interactive facility. β does not appear in a "program listing" form where one sees all the detail at one time; rather, the user "probes" and "questions" the program at a terminal.
- β programs require little or no separate documentation. What commentary is required is embedded in the β program. Changes to the β program thus require no corresponding documentation changes.
- Both the data that β deals with and the processes that perform operations on that data are represented in two-dimensional displays.

This paper is organized as follows: First, the β notions are described. Then some details are given concerning β program execution and the conditional and iteration facilities. An example user session is sketched; an application program is examined and run; then it is modified. After some remarks about β and its implementation, the paper concludes with a summary. The β built-in operators are listed and briefly described in Appendix A; this provides an indication of the level of function in the β primitives.

Fundamental notions

The notions in β are described informally, using slips of paper, files containing slips of paper, and processes that use and produce files as the physical items.

- *Data*

All data are considered to be written indelibly on slips of paper. A slip of paper may contain one of the following:

- A single number; *e.g.*, \$52.11.
- A string of characters; *e.g.*, John Doe.
- A collection of numbers, strings, or other subcollections.

A collection of numbers, strings, or other subcollections is called a "record"; each subpart of the record has a location on the slip of paper. Whenever one views that record, the subpart is placed in that location.

For example,

John Doe $52.11

---

is a record that contains a string and a number, each located in two dimensions as shown.

A *kind* of data is "defined" by specifying what it is—a number, or a string, or a collection—and what its format is. If it is a collection, the definition also specifies the names of each of its constituents. Different kinds of slips of paper have different names. A name begins with a capi-

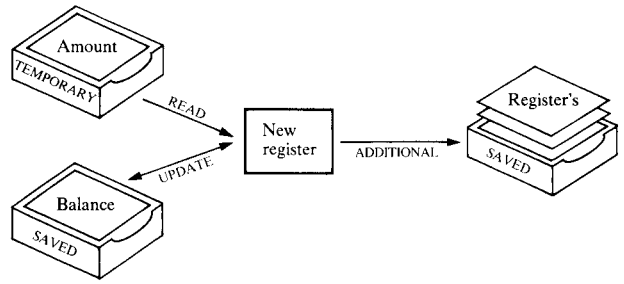

Figure 1 A fragment of the "New deposit" process.

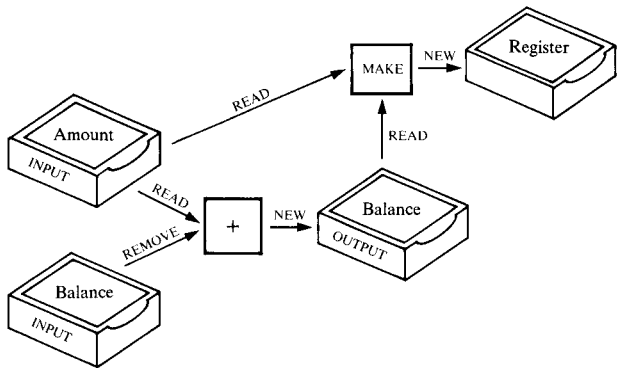

Figure 2 Expansion of process "New register."

tal letter; the rest of the name consists of lower case alphanumerics and spaces. For example, an Amount might be defined to contain one single number expressed in a certain format, say \$52.11. Then all slips named Amount would hold a single number, expressed with at most two whole digits, two decimal places, and a leading dollar sign. A "Customer name" kind might contain one string of characters, where the example string furnished delimits the maximum number of characters. A "Bill" kind might be a record containing a Name and an Amount, as illustrated by the example record above.

- *Files*

A file is a holder for either a single slip or a sequence of slips of paper. One might visualize the sequence of papers as a stack or pile of slips, where new slips are added only to the bottom (the last in the sequence) and where processing is from the top of the pile (the first in the sequence).

The file is either empty or it holds slips of paper; by looking at the file, one can tell which state it is in. Moreover, if it does hold slips of paper, one may examine each of those slips.

Each file has a name which indicates the kind of slips of paper it may contain. A file named "Amount," for example, may contain only a slip of paper of the Amount kind.

A file which may hold a sequence of slips of paper has an " 's" suffix on its name; one without an " 's" may not hold a sequence. Thus "Register's" is the name of a file that holds a sequence of Register slips of paper; the file named Amount may hold only one slip.

- *Processes*

Processes use slips of paper from some files and produce slips of paper which are placed in other files. Figure 1 shows a part of a process which is named "New deposit." In Fig. 1, files named "Amount," "Balance," and "Register's" are shown as boxes containing slips of paper. "New register" is a subprocess that uses those files. (Note that process names are formed in the same way as file names—using a leading capital, followed by lower case alphanumerics and spaces.)

Words such as "TEMPORARY," "SAVED," and "READ" are associated with various objects in Fig. 1. These words are known as "tags"; each furnishes information about the item with which it is associated.

The "SAVED" tag on the Balance and the Register's files indicates that these are permanent files whose values are to be saved from one execution of New deposit to the next. The "TEMPORARY" tag on the Amount file indicates that it may be emptied at the conclusion of New deposit; it starts out empty each time New deposit is performed.

Arrows connecting the files to the subprocess describe in general terms what the subprocess is allowed to do. The "READ" tag on the arrow pointing from the Amount file to New register indicates that the subprocess reads the slip of paper in the file, but does not further affect it. The arrow direction indicates that Amount is an input to the subprocess.

The "UPDATE" tag on the double-ended arrow connecting Balance to New register indicates that the subprocess may read the slip of paper in the Balance file, then may remove that slip of paper and replace it with a new slip. (Recall that values are written indelibly on the paper—no erasure is allowed. But processes may create new slips and throw away old ones, as is done here.) Balance is both an input to and an output from New register.

The "ADDITIONAL" tag on the arrow connecting New register to the Register's file indicates that New register may add new Register slips to those in the Register's file.

Because the arrow is directed from the subprocess, the Register's file is considered to be an output from New register.

Each arrow tag delimits the authority afforded the subprocess New register. For example, it may do no more to the Amount file than to read it; it may not remove the slip of paper in Amount. On the other hand, the subprocess need not use that authority on every invocation. Thus, a particular invocation of New register need not update the Balance file, and it need not produce any new Register slips in the Register's file.

• *Inside a subprocess*

Figure 2 shows the files and subprocesses making up the New register process. Inside New register, each file box connected to the process is tagged as an INPUT or an OUTPUT. The Balance file shown as updated in New deposit is treated as two separate files, one an input, the other an output. Two built-in subprocesses are used in New register. Built-in subprocesses have names that are either a special character such as the "+," or two or more capital letters. The + subprocess adds the values on the slips of paper in its two input files, and produces a slip of paper as its output. The arrow tags again specify the effect on the files. In particular, the REMOVE tag on the arrow from Balance indicates that the slip of paper is removed from the file. Hence, the input Balance file is emptied by the + subprocess.

The arrow from + to the output Balance file is tagged "NEW." This indicates that the slip of paper resulting from + is to be placed into the originally empty Balance file.

The MAKE subprocess produces a record slip of paper from slips that are its subparts. (For this to be a legitimate β program, the Register record must be defined to consist of Amount and Balance subparts.) MAKE in this instance has its output arrow tagged "NEW"; hence, the resulting Register slip is placed in the originally empty Register file.

Built-in processes are special in that they do precisely what the arrow label says. For example, a REMOVE on a built-in process empties the file. (Processes, such as EDIT, REVIEW, or ENTER, which interact with the user are special cases in which user actions determine the precise effects that occur.)

There is one further kind of input arrow tag—CONTROL. It specifies only that the file must be nonempty before the process may be invoked. The file is not referred to in the process expansion at all. CONTROL is used to synchronize

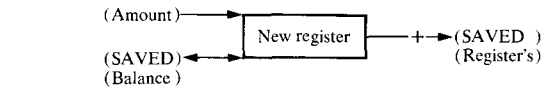

Figure 3 A fragment of the "New deposit" process of Fig. 1 as it is actually expressed in β.

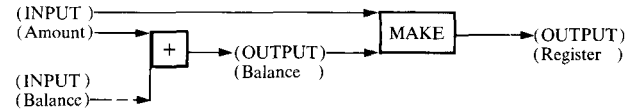

Figure 4 Expansion of process "New register" of Fig. 2 as it is actually expressed in β.

processes and is of value primarily when parallel operation of processes is allowed. (Because parallel operation is not detailed in this paper, no examples using the CONTROL tag are presented.)

• *Actual process expressions*

Figures 1 and 2 show stylized versions of the actual β representations. In the actual representation (designed to be within the capabilities of a character display device), files are shown as names enclosed in parentheses. Tags on files and processes prefix the name; those on the arrows are encoded as follows:

READ—no symbol
REMOVE—the "−" symbol on the arrow
CONTROL—the "c" symbol on the arrow
NEW—no symbol
ADDITIONAL—the "+" symbol on the arrow
UPDATE—double-ended arrow

Figures 3 and 4 show the actual representations corresponding to the stylized versions in Figs. 1 and 2.

To save space on the diagram the TEMPORARY tag is elided. A file with no tag is therefore considered TEMPORARY. Likewise, as the above encoding suggests, an arrow with no tag is considered to be either READ or NEW, depending on its use.

In the remaining discussion, the notion that a file may contain a slip of paper or a sequence of such slips is maintained. However, to shorten explanations, the term "value" is used to refer to what a file holds, instead of "a slip of paper" or "a sequence of slips of paper."

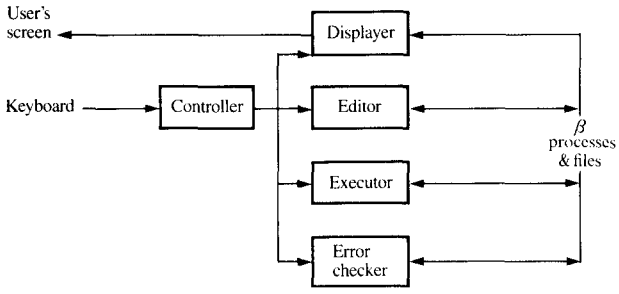

Figure 5 β system configuration.

β language and processor details

The user views, executes, and modifies processes and files using the β system shown in Fig. 5. The user gives commands to the controller; the controller dispatches one or more of the following processes to service that command.

1. Editor—modifies values or definitions under control of the user.
2. Executor—executes β processes.
3. Error checker—analyzes modifications performed by the editor. Each modification is checked for consistency with the rest of the process and file definitions. It applies ERROR tags to erroneous items.
4. Displayer—shows process expansions or file values to the user.

Controller commands allow one to select an item, say a subprocess, from the screen display and then to enter a command which is to do something to that item. The run command, for example, would call on the executor to execute the selected subprocess.

- *Process execution*

The effects that processes have on input and output files were described earlier. We consider now the conditions under which processes are allowed to be executed.

A process is allowed to be executed when it meets the following "firing rule."

1. All input files have values.
2. All output files having NEW arrow tags are empty.

This firing rule is similar to that used in Petri nets [8]. β networks can be modeled using Petri nets, but because

1. β files may hold at most one value, and
2. Output file states are examined by the firing rule,

the Petri net equivalent must contain more nodes than the β network does. As Petri nets, β networks resemble the pipeline control model described in [8].

When a β process is executed, its files are mapped into the process expansion. Then that expansion is executed. When execution of the expansion terminates, the files are mapped back from the expansion. Details of this "execution sequence," using the example of invoking New register in Fig. 1, are as follows.

1. Values in the input files (named Amount and Balance in Fig. 1) are copied to the corresponding files tagged INPUT in the expansion of New register (as in Fig. 2).
2. The firing rule is applied to the subprocesses in the expansion. Each subprocess is examined in sequence ("sequence" is described below). When a subprocess meeting that rule is determined, it is executed.
3. Step 2 is repeated until no more subprocesses may be executed.
4. Files inside the subprocess are reflected back to the corresponding files in the invoking process (to those in Fig. 1) each according to its arrow tag. The following actions are taken:
   a. READ or CONTROL tag—the file is not modified; no action is taken.
   b. ADDITIONAL tag—the sequence in the subprocess file is added to the existing sequence in the file in the invoking process.
   c. REMOVE or NEW tag—the value of the corresponding subprocess file is substituted for and replaces the original value in the invoking process file.
   d. UPDATE tag—treated the same as a NEW tag. The subprocess OUTPUT file is substituted back into the corresponding file in the invoking process.

Each subprocess in a process expansion has a SEQUENCE tag which is accompanied by a number value. This value is the ordinal position in which subprocesses are examined in step 2 of the above execution sequence. The SEQUENCE tag value is not of importance unless two processes are ready to be executed simultaneously. In that case, the one with the lower sequence number is executed.

The above execution sequence is performed when a process is "run." The user would use "run" in his daily operation to produce Invoice's.

To see each subprocess before it is performed, the user may use the "watch" command. When watching a process, the above execution sequence is started, and the process expansion is displayed to the user. Before executing each constituent process (as called for in step 2 above), that subprocess is highlighted on the display, and the processor waits for instructions from the user. The

user may examine and modify files, or he may control execution by one of the following command choices:

1. "go"—execute the subprocess, continuing the execution sequence in step 2. If the subprocess is not built in, its execution will be shown on the display so that the user can watch its execution.
2. "run"—execute the subprocess in normal execution mode.
3. "automatic"—stop displaying the subprocess. Execution continues as if "run" had been said originally instead of "watch."
4. "back"—terminate execution of the expanded process after performing step 4 of the execution sequence, mapping the files back to the invoking process.
5. "abort"—terminate execution of the expanded process. No files are mapped back to the invoking process.

A watched process terminates normally when step 2 of the execution sequence finds no more subprocesses ready to be executed.

• *Tags*

READ and REMOVE are examples of tags that are complete in themselves. Other tags, such as SEQUENCE, require a value. Some others, such as the ERROR tag, may be accompanied by a sequence of values. An ERROR tag is placed on a file, process, or arrow by the β error-checker when it has determined that that entity is inconsistent with the rest of the β program. The value of an error tag is a sequence of one or more strings, each explaining one error that the processor has determined.

Tags with values are not represented on the process expansion diagrams. One may see all the tags on an item by selecting that item and issuing a "show tags" command. On the resulting display, each tag is shown, along with its value.

• *Definitions*

It was stated earlier that data kinds are each defined. The definition of a data kind specifies the type (NUMBER, STRING, etc.), the location on the page, and the format which that kind of data will have. There may be many files holding any certain kind of data; each has the same name as the kind, yet each is distinct.

A similar notion holds for processes. A process is defined to consist of some collection of files, subprocesses, and arrows. Many instances of such a process may occur, each distinct from the other, and each containing its own private files.

In Fig. 1, for example, the three files are a part of this particular instance of New deposit. If we were to have a different occurrence of New deposit, it would have a separate, distinct set of files whose values are unrelated to those in the first occurrence of New deposit.

• *Adjectives*

Adjectives have the same form as names (a capital letter followed by lower case letters and blanks); they precede the kind name. Adjectives are required to distinguish between record subparts of the same kind. In an Invoice, for example, there may be three different "Amount" subparts labeled "Total Amount," "Tax Amount," and "Current Amount." Each is of the Amount kind. MAKE and BREAK processes for the Invoice record would use "Total Amount," "Tax Amount," and "Current Amount" as inputs and outputs, respectively.

The spatial discrimination apparent on a process expansion suffices for the β processor to distinguish files. However, adjectives may be used to help the user to distinguish lexically and to associate semantics with files. In Fig. 2, for example, we might distinguish the Balance files by calling the leftmost one "Original Balance" and the rightmost one "New Balance," using "Original" and "New" as adjectives. Both files function exactly as before: the adjectives are not used in the process to expansion file matching.

• *Matching a process to its expansion*

Input files in a process invocation are matched to the files tagged INPUT in its expansion by matching file kinds. The same holds true for the output files. But if there are two or more files of the same kind in the input, an additional criterion must be used.

The input arrows on each process are considered to be organized into a sequence; the ordinal position of each arrow is given as the value of its SEQUENCE tag. Likewise the ordinal position of each input file in the process expansion is given by the value of the SEQUENCE tag on each input file. Ambiguity in file kind matching is resolved by using these ordinal positions: the first of a number of one file kind in the invoking process is matched to the first of those of that kind in the expansion, and so on.

• *Constraints on subprocess arrangements*

The execution sequence outlined earlier could cause some subprocesses, say one with no inputs or outputs, to be executed repeatedly. To prevent looping of the process executor, at least one of the following must hold for each subprocess:

1. The subprocess has an input arrow tagged REMOVE.

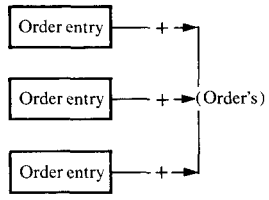

Figure 6 Parallel "Order entry" processes.

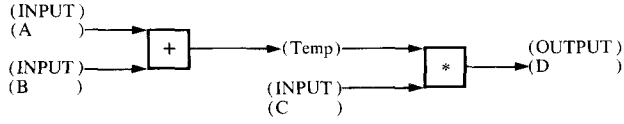

Figure 7 Expansion of "A + B * C GIVES D."

2. The subprocess has an output arrow tagged NEW.
3. The subprocess has a SINGLE tag. This tag specifies that the subprocess may be executed but once per execution of the containing process.
4. The subprocess is one of the built-in interactive processes—DISPLAY, ENTER, EDIT, or REVIEW. In these processes, the user may enter the "nomore" command. This removes the process from further execution consideration for the duration of execution of the containing process.

The error-checker in the β processor examines each subprocess for the above criteria; it tags as erroneous any subprocess that does not meet at least one of them.

- *Error handling*

Errors may occur at execution time. When a subprocess encounters an error, execution of that subprocess is interrupted. The containing process expansion is displayed on the screen along with a diagnostic message; then the process waits for a user command. Files in the displayed process contain the same values as when execution of the subprocess giving rise to the error began.

The user may modify files, abort, go back to the process that invoked the one displayed, or merely run the subprocess again. (Because saved files in the subprocess having the error are not restored to their original values, another execution attempt may be effective.)

An IFERROR tag on a subprocess can specify actions for the processor to perform after an error occurs. The IFERROR tag has a string value which consists of controller commands to be executed after the files have been restored to their previous states. Controller commands might consist, for example, of a series of N "back" commands, followed by a command to "run" some named process. This would cause execution of the named process in a process "back" N levels.

The guarantees that β seeks to provide are these:

1. All errors can be explained using β terms and notions.
2. No error short of hardware failure may cause the β file-arrow-process environment to be abandoned.
3. The user may abort a process at any time. In particular, he may abort after an error has occurred. The only effect is that modifications made to SAVED files in the aborted process remain. They cannot be backed out.

- *Parallelism*

Subprocesses may be allowed to be executed in parallel. This is particularly useful in multiple workstation environments, where, for example, the situation calls for a number of distinct Order entry processes to be executed simultaneously, each communicating with a different display terminal. Figure 6 shows an example using three such processes, all contributing Order's to one file.

The PARALLEL tag applied to a process definition indicates that its subprocesses are to be executed in parallel. (Such a tag would be applied to the process whose expansion is shown in Fig. 6.) Detailed description of parallel operation is beyond the scope of this paper. We note, however, that the execution sequence must be revised to preclude simultaneous use of files which are to be modified.

- *Calculating expressions*

When calculations involving more than a single operation are to be performed, treating each operator as a separate subprocess in the process graph (as was done with + in Fig. 2) causes that graph to expand.

An alternate method for calculations uses the built-in process "CALC," tagged by a CALCULATION whose value is the expression to be evaluated. Suppose, for example, that we have the process shown in Fig. 7. It produces file D given the files A, B, and C. File Temp is used to hold the result of the first operation. Each of these files is defined to be a NUMBER file.

This same process can be expressed in a tag:

CALCULATION A + B * C GIVES D

This tag applied to a CALC process makes it equivalent to the process whose expansion is shown in Fig. 7.

Some characteristics of the expression used as the value of the CALCULATION tag are

1. Expressions are evaluated left-to-right, with no precedence.
2. Parentheses may be used to create intermediate files that are results of subexpressions.
3. The rightmost operator in the expression must be the built-in GIVES; the file given on its right side specifies the name of the output file which is to receive the value generated by its left side.

(See the Remarks section for a discussion of a possible alternate way of expressing tag values.)

Conditionals

Much of the program logic normally performed by conditionals is accomplished in β by the process firing logic. However, the firing logic does not handle the class of conditionals that deals with value comparisons.

The CALC built in and the CALCULATION tag provide a multi-way conditional. Suppose, for example, that a process is to produce a value in file X if A = B, and a value in file Y otherwise. One may use a CALC process tagged with

CALCULATION
A = B THEN 1 GIVES X,
"yes" THEN 1 GIVES Y

As shown in this example, the CALCULATION tag has a value which is a sequence of expressions. These expressions are to be evaluated in sequence; if an expression produces an output, evaluation stops. Each expression in the example uses a THEN; the THEN operator evaluates its right side if and only if its left side has the value "yes." Otherwise it produces no value. Thus, in the example, if the value in file A does equal the value in file B, file X is assigned the value 1. The CALC process produces file X as its output. Otherwise Y is produced.

The example above has but two alternatives; any number may be used. Each is made a member of the CALCULATION value sequence.

Iteration

If one wants to perform a certain process on each member of a file sequence, he may accomplish it by performing a sequence of such processes on the entire file. Say, for example, that one wants to increment each number in a file of numbers named X's. The process labeled "+'s" in Fig. 8 stands for a sequence of processes labeled "+." Figure 8 may be considered to be a shorthand for the schematic process shown in Fig. 9.

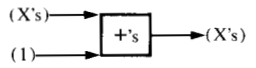

Figure 8 A sequence of processes.

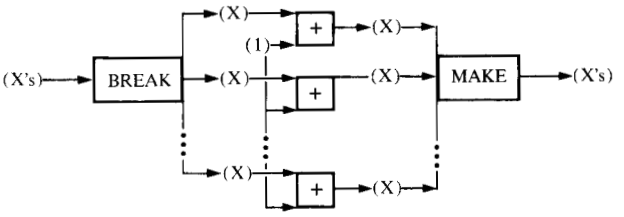

Figure 9 Expanded equivalent of Fig. 8.

In Fig. 9, BREAK decomposes the file X's into member files X, as many X records as there are in the sequence. A + process is performed on each X and on the same 1. Then the resultant X files are combined into the rightmost X's file using the MAKE subprocess. The proper ordering of constituent X files is preserved in the BREAK and the MAKE processes.

Sequences of user-defined processes may also be invoked. Suppose that a process named Pay bill is defined to use a Bill as an input and to produce a Register file output. To do the processing on a file of Bill's and produce a file of Register's, one invokes the sequence of processes Pay bill's.

Precisely which files must be iterated when performing such a sequence of processes is determined by comparing the count of the " 's" on each file in the process sequence invocation with those on the corresponding files in the process definition. Files with matching " 's" counts enter into each of the processes to be performed. The iteration is driven by the "inner product" of the input files having a greater " 's" count on the invocation than on the definition.

Using the above definition-invocation " 's" count matching rule, one might use a sequence of GIVES processes to obtain the first member of a sequence. Figure 10 shows the process sequence; Fig. 11 shows the schematic expansion. Thus only the first GIVES in the expansion in Fig. 11 will ever execute (the remaining ones will not exe-

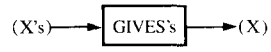

Figure 10 A process sequence producing the first value in the X's file.

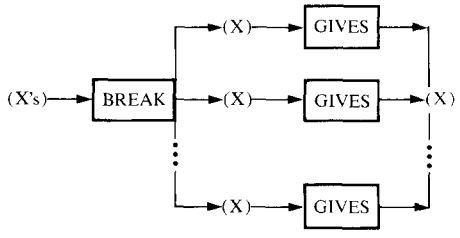

Figure 11 Expanded equivalent of Fig. 10.

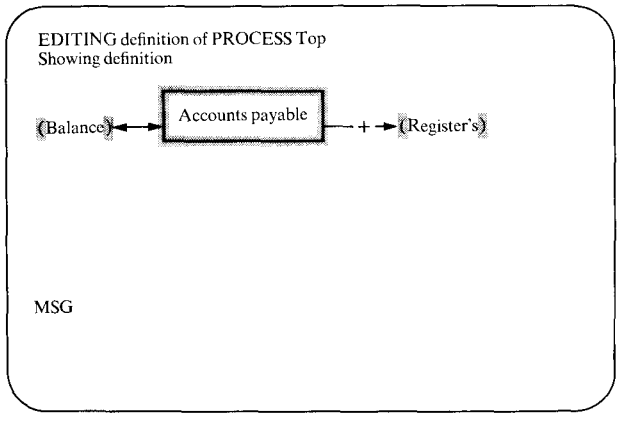

Figure 12 Examining a Top-level process.

cute because the output file is not empty). Hence, the output X will always take on the value of the first member of X's. (A better way to obtain the first member of a sequence is to use the FIRST built-in process. FIRST is described in Appendix A along with the other built-in β processes.)

An example user session

Let us examine an example application, viewing it as a user would see it. We suppose now that the β application has been handed to us for our perusal and use.

Figure 12 shows the β expression of the top-level process in a simple Accounts payable application as it appears to the user on the display screen. The screen is headed by a line stating that we are editing the definition of the top-level process—one named "Top." The second line indicates that the definition itself (as opposed, say, to attributes or values of some constituent) is being displayed. Toward the bottom of the screen is an area headed "MSG"; this is where the processor places messages.

Based only on interpretation of the β notation, Fig. 12 may be paraphrased as

> "Accounts payable" is a process that updates the value in the "Balance" file and produces additional values in the "Register's" file. No other files are affected by Accounts payable.

Highlighting the outlines of the files Balance and Register's in Fig. 12 indicates that these files have values. Similarly, highlighting the outline of the subprocess Accounts payable indicates that it is ready-to-run. In Fig. 12, highlighting is indicated by shading, whereas on a display, brightening or reverse imaging is used.

As the user types commands, *e.g.*, "go," "run," or "watch," at the terminal, command text appears in a command line which is located below the area labeled "MSG" in Fig. 12. When the user presses the "enter" key, the command is executed and the command text disappears. Figure 12 and subsequent figures show only the displays resulting from command execution. The commands themselves are not shown.

- *Examining the top level process*

Upon seeing Fig. 12, we may require further explanation as to what the Register's file is all about or what Accounts payable is supposed to do. The tell facility is used to find out.

If we select a file or process and then enter the command "tell," textual information associated with that item is displayed in the screen message area (the one labeled "MSG" in Fig. 12). The textual information displayed is actually the value of the TELL tag attached to the item selected. If we were to select the Register's file, for example, and enter the command "tell," the following text would appear in the message area:

> This file contains one Register for each Deposit that has been made and for each Check that has been written. Register's are kept in order of their creation.

The facility to attach TELL tags to any file or process in the definition provides a means for attaching documentation which is keyed to the particular occurrence of the file or process in that definition. Because the definition itself may have a TELL tag, it too may be documented.

Suppose we want to examine the Register's file. If we select the word "Register's" on the display and then enter the "showvalue" command, we will see the first Register in the sequence of Register's in the file, as shown in Fig. 13. Commands such as "next" and "item *N*" allow us to see the other Register's in the sequence.

But how do we find out the meaning of each of the numbers on the Register in Fig. 13? We can ask to see the definition of the Register; we do that by entering the command "showdefinition." The display shown in Fig. 14 results. In Fig. 14, we see the definition for all Register items. It has the name of each subpart displayed in the position occupied by that subpart. We may obtain a display having examples in each subpart position by entering the command "showexamples"; the resulting display is shown in Fig. 15.

From the displays in Figs. 14 or 15, we may ask for prose descriptions of each subpart. For example, if we select the subpart called "Name" and enter the command "tell," the following text will be displayed in the message area:

> For a Register corresponding to a Check, Name holds the name of the payee.
> For a Register corresponding to a Deposit, Name holds the string "Deposit."

We may return to the previous displays, from Figs. 15 to 14 to 13 to 12, by entering the command "back" at each step.

• *Running the application*

Suppose now that instead of looking further we decide to run the Accounts payable process. We do not know what will happen when we do; we just try it and see. We select the words "Accounts payable" and enter the command "run." Figure 16 is displayed; it contains a Deposit record and a message describing what is expected. (In the message, PF2 and PF3 refer to function keys 2 and 3, respectively, on the display.)

Suppose that we change the Deposit so that the display appears as shown in Fig. 17; then we press PF2 to enter it into the system.

Now we are confronted with the display in Fig. 18; it is inviting us to enter some Bill's into the system. Let us suppose that we do not want to go into that just yet, so we press PF1 (this is one of the options given in the message) to abort. Figure 12 is restored to the display, indicating that Accounts payable has finished and that it is again ready-to-run.

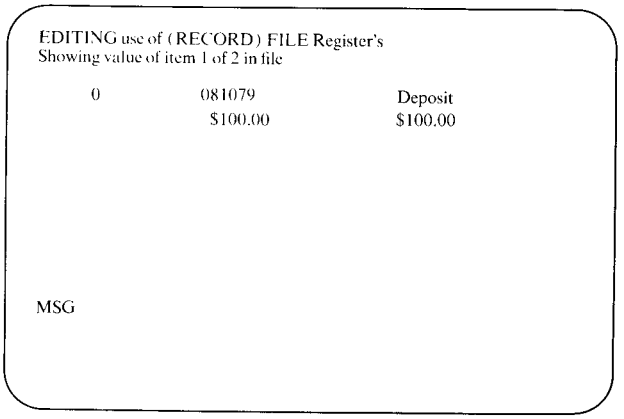

Figure 13 After entering command to "showvalue" of Register's.

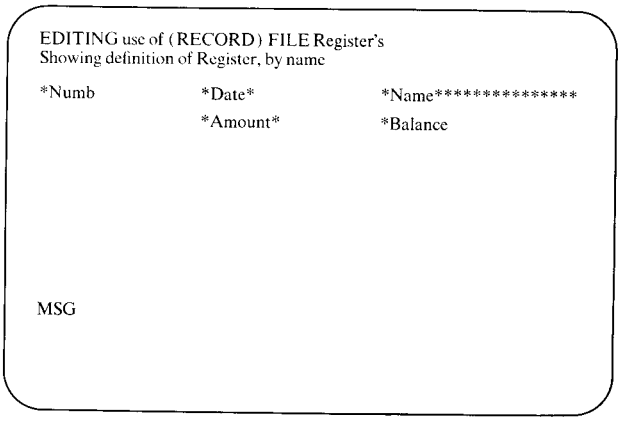

Figure 14 The definition of the Register record.

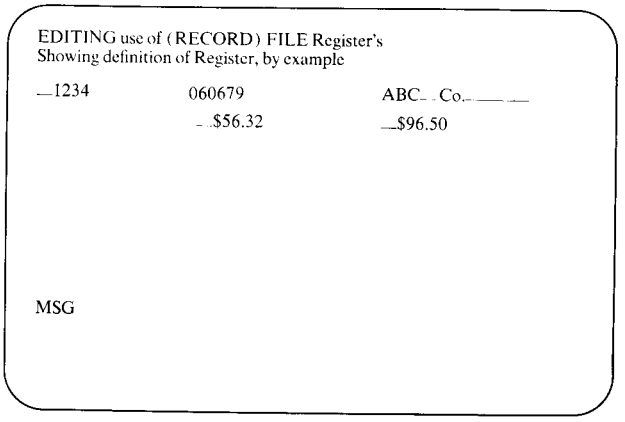

Figure 15 Register record with examples.

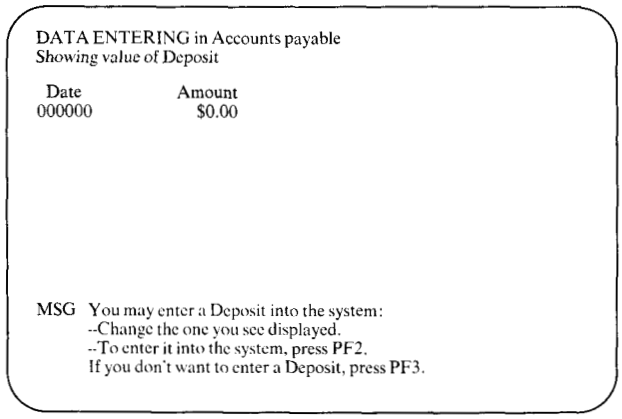

Figure 16 Invitation to enter a Deposit.

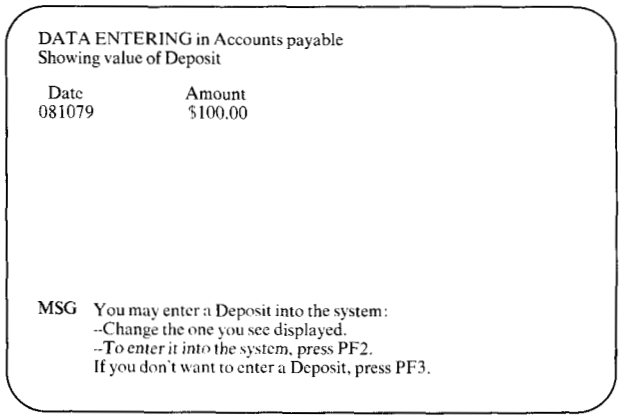

Figure 17 After changing the values on Deposit.

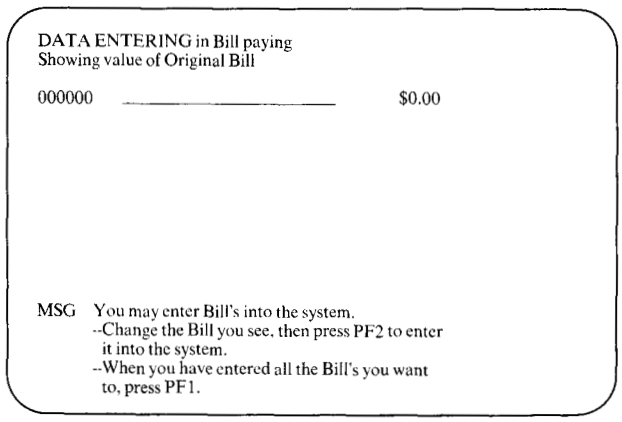

Figure 18 An invitation to enter a Bill.

If we examine the Register's file, we find that a new Register has been created; this Register corresponds to the Deposit we just made. We also find that the value in the Balance file has increased by the Deposit amount.

We have been rather blind in this execution; we did not know what would happen next. Let us try it again, only this time we will "watch" every step that takes place.

- *Watching the application*

To watch Accounts payable run, we select that process on the display and enter the command "watch." We see the display in Fig. 19; it shows us the files and subprocesses making up Accounts payable. The message says that Deposit entry is about to be run by the processor. If we enter the command "go" to the processor, it allows us to watch the operation of Deposit entry.

In Fig. 19, the Balance file is identified as both an INPUT and an OUTPUT. Register's is identified as an OUTPUT. These identifications agree with uses of those files in Fig. 12. A file named Paid Bill's that we had not seen before is shown as the output of Bill paying. If we ask about Paid Bill's via "tell," the description confirms, as its name suggests, that this is a file of Bill's that have had checks issued and hence have been paid.

Paid Bill's is a file internal to Accounts payable. The SAVED tag indicates that its value is retained from one invocation of Accounts payable to another. This file is not known outside of Accounts payable. If the invoking process, Top in Fig. 12, were to require access to Paid Bill's, it would be necessary to modify the program so that Paid Bill's would appear in the definition of Top as a SAVED file which is an output from Accounts payable.

When viewing Fig. 19 in the watch mode, we may examine or change file values. (If the application had specified a NODISPLAY or a NOMODIFY tag on any file, we would not be able to perform that respective operation.) The choices listed earlier in the "Process execution" section, namely, "go," "run," "automatic," "back," or "abort" apply here.

We choose to continue watching by entering the command "go." This causes Fig. 20 to appear. It shows us the detail of Deposit entry; the subprocess ENTER is about to be run.

If we enter the command "go" to the display of Fig. 20, Fig. 16 is shown to us. This display is produced by the ENTER process. ENTER is a built-in β process that invites the user to enter a new value into its output file; it is merely performed without further ado. After we enter a

Deposit as instructed, Fig. 21 is shown to us. Here the processor is about to run the process "Makereg."

ENTER is not ready to be executed again because Deposit is a NEW output file that has a value. But Makereg now has the necessary Deposit input file, so it is ready.

If we continue by entering the command "go" again, we see the detail of Makereg, as shown in Fig. 22.

In Fig. 22, we see that the Deposit value is used to create a new Register. In detail, as we would observe after entering each of a succession of "go" commands, the input Deposit is broken into its Date and Amount subparts. Then the Amount is added to the input Balance value to produce an output Balance. Finally, a Register is made from the Balance, Amount, Date, Name, and Number subparts. The Name and Number files are internal to process Makereg; they are SAVED files. The values in those files are to be used for each Register produced in Makereg. If we were to examine the value of Name, we would find that it contains the string "Deposit," which earlier we saw appeared on those Registers resulting from Deposits.

When execution of subprocess MAKE has been completed, the message

> No more subprocesses are ready-to-run; ready to go back.

appears. A "go" shows Deposit entry (see Fig. 21) with the same message. Finally we would see the display as shown in Fig. 19, only with the Register's file highlighted and Bill paying about to be run.

• *Modifying the application*

We can change anything in the application program; following are some samples:

1. Change a file format—say, to increase the number of significant digits in the Balance file. To do this, we extend the example number in the Balance definition to have more digits.
2. Change the message that invites the user to enter a Deposit. (This is the message shown in Fig. 16.) This message is the value of the MESSAGE tag on that specific ENTER subprocess (as shown in Fig. 20); we merely change the value of that tag to be whatever we want displayed.
3. Rework the Deposit entry process definition (see Fig. 20) so that the program can accept more than one Deposit at a time. We would change the Deposit file to "Deposit's." To test this sequence of Deposit's, we would change "Makereg" to Makereg's."

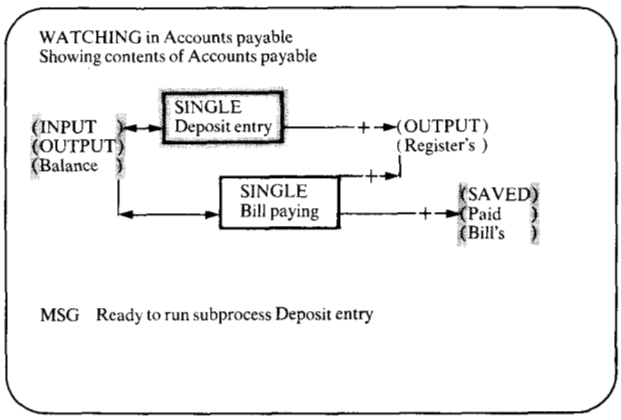

Figure 19 After asking to "watch" Accounts payable.

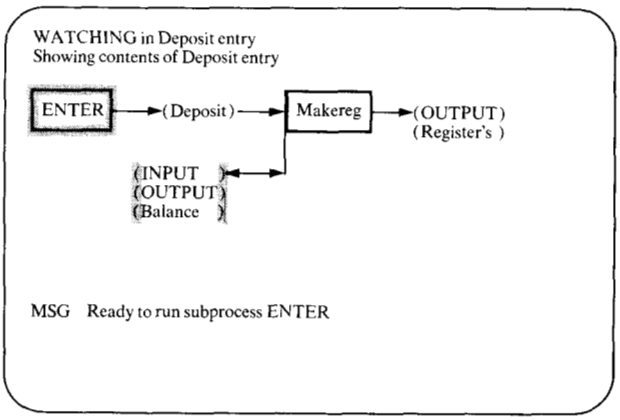

Figure 20 Now inside Deposit entry.

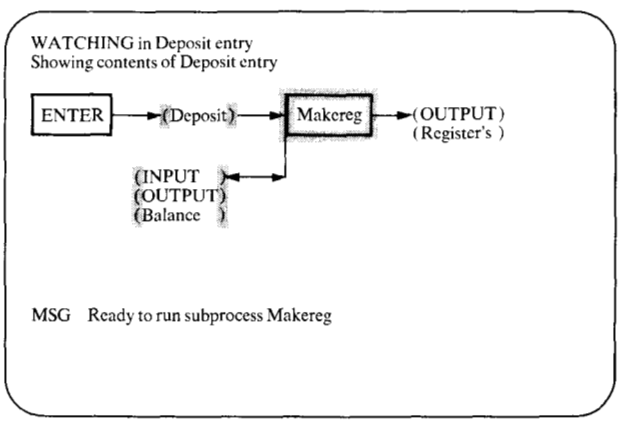

Figure 21 The ENTER subprocess has produced a Deposit.

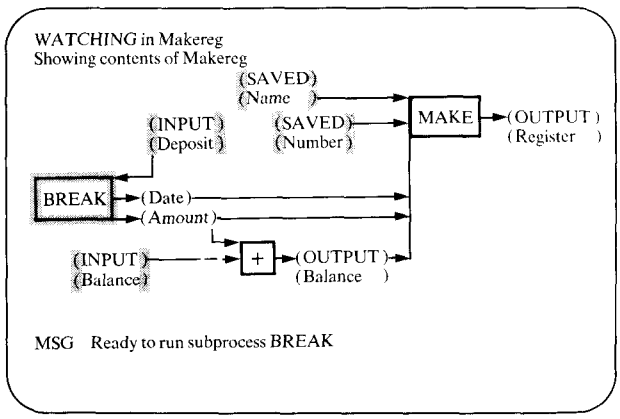

Figure 22 Having begun Makereg.

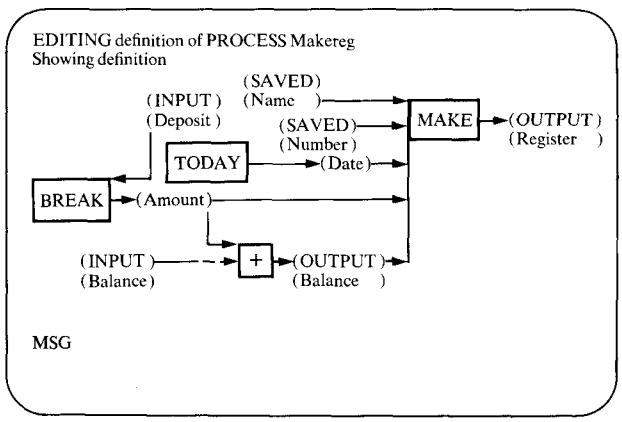

Figure 23 The modified Makereg process.

Suppose we want to change the Deposit record definition to eliminate the Date subpart. (The application will have to use the date that the Deposit is entered into the system as the deposit date.) To accomplish this, we select the name "Deposit" and enter the command "edit-definition." On the resulting display of the Deposit definition, we delete the Date subpart.

The error checker will object; it will tag the BREAK subprocess in Makereg (shown in Fig. 22) with an ERROR tag whose value is

> This BREAK produces a Date subpart which is not in Deposit.
> Suggest you remove the output Date.

We may eliminate Date as an output of BREAK; then we must produce that file as the output of some process. Suppose we produce the Date file as the output of the built-in TODAY process; we add the appropriate process and arrow to Makereg as shown in Fig. 23. This will satisfy the error checker; no more error tags will appear.

We may now test our modifications to Makereg by performing any of the following:

1. Watch Accounts payable from Top (Fig. 12), using the real application data.
2. Watch from the definition of an intermediate process such as Deposit entry (Fig. 21), which invokes Makereg. We would give a value to the Deposit and Balance files. Then we would select and watch Makereg.
3. Single-step the Makereg definition. To do this, we would give values to each of the INPUT files in Fig. 23, and then select and run each process individually.

The latter two methods do not use the application data files. This is because β recognizes the environment from which a process is invoked and keeps separate files for each environment. Suppose, for example, that a subprocess has a file, F, which has a SAVED tag. If two processes, X and Y, each invoke that subprocess, β keeps separate F files, one associated with the invocation from X, and the other with the invocation from Y. Thus, if we execute Makereg from its definition environment or from the environment of the definition of Deposit entry (both of which differ from the Top environment), then we may be assured that nothing in that execution can affect any files in the normal Top environment.

In summary, when we modified this application, the error checker led us to produce a consistent program. Then we outlined some possible tests for our modifications to see if the programs perform as we expected. The watch and run that we could invoke on the process definition do not use or affect the files belonging to Top or to Accounts payable—β allows us to test our modifications independently of the real files.

- *User session summary*

In this example user session, we have explored an unfamiliar application program. We saw that we could ask what each of the files was supposed to do and that we could look at values in the files.

We chose to run Accounts payable to see what it would do; then, to examine its operation more closely, we watched it run. The watch showed us each subprocess as it was about to be invoked. At each step in the execution, we could inspect the files, modify them, or ask for descriptions of the individual files and processes.

Then we outlined a modification. The error checker pointed out that the change we made in the Deposit record definition caused an inconsistency with a process definition. We remedied the problem; we were then ready to test the modification by watching the changed program run.

Remarks

A β prototype was written in LISP/370 [9]; it runs on an IBM VM/370 system. The prototype contains the facilities shown in Fig. 5—the controller, editor, executor, displayer, and error checker.

The complete Accounts payable program, a fragment of which is shown in Figs. 12 to 22, was produced in its entirety in the prototype. In addition, parts of some other applications were produced and demonstrated.

β language objects were chosen so that they correspond to physical objects familiar to the user—slips of paper and processes on those slips. An effort was made in the language design to let the simplicity of the file and process relationships be apparent to the user. The tag mechanism allows specification of all necessary exceptional detail in a way intended not to interfere with that simplicity. Single word tags are encoded on the process diagrams. Tags with values provide most of the real detail in their value expressions; they are kept hidden until the user explicitly asks to see them. Then they are presented in an orderly fashion; the user may see and modify any or all tags on any one item at a time.

In the original β design, all processes were allowed to operate in parallel, as is the case with normal "data-flow" systems [10]. In using β to write applications, however, it was found that additional files were introduced merely to provide sequencing. Because (a) many processes are simpler if parallel operation is not allowed, and (b) users did not appear to be bothered by the sequencing notion, it was decided to presume sequential execution. Hence SEQUENCE tag values are used to decide which of two apparently simultaneous subprocesses is selected for execution. Parallel operation—indicated by the presence of the PARALLEL tag—is reserved for only those processes where it is necessary.

We have not described how new β programs are constructed. This is important to the professional programmer who is to produce the application programs originally. How subparts are located and how arrows are routed in definitions are obvious concerns in specifying new programs. But it turns out that the editor handles both these concerns already. When a definition is modified by having a new subprocess or subpart added, the editor places it in an available space; the user may move it to where he wants. He does this by using commands "up," "down," "left," or "right" along with the number of spaces to move. The user constructs a new arrow by giving the "arrow" command; then he selects in turn the file or process on the "from" end and the "to" end. The editor contains a relatively simple arrow routing process that decides on the vertical and horizontal order in which to display arrows. Leaders (busses) are constructed as needed to extend file and process delimiters in the vertical direction. As files and processes are moved, then, the arrows are reordered and rerouted by the editor.

Tag values form a sublanguage. These values, as shown in the CALCULATION tag examples given earlier, consist of expressions which apply β built-in processes to β files. The left-to-right evaluation rule allows β networks to be expressed linearly.

In place of this new sublanguage, BASIC [11] or some similar existing language could have been substituted. Using BASIC expressions and BASIC programs as tag values would have allowed BASIC program fragments and BASIC programs, respectively, to be invoked as primitive β processes. The only requirement would have been to define a clear mapping between β files and BASIC data.

Though it was attractive, the BASIC sublanguage was not used in the prototype. The β-compatible tag value language was chosen for the prototype so that entire applications could be expressed in β. It was felt that the significant challenge was to the capabilities of β as a stand-alone means of application expression, and that a BASIC sublanguage facility could be added later.

Summary

The β system consists of a file- and process-oriented language plus a processor used to examine, modify, and execute programs expressed in that language. The system adapts techniques developed for communication among professional programmers to the particular problem of communicating business applications to the end user.

The application end user is to be furnished with application programs written in the β language. With the β processor, the user examines the applications and modifies, tests, and runs them. In all of this, his communication is always in β terms. Even when errors occur in executing, the user is still able to recover and to proceed without leaving the β environment.

Appendix A: β built-in operators

β built-in operators are listed below. These built-in operators were necessary for the example programs used dur-

ing β development. The list may be extended to include additional ones of the sort provided in conventional programming languages, *e.g.*, an operator to deliver the maximum value in a sequence.

- Arithmetic processes: +, −, *, /, REMAINDER, and EXPONENT.
- Comparison processes: =, <, >, <=, >=, <>. These produce a STRING "yes" or "no" output file value.
- Logical processes: AND, OR, and NOT that operate on STRING "yes" and "non-yes" values.
- Interactive processes: DISPLAY, REVIEW, and EDIT. Each shows its input file to the user. REVIEW lets the user select and copy the input file values to the output file; EDIT lets him copy and change the values before placing them in the output file.
- Interactive process: ENTER. It is a source of file values; it shows the user a blank datum (where NUMBER subparts have zero values and string subparts have values which are strings of underlined characters). The user can EDIT this blank datum and place it in the output file.
- SUBFILE and SORT require sequences as input and output files. SUBFILE produces those members on which the expression value of the WITH tag evaluates to "yes." SORT produces a file in which the members are arranged in ascending order of the expression value of the BY tag.
- MAKE and BREAK processes are used with a RECORD file to compose it from and decompose it into its subparts.
- GIVES is the fundamental value mapping process; it copies the input file value into the output file.
- CALC is used to specify a calculation. See the sections "Calculating expressions" and "Conditionals."
- COUNT gives the cardinality of the sequence in the input file.
- SUM delivers the sum of the values in the input file sequence.
- FIRST and LAST produce the first and last values, respectively, of the input file sequence.
- TODAY produces the value of the data which are kept in a system file.

Acknowledgments

The author acknowledges the assistance afforded by P. R. Kosinski of the IBM Research Division for his constructive criticism of the design of β and, most important, for his help in producing the prototype system. Others in IBM Research who contributed through discussions and support include A. K. Chandra, V. J. Kruskal, and I. Wladawsky. Support from IBM General Systems Division, in particular from H. D. Wyngarden, W. E. Frey, and B. W. Landeck, made the project possible. The thorough and careful work of the referees is much appreciated.

References

1. K. Jensen and N. Wirth, *PASCAL User Manual and Report*, 2nd Ed., Springer-Verlag, New York, 1974.
2. B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. L. Popek, "Report on the Programming Language Euclid," *ACM Sigplan Notices* **12,** 2 (1977).
3. M. Shaw, W. A. Wulf, and R. L. London, "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators," *Commun. ACM* **20,** 553-564 (1977).
4. B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," *Commun. ACM* **20,** 564-576 (1977).
5. M. Hammer, W. G. Howe, V. J. Kruskal, and I. Wladawsky, "A Very High Level Programming Language For Data Processing Applications," *Commun. ACM* **20,** 832-840 (1977).
6. T. Winograd, "Beyond Programming Languages," *Commun. ACM* **22,** 391-401 (1979).
7. J. B. Goodenough, "Exception Handling: Issues and a Proposed Notation," *Commun. ACM* **18,** 683-696 (1975).
8. J. L. Peterson, "Petri Nets," *ACM Comput. Surv.* **9,** 223-252 (1977).
9. *LISP/370 Program Description/Operations Manual*, Order No. SH20-2076-0, available through the local IBM branch office.
10. R. E. Bryant and J. B. Dennis, "Concurrent Programming," *Research Directions in Software Technology*, P. Wegner, Ed., MIT Press, Cambridge, MA, 1979, pp. 584-610.
11. J. G. Kemeny and T. E. Kurtz, *BASIC*, Sixth Edition, Dartmouth College Computing Center, Hanover, NH, 1974.

Received January 24, 1980; revised June 18, 1980

The author is located at the IBM General Systems Division Laboratory, P.O. Box 2150, Atlanta, Georgia 30301.