AT&T Technical Journal
Vol. 64, No. 9, November 1985
Printed in U.S.A.

IFS—A Tool to Build Integrated, Interactive Application Software

By K.-P. VO*

(Manuscript received April 9, 1984)

The Interpretive Frame System (IFS) is a tool for creating application software with sophisticated interactive interfaces. IFS is based on the notion of a frame network. A frame network consists of many interconnected modules called frames, each of which represents a logical activity in the system. Frames are written in a high-level language. Besides the usual computational constructs such as conditionals, loops, or arithmetics and Boolean expressions, the IFS language also includes facilities for building program/program interactions, such as subprocess invocations or coprocess communications, and constructs for building user/program interactions such as menus or forms. IFS is a suitable tool to integrate existing programs by providing a uniform and easy-to-use user interface. It can also be used to build a new system in a top-down manner by first defining the network of frames and their interactions and user interface, then programming problem-specific parts. Therefore, it provides a general framework supporting any combination of top-down and bottom-up software development methodologies. This paper gives an overview of the frame network concept, the user interface of frame network systems, the frame programming language, and the IFS system implementation.

I. INTRODUCTION

Much effort in current application software development is directed toward building systems to be used by users with little expertise in computing. These application systems typically combine the functions of many generic computing systems, such as data management and

* AT&T Bell Laboratories.

statistical data analysis. However, the users of such a system often do not care about how the system was implemented, but only how useful it is in their work and how easy it is to use and learn. Therefore, the user interfaces of these systems are usually interactive, and provide users with guidance to the various system resources and their proper use.

There are difficulties in building application software. First, there is a lack of a convenient way to put together collections of programs and control the information flow among them. The problem is further aggravated, since generic computing functions are often available in separate programs but with very dissimilar interfaces. Second, there is a lack of a good tool to create and control sophisticated, interactive user interfaces. Given the intended community of users, it is easily arguable that the major (and hard) part in building a system lies in creating the right interface.

The Interpretive Frame System (IFS) provides a solution to the above problems. An application software system is conceptually viewed as a network of modules called *frames*. Each frame represents a logical activity in the system. The logical activity includes all necessary computing functions, their interactions, and user interactions. Depending on the usage, the network transitions between frames serve two different purposes: to ensure correct execution of cohesive task sequences and to provide guidance in proper use of loosely coupled tasks.

IFS consists of a language to write frames and a system to interpret frame actions. The IFS programming language provides the following:

1. Constructs for user/program interactions, such as menus or forms,
2. Constructs for program/program interactions such as subprocess invocations and coprocess communications,
3. Constructs for transition and passing information among frames, and
4. The usual computational constructs such as conditionals, loops, arithmetics, string operations and expressions to control the interactions of (1), (2), and (3).

The IFS interpreter carries out the actions encoded in frames. It further provides facilities to be used directly by end users to customize the user interface to local requirements and to arbitrarily access frames where appropriate.

IFS is a tool usable to support a large spectrum of software development methodologies.[1-4] At one end, a software product can be developed from bottom up by putting together existing programs and providing a high-level, uniform interface based on frames. At the other end, a product can be developed from top down by first defining the frames, their interactions, and user interactions before implementing the low-level computing functions. In practice, we have observed compromises in which a prototype is built based on some initial and perhaps incomplete specifications, but with a realistic user interface, and partially supported by existing programs. The prototype is then continuously refined to fill in gaps in requirements and to tune efficiency until the final product results.

The remainder of the paper is organized as follows. Section II describes an architecture for application systems and its relation to the frame network model. Section III first gives an example of a frame network system and its user interface, then gives an overview of frame programming and other IFS system features. Section IV describes the current IFS implementation. Finally, in the conclusion, we discuss some aspects of current IFS usage, problems, and possible future enhancements.

II. APPLICATION ARCHITECTURE AND FRAME NETWORKS

2.1 The architecture of an application

The architecture of an application software system can be roughly divided into four layers: a data component, a set of low-level functions, a structure organizing the functions into logical tasks, and a user interface.

At the lowest layer is the data component describing the data format and its physical storage. Depending on the application, the data architecture can be simple flat files or sophisticated databases or combinations thereof. The next layer is a supporting set of generic functions to act on the data. These functions range from single programs for file listing or sorting to systems for data analysis and graphics or report generation. The elements comprising the first two layers are usually present as services provided by the computing environment. Even if some of these functions are not available, from a software reuse standpoint it is probably worth it to implement them as independent programs because of their applicability. The third layer comprises the computing solutions to the problems of the application. It organizes the appropriate set of generic functions into logical tasks with the right granularity, and controls the information flow among them. Finally, the top layer is the user interface that controls how users will use the system.

In a typical application, the top two layers are where developers should spend the most effort. They form the facade from which users perceive the system and, as such, determine its success or failure. Because of a lack of proper tools, however, the two top layers are often done in an ad hoc manner, resulting in systems that are hard to use

USER INTERFACE
(MENUS, FORMS, ...)

SYSTEM STRUCTURE
(MODULES, CONTROL STRUCTURE, DATA FLOW)

GENERIC FUNCTIONS
(STATISTICS, GRAPHICS, DATA MANAGEMENTS, ...)

DATA ARCHITECTURE
(FILES, DATABASES, ...)

Fig. 1—The architecture of an application system.

and to maintain. IFS solves the problem by providing developers with a robust framework to modularly structure the activities in their systems. Therefore, local changes in the application structure do not necessitate or propagate changes in the entire system. IFS also provides a large and consistent repertoire of tools to customize the proper interfaces among programs, as well as between programs and users.

2.2 Frame networks

A frame network provides an abstraction for the top two layers (see Fig. 1) of an application system. A frame represents the totality of a logical task and all aspects of its internal as well as external interactions. At the program level, it contains all necessary computing functions to carry out the task. At the system level, it controls the information flow among subprograms and to and from other frames. At the user level, it provides users with interactions and guidance to ensure proper inputs and correct execution of the task. A frame can call other frames to perform subtasks or simply transfer control to another frame. The call and control transfer structure forms the frame network connectivity. Thus far, the frame network structure is described similarly to a subroutine network structure in regular procedural programs. This is not always the case. Indeed, for many applications, the sequences of task execution are highly cohesive and in such cases frames do behave as subroutines. However, in many other applications, a frame only represents a high-level abstraction of a task in a collection of loosely coupled tasks. In such a case, the order in which frames are executed is immaterial. The frame network transitions are used mainly to guide users to various resources available in the system. The IFS interpreter in fact allows users to arbitrarily access frames in such cases.

III. SYSTEM DESCRIPTION

3.1 A reminder service: example of a frame network system

In this section, we present an example of how a frame network system is put together on top of a set of programs constituting a reminder service. Then, we show a scenario of using the final system.

The reminder service maintains a database of reminder items. It provides four functions: adding new items, deleting old items, peeking to see items in some time range, and sending out reminders at appropriate times. For simplicity, we shall assume that these functions are implemented as independent programs: `add`, `delete`, `peek`, and `remind`. Their actual implementation details are irrelevant, since we are interested only in their use. For example, to add into the database a new reminder item to be activated on a particular date and at a particular time, `add` is called as

```
add "ContentOfReminder" "date" "time"
```

At the respective date and time, the program `remind` will generate a reminder to be sent via the appropriate media. Note that `remind` is an automatic service of the system that users never have to invoke directly.

As a system of programs, the reminder service can be used directly but not in a transparent manner. For example, the syntax of calling `add` requires the arguments to the program to be in a rigid sequence of positions. Therefore, it is desirable that a more integrated and flexible interface be put on top of the programs. We do this by putting together a frame network on top of the programs `add`, `delete`, and `peek`. The frame network consists of four frames: Reminder, Adder, Deleter, and Peeker connected as a rooted tree. Reminder is the root of the tree and consists of a menu offering the services of the other three frames. The sole action associated with each menu choice in this case is to activate the appropriate frame. Adder, Deleter, and Peeker are frames directly interfacing the programs `add`, `delete`, and `peek`. Their jobs are to collect the necessary information to invoke the underlying programs. For example, Adder is implemented as a form to collect the content of a reminder and its date and time, then to invoke `add` with the collected information. Figure 2 summarizes the architecture of the final reminder system. The reader should compare the layers of Fig. 2 to those of Fig. 1.

The rest of this section shows a scenario of interacting with the reminder service frame system. A few words should be said about the screen organization employed by IFS. The top line of the screen shows the title of the current frame, a piece of information summarizing the function of the frame. Each frame is uniquely known in its network by its Identification (ID) string. The second line of the screen shows the stack of IDs of frames traversed to get to the current frame. The IDs are separated by either the colon : or the vertical bar |. As frames are called, the ID stack grows from right to left so that the current frame ID is the rightmost string (see Figs. 3, 4, and 5). Users of a frame system can observe the frame ID stack and learn the functions

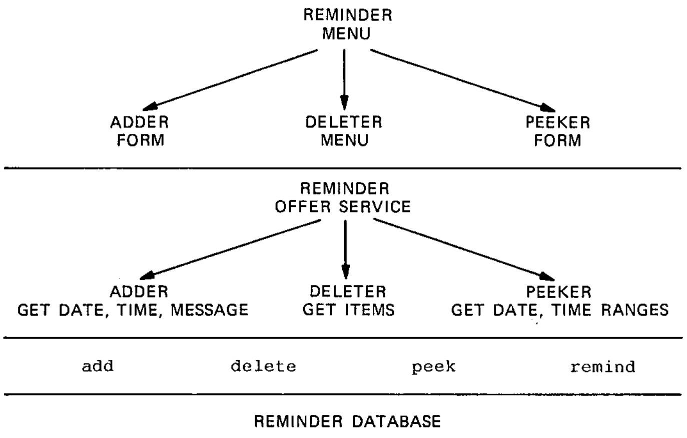
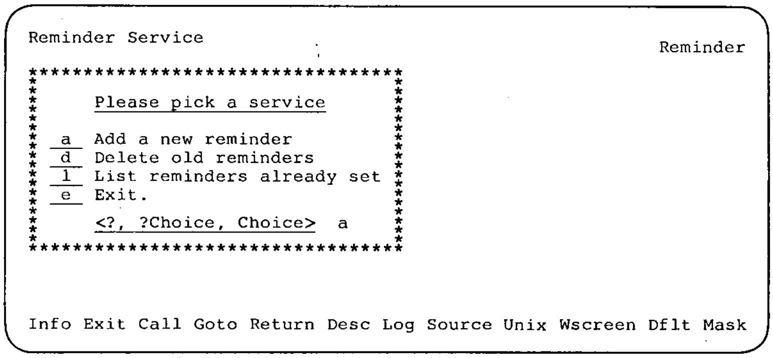


Fig. 2—The reminder service frame network system.



```
Reminder Service
                                                          Reminder
**********************************
*                                *
*     Please pick a service      *
*                                *
*  a  Add a new reminder         *
*  d  Delete old reminders       *
*  l  List reminders already set *
*  e  Exit.                      *
*     <?, ?Choice, Choice>  a    *
*                                *
**********************************


Info Exit Call Goto Return Desc Log Source Unix Wscreen Dflt Mask
```

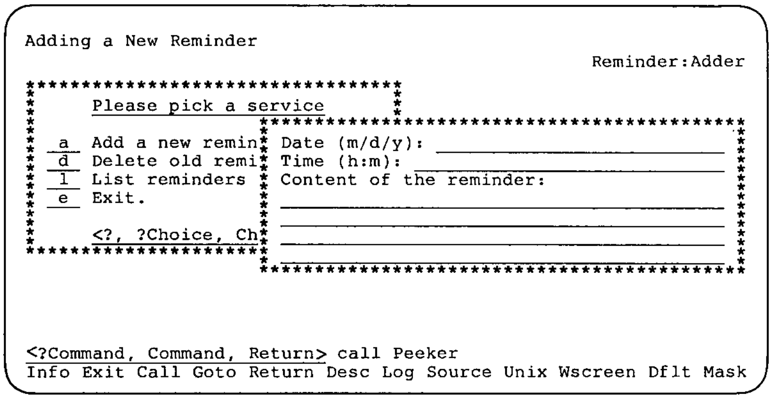
Fig. 3—Interacting with the reminder service—the Reminder frame.

of different frames and how they are structured in the network. The bottom line of the screen shows global commands that are provided by the IFS interpreter independent of the frame programming. These commands can be accessed by users by typing the escape key (the key labeled with ESC) to make direct use of certain frames in the application system or to customize their interaction environment such as changing the display or setting defaults to repetitive questions. Some of the more interesting commands will be discussed in Section 3.3.

Figure 3 shows the root frame, Reminder, of the Reminder Service. The second line of the screen shows `Reminder`, the ID of the Reminder frame. The middle of the screen shows a menu of available services. Each menu item has two parts: a selector, which is an underlined string, and a brief description summarizing the functions of the item. The parts of the menu prompt <`?`, `?Choice`, `Choice`> indicate that an item is chosen by typing its selector, and further information on a particular item, if available, can be obtained by typing `?` followed by its selector. The `?` when typed by itself always signifies a request for further information on the current interaction item, in this case, the menu prompt itself. Here, the user had typed `a` to choose the option of adding a new item into the reminder database.

Figure 4 shows the input form of the frame Adder appearing in a window overlaying the window of the menu. A window is a display device used to retain as much of the previous context of interaction as possible. The second line of the screen shows the string `Adder` separated from `Reminder` by a colon, `:`, indicating that Adder was called from Reminder as programmed in the network. Before actually adding a new item, the user decided to check the database for any possible conflict. The escape key was typed to gain access to the global functions. The global command prompt `<?Command, Command, RETURN>` was displayed. Now the user could have used the command `return` to return to the Reminder frame, then pick option `l` to bring up the Peeker frame. However, since the user had used the system before and knew (by observing the ID stack) that Peeker was the right frame to use, the command `call Peeker` was used to directly call up the Peeker frame. The semantics of the global command `call` is to suspend the execution of the current frame and start the execution of the called frame. Thus, Adder was suspended while Peeker executed. This gives an example in which an experienced user of a frame network system can directly access any part of the system, bypassing the programming of the network.

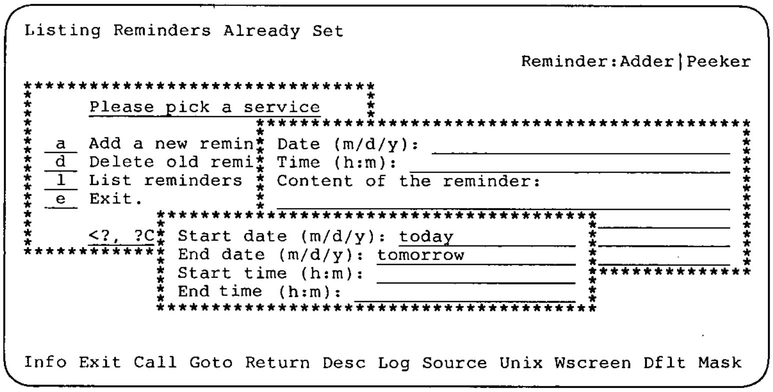
In Fig. 5, upon receiving the user's request for executing Peeker, the execution of the frame Adder was suspended and the frame Peeker was brought up as an input form in another window. The second line of the screen now showed the ID of Peeker separated from the other



```
Adding a New Reminder
                                                       Reminder:Adder
********************************
*                              *
*     Please pick a service    *
*                    *******************************************
*                    *                                         *
*  a  Add a new remin* Date (m/d/y): __________________________ *
*  d  Delete old remi* Time (h:m): ____________________________ *
*  l  List reminders * Content of the reminder:                 *
*  e  Exit.          *                                         *
*                    * _______________________________________ *
*                    * _______________________________________ *
*     <?, ?Choice, Ch* _______________________________________ *
*********************  _______________________________________ *
                     *******************************************


<?Command, Command, Return> call Peeker
Info Exit Call Goto Return Desc Log Source Unix Wscreen Dflt Mask
```

Fig. 4—Interacting with the reminder service—the Adder frame.



```
Listing Reminders Already Set
                                                        Reminder:Adder|Peeker
********************************
*                              *
*     Please pick a service    *
*                        ******************************************************
* _a_ Add a new remin* Date (m/d/y): ________________________________         *
* _d_ Delete old remi* Time (h:m): __________________________________         *
* _l_ List reminders * Content of the reminder:                               *
* _e_ Exit.          * __________________________________________________     *
*           ***************************************** __________             *
*     <?, ?C* Start date (m/d/y): today        *    __________                *
************* End date (m/d/y): tomorrow       *    __________                *
            * Start time (h:m): _______________ ***************
            * End time (h:m): _________________ *
            *************************************

Info Exit Call Goto Return Desc Log Source Unix Wscreen Dflt Mask
```

Fig. 5—Interacting with the reminder service—the Peeker frame.

IDs by a vertical bar indicating that Peeker was called directly by a user's request. The user filled in the first two fields of the form, indicating that all reminder items between the present day and the next day were to be shown.

In the remainder of the session, Peeker invoked the program **peek** to extract the relevant set of reminder items from the database and showed them to the user. Then, the Peeker window was popped from the screen and the execution of Adder continued.

3.2 Programming frames

Frames and their interactions are written in a frame programming language. The language provides the following:

1. Constructs for interactions between the program and user such as menus or question-answer form of dialogues,
2. Constructs for interactions among the programs such as subprocess invocations or coprocess communications,
3. Facilities for transition and passing information among frames, and
4. The usual computational statements such as conditionals, loops, and expressions for control over those interactions.

Syntactically, each frame consists of nested labeled blocks where each block represents an "interaction unit." For example, the menu in Fig. 3 of the Reminder Service can be specified by a menu block, called an {m block, as in Fig. 6.

In this example, the menu block, enclosed between the pair of strings {m and }m has a title block (enclosed between {t and }t and four option blocks (each one enclosed between {o and }o). Each option block in turn has a title block. The semantics of this menu block is that a menu of four items as shown in Fig. 3 is to be displayed and

```
{m : (0 != 0)
      {t
      Please pick a service
      }t
      {o a
            {t
            Add a new reminder
            }t
            ~call Adder
      }o
      {o d
            {t
             Delete old reminders
            }t
            ~call Deleter
      }o
      {o l
            {t
            List reminders already set
            }t
            ~call Peeker
      }o
      {o e
            {t
            Exit.
            }t
            ~exit
      }o
}m
```

Fig. 6—A Menu block.

when the user picks an option the actions, if any, specified within that option, are to be performed. The sole action for the first option in the above example is `~call Adder`, which will call the Adder frame. The condition `:(0 != 0)` is called an exit condition. When a block (or any other syntactic unit) is executed, it is iterated until the exit condition becomes true. In this case, since `0 != 0` is always false, the menu block has been specified to loop forever. The system will exit altogether, however, when the user picks option `e`.

Each frame description in general is a frame block enclosed between a pair of strings `{f` and `}f`, and has within it several other types of blocks such as menu blocks, action blocks, context blocks, etc. Each block is optionally associated with an entry condition and an exit condition, and has within it an optional title block, a description block and a sequence of other blocks or actions. A partial Backus Normal Form (BNF) specification of the frame language syntax is in Appendix A. To illustrate the simplicity of the language, the complete descriptions of the Reminder frame and the Adder frame of the Reminder Service example are provided in Appendix B. We shall discuss here the semantic issues involved in interpreting the different types of blocks and transitioning among frames.

3.2.1 Frame identification

A frame block is the outermost block of a frame and has the following form:

```
{f frame_ID frame_args
. . .
}f
```

It defines the frame identification and any arguments that the frame may require when it is called. A frame is uniquely identified in its network by the `frame_ID`. Frames can be called directly by the users by their IDs. Thus, frame IDs serve as high-level abstractions of functions in a frame network.

3.2.2 Conditions, loops, and variables

Each block in the language can be optionally associated with two conditions as follows:

```
{. (entry_cond) : (exit_cond)
      . . .
}.
```

Either or both of the conditions can be absent with the default value `true`. If the entry condition is true, the block is executed. After that, it is iterated until the exit condition becomes true. The conditions are logical expressions involving built-in numerical string constants, variables, regular expression matching, and numerical comparison operators. Variables in IFS have the form `variable_name` and are of string type. However, when a variable is involved in numerical computations or comparisons, it is cast into a real number. At that time, if the string value does not represent a real number, it is assumed to be zero. By default, the scope of a variable is the frame in which it appears. It is also possible to dynamically change the scope of a frame variable to be shared with other frames.

3.2.3 Dynamic menus

The format of an option block in a menu block is

```
{o (entry_cond) : (exit_cond) selector
      {t
      Brief description
      }t
      {d
      Online help text for the option
      }d
      Actions
}o
```

The entry condition of an option, if true, signifies that the option is available to be offered. Otherwise, it will be suppressed. Systems such as ZOG[5] provide only static menus, i.e., the list of items in a menu is fixed by the specification. In IFS, a menu is a dynamic entity that can be customized to the context of user interaction at that instance. An option is selected by users by typing in its selector. Upon the users' selection, the actions associated with the option are executed and iterated until the exit condition becomes true. The actions are composed from other general constructs of the frame language including new menus, forms, transitions to new frames, or process communications. The description block `{d` provides context-sensitive help for the option. The help text can be parametrized by embedding frame variables. Users can obtain help on an option by typing the question mark and the option selector.

3.2.4 *Gathering user values*

Question blocks provide a mechanism for gathering user values that can be used in subsequent actions. The format of a question block is

```
{q (entry_cond) : (exit_cond) ~input_var
The Question
        {d
        Online help text for the question
        }d
{q
```

The string entered by the user in response to this question is assigned to the variable `~input_var`. The entry condition in a question block, if true at that instance, indicates that the input variable must be set interactively and the question is asked. If the entry condition is false, the question is irrelevant in the current context and is skipped. Thus, if the questions were displayed in a form, their entry conditions define dynamic field protection. The exit condition of a question represents input validation. After the user input is received, the exit condition is evaluated and if it is false, the input is rejected and a new input is requested. To aid input validation, the frame language provides a rich set of comparison operators that includes regular expression matching as well as numerical comparisons. The Adder frame example in Appendix B shows how the values for three variables, `~date`, `~time`, and `~content`, can be obtained from the user before the reminder database can be added with the new data. The only validation check in that case is that the input for `~date` must be non-null.

3.2.5 *Process invocation and communication*

Problem-specific processing in a frame can be done by invoking in an action block a subprocess which is usually a *UNIX*™ operating system shell program.[6] The simplest form of an action block is

```
{a (entry_cond) : (exit_cond) > ~ret_val1 ~ret_val12 . . .
     shell_program_name argument_1 argument_2 . . .
     . . .
}a
```

The interpretation of the entry and exit conditions is as in other blocks. A new shell is invoked every time the {a block is executed. The lines inside the {a block define a shell script to be executed. There are also mechanisms in the IFS language to allow a subprocess to return values back to IFS. The returned values, if any, are assigned to the return variables ~ret_val's.

Provided some concurrency conditions are met by the invoked processes, the shell can be run as a coprocess as follows:

```
{p (entry_cond) : (exit_cond) "/bin/sh" "-i" >  ~ret_val1
   . . .
~!EndOfInput,EndOfOutput
   shell_program_name argument_1 argument_2 . . .
   . . .
}p
```

In this method, the shell is invoked once to run interactively and stays in the background with its standard input and output channels connected to IFS. The communication protocol between IFS and a coprocess is defined by the control line consisting of EndOfInput and EndOfOutput strings. The EndOfInput string defines a pattern that indicates the end of a message from the coprocess to IFS. A minus sign for that string indicates that no input is expected from the shell coprocess. Similarly, EndOfOutput defines a delimiting string to be sent by IFS after the message has been sent. A minus sign again signifies that IFS does not have to send any message delimiter to the shell coprocess after the message is sent. For example, the action block in the Adder frame of Appendix B can be replaced by the following {p block shown below:

```
{p (~date != ~null && ~content != ~null) "/bin/sh" "-i"
~!-,-
add "~content" "~date" "~time"
}p
```

Here, the entry condition shows that the {p block is only executed if either ~date or ~content are not empty. The first time the {p block is executed, an interactive shell program is invoked and put in the background with its standard input and output channels connected to IFS. Each time the {p block is executed, the body of the message contained in the block is sent to the shell coprocess. Subsequently, the shell executes the program add to update the database and then waits for other messages from the frame system. Meanwhile, the frame system resumes its normal activities.

3.2.6 Compound actions

A sequence of related actions can be grouped together in a context block, the {c block. The actions of a context block can be other contexts, menus, process communications, dialogues, etc. In particular, a group of questions to be displayed and executed as a form must be together in a context block. Besides the interactive nature of the actions inside a context block, the context block is similar to a compound statement in a procedural language. In the Adder frame in Appendix B, three question blocks and an action block are grouped together in a context block.

3.2.7 Transitions among frames

Transitions among frames are accomplished via the operators `~call`, `~goto`, and `~return`. The operators `~call` and `~return` behave like calling and returning from subroutines in a regular programming language with the additional feature that frames can return multiple values. The action of the first menu option in the Reminder frame is to call the frame Adder. A sequence of `~call` creates a stack of frames. The operator `~goto` restarts such a stack with a new frame.

3.2.8 Other remarks on programming frames

The reader should note that there is no display information involved in the definitions of any of the frames. In IFS, display programming is separated from logical programming. This aspect of IFS contrasts with other screen definition languages in the literature (for example, see Ref. 7) where the system builder has the burden of laying out the complete design of screen displays. As far as a frame is concerned, the parameters that it requires are obtained by asking a series of questions, user choices are obtained by selections from menus, and processing of user data is done by running subprocesses or coprocesses. Navigation in the frame network is performed by IFS actions. Display organization is either done in a default mode by the interpreter or customized by using a display editor or a display language. The combination of all these facilities in a system makes IFS unique in its ability to facilitate the building of integrated and interactive software systems that can be used in a variety of environments.

3.3 IFS system features

An example of interacting with a reminder service frame system was shown earlier. The user of a frame system faces a terminal screen showing various types of information. The action expected from the user is clearly defined from the interaction context. If a menu is showing, a choice is expected. If a form is showing, some response is expected to fill in the field of the form where the terminal cursor is situated. The user, however, can also initiate other actions by typing either the question mark or the escape key (the key labeled with ESC). The question mark indicates that the user needs more information on the current interaction item and such information, if extant, is displayed by the system. For example, if the current interaction item is a field in a form, the information explaining this field is displayed. The escape key, on the other hand, indicates that the user wishes to make use of the global commands shown on the bottom line of the screen. The set of global commands that can be augmented or partially suppressed by system developers provides functions usable independent of the frame programming. The global commands provided by IFS itself range from customizing the interaction environment of the system to randomly accessing different parts of the frame network. A full description of these commands is beyond the scope of this paper. We present some of the more interesting commands below.

3.3.1 Display editing: `Mask`

The display environment of a frame system can be customized using the global command `Mask`. `Mask` lets users interactively define new layouts for forms and menus as well as windows containing these constructs. For example, to lay out a form, it is necessary to know where on the screen to display field labels, what video attributes to display them in, and whether the fields are left, center, or right justified. To ease the definition of such information, the display editor `Mask` lets users move the cursor freely on the screen to indicate where a label should be drawn. It also presents users with lists of available colors to display different parts of the form.

The ability for redefining the display environment at will is important because different users have different requirements dependent on their local environments; therefore, it is unreasonable to force conformity a priori. Further, for the system builders, having `Mask` makes it easy to experiment with different styles of display.

3.3.2 Default answer setting: `Dflt`

The global command `Dflt` lets users set up default answers to questions or fields in a form. During the execution of such a question or field, the user can choose one of the default answers with a single keystroke or overwrite them by typing a new answer.

`Dflt` is menulike in the following sense. A question or form field is a parameter collector whose domain of values is large in general but may be restricted for individual users. For example, a question for names has an infinite domain in general but for an individual user, its domain probably has size one. The system builder who solves a general problem must program such a parameter collector for the general case. The command `Dflt` lets individual users tailor such a collector into a personalized menu for transparency and efficiency.

3.3.3 Frame random accessing: `call`

The global command `call` can be used to randomly call any frame in the system, provided that the ID of the called frame is known. The execution of the frame previous to the call is temporarily suspended and is resumed after the completion of the called frame.

The command `call` is one of a family of network movement commands which includes `goto`, `return`, `break`, and `exit`. These commands let advanced users of a system directly access parts of the system independent of the network programming.

IV. CURRENT IMPLEMENTATION

IFS is written in the C language[8] and based on the *UNIX* operating system. It has been used on many flavors of the *UNIX* system, including versions of AT&T System V and University of California at Berkeley 4.1 BSD and 4.2 BSD.

In the current implementation there are four main programs: a frame language compiler (`ifc`), a display language compiler (`ifv`) and a decompiler (`vfi`), and an interpreter (`ifm`). The steps in using these programs to create and run a frame are roughly as follows:

1. Write/edit and compile (`ifc`) a frame language script.
2. Write/edit and compile (`ifv`) a display language script if desired.
3. Interpret (`ifm`) the frame.
4. Modify the display using the global command `Mask` if desired.
5. Decompile (`vfi`) display information into a readable ASCII form.
6. Stop or go back to step 1.

These IFS programs are presented below.

4.1 Frame language compiler: `ifc`

The frame language compiler, `ifc`, compiles frame scripts into intermediate data structures which are interpreted by the interpreter. In the compiling process, the frame scripts are checked for valid syntax and any static text processing is done. Because IFS is interpretive in nature, it is possible to fold the functions of the compiler into the interpreter itself and reduce the complexity in using the system. However, we chose this division to speed up run-time execution. The trade-off is attractive because most frame systems are built once but would be used many times and certain costs in frame processing such as text processing are significant.

4.2 *Display language compiler and decompiler:* `ifv`, `vfi`

Display information of a frame is kept in a separate data structure whose structure closely reflects that of the frame structure. The display structure can be created in two different ways, by using the global command `Mask` (Section 3.3.1) or by writing a display language script and applying the display compiler `ifv`. In an application, it may be desirable to standardize the display of certain collections of information such as on-line help texts. The display language eases the display standardization of many frames since the display script of one frame can be easily modified and replicated for other frames using standard text editors in the operating system. On the other hand, creating the first display description of a series of frames is easier using the `Mask` command than by laying it out on paper and writing a display script. Therefore, `Mask` is usually used to make the first prototype of a display structure. Afterward, the display decompiler, `vfi`, is used to convert the structure into the display script form which can be further processed for other frames using text editors.

4.3 *Frame interpreter:* `ifm`

The interpreter, `ifm`, executes actions encoded in frames (Section 3.2), controls the display, and processes users' inputs (Section 3.3). It also provides global commands that users can use independently from the frame network programming to customize their local interaction environment (Section 3.3). Internally, the interpreter is divided into two parts: a control part that interprets frame actions, and an interactive part that defines appropriate displays and interactions for different interactive constructs.

The control part of the interpreter consists of routines to get frame and display structures from the file system and routines to interpret the programming language constructs such as menu, form, or network movements. Each construct of the language is interpreted by one routine, and the nesting of constructs is implemented by recursion. Since frame and display structures are kept as files in the file system, there is a cost to read them into memory when a frame is executed. To increase system efficiency, the action code of direct or indirect recursive frames is shared. Further, a marking technique was implemented to retain popular frames in memory for some period of time after their executions are completed.

The interactive part of the interpreter is similar to the control part in structure. It consists of routines to take care of the display and interaction with different interactive constructs of the language. The routines assumes a minimal capability from the terminals, the ability to move the cursor to any point on the screen.

V. FINAL REMARKS

We have presented an overview of IFS, a software tool to build interactive and integrated software. The main contributions of IFS are the concept of a frame network as a model for structuring application systems and a high-level programming language embodying this model. The programming language includes a large set of interactive facilities such as menus or forms for building user interfaces and constructs such as interprocess communications designed to aid the building of interfaces around existing programs. At this writing, IFS has been in use for a few years. Many frame systems have been built, some with hundreds of frames. These applications span a wide range from analytical systems to office automation systems and systems integrating Computer-Aided Design/Computer-Aided Manufacturing (CAD/CAM) tools. The experience gained in building these applications shows that IFS has been effective in reducing the work between conception and realization of an application system. To conclude the paper, we make a few observations about the current use of IFS, some perceived problems and possible future improvements.

In IFS, interaction programming and analytical programming are separated. The main work of frames is to coordinate the flow of information among the modules and the user. Problem-specific computations are often carried out by other programs. This architecture makes it easy to reuse software in building new systems. Further, it encourages experimentation with high-level modules and user interactions before actual work on the analytical part begins. In fact, we have observed a successful software construction method in which prototypes are built and continuously refined until the final products result. The benefit of constructing software this way is that at any time, the user interface of a prototype is fully functional so that selected users can make trial use and help developers debugging it.

In building interactive systems, a balance needs to be maintained between ease of learning and ease of access. For example, a straightforward menu hierarchical system may be easy to learn at first, but can also quickly become restrictive when users get familiar with the system functions. Within the IFS framework, there is enough flexibility for system builders to make poor judgment and build systems that are awkward to learn or to use. Nonetheless, the applications built using IFS have been generally satisfactory. This is helped in part because the frame network concept and the programming language are easy to learn. Therefore, many of the applications were built by people who are not expert programmers, but who intimately know the use of the applications and sometimes are users themselves. Further, IFS provides integrally many different mechanisms to build and dynamically control interactions. With the right judgment, the user interface of an application can be made to strike the balance between ease of learning and ease of access. By and large, this has been observed in practice.

As a programming environment, IFS lacks some desirable features. For example, in the programming language, there are currently no *explicit* ways for system builders to raise or handle exceptions such as asynchronous events like terminal hangup or other operating system signals. The lack of such facilities have made the building of interfaces to certain database operations rather cumbersome. There is also a less frequent need for a debugging tool, especially to examine the state of a frame system upon unexpected computational results. This need has not been acute because frame systems are interactive by nature and their anomalies tend to manifest quickly during trials.

To accommodate a wide range of applications, the current implementation of IFS assumes a minimal requirement on user hardware, a character terminal with cursor addressing. Terminals with bitmap graphics and pointer devices such as mice or light pens are becoming cheaper and accessible to a wider class of computer users. On such terminals, it is desirable to have graphical interactions as well as character-oriented interactions. For example, in many applications, menu items can be better represented with pictures than with texts, and their selections can be done faster with a pointer device than with keyboard typing. The two-part design of the interpreter, separating control from interaction, makes possible improvements in the interaction part with a minimal amount of change in the system entire. Perhaps some future version of IFS will be enhanced to make more use of the new hardware.

VI. ACKNOWLEDGMENTS

Since the conception of IFS, I have been fortunate to benefit from ideas and constructive criticisms from many colleagues and courageous, friendly users of various versions of IFS. I thank in particular: J. M. Chambers, E. R. Fisher, R. G. Kayel, C. M. R. Kintala, G. Perlman, R. M. Prichard, W. J. Shugard, and D. E. Swartwout.

REFERENCES

1. G. D. Bergland, "Structure Design Methodologies," Software Design Strategies, IEEE Catalog No. EH0184-2 (1981), pp. 297–315.

2. F. DeRemer and H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small," IEEE Trans. Soft. Eng., *SE-2*, No. 2 (June 1976), pp. 237–43.
3. F. Beichter, O. Herzog, and H. Petzsch, "SLAN-4: A Language for the Specification and Design of Large Software Systems," IBM J. Res. Dev., *27* (1983), pp. 558–76.
4. A. I. Wasserman, "Characteristics of the User Software Engineering Methodology," IEEE Proc. Soft. Proc. Work, (February 1984), pp. 125–29.
5. G. Robertson, D. McCracken, and A. Newell, "The ZOG Approach to Man-Machine Communication," Int. J. Man-Machine Studies, *14* (May 1981), pp. 461–88.
6. S. R. Bourne, "The UNIX Shell," B.S.T.J., *57* (1978), pp. 1971–90.
7. L. A. Rowe and K. A. Shoens, "Programming Language Constructs for Screen Definition," IEEE Soft. Eng., *9* (1983), pp. 31–9.
8. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, N.J.: Prentice Hall, 1978.

APPENDIX A

Partial BNF Specification of the Frame Language Syntax

```
frame ::=              "{f" frame_id frame_args
                            frame_blocks
                       "}f"
frame_blocks ::=       [title_block]
                       [descript_block]
                       {context_block | menu_block}*
title_block ::=        "{t"
                           Title_string
                       "}t"
descript_block ::=     "{d"
                           Description_string
                       "}d"

context_block ::=      "{c"
                            {activity}*
                       "}c"
menu_block ::=         "{m" [cond_pair]
                             {option_block}*
                       "}m"
option_block ::=       "{o" [cond_pair]
                             title_block
                             [descript_block]
                             {activity}*
                       "}o"
cond_pair ::=          [(entry_cond)] [: (exit_cond)]
activity ::=           {context_block | menu_block |
                       question_block | write_block |
                       subproc_block | coproc_block |
                       arithmetics | string_operation |
                       network_transition |
                       change_scope | change_env}*
```

```
question_block ::=    "{q" [cond_pair] ~input_var
                           Question_string
                           [descript_block]
                      "}q"
write_block ::=       "{w" [(entry_cond)] [file_name]
                           Format_text
                      "}w"
subproc_block ::=     "{a" [cond_pair] [proc_args] [>ret_
                             vars]
                           Subprocess_program
                      "}a"
coproc_block ::=      "{p" [cond_pair] program [>ret_vars]
                           ~!End_of_input,End_of_output
                           Message_to_program
                      "}p"
```

APPENDIX B

Figures 7 and 8 are the actual programs for the Reminder and Adder frames of the Reminder Service System.

```
{f Reminder
      {t
      Reminder Service
      }t
      {m : (0 != 0)
            {t
            Please pick a service
            }t
            {o a
                  {t
                  Add a new reminder
                  }t
                  ~call Adder
            }o
            {o d
                  {t
                  Delete old reminders
                  }t
                  ~call Deleter
            }o
            {o l
                  {t
                  List reminders already set
                  }t
                  ~call Peeker
            }o
            {o e
                  {t
                  Exit.
                  }t
                  ~exit
            }o
      }m
}f
```

Fig. 7—The Reminder frame of the Reminder Service.

```
{f Adder
     {t
     Adding a New Reminder
     }t
     {c
          {q : (~date != ~null) ~date
          Date (m/d/y):
               {d
               A date can be entered in the format:
                   month/day/year
               or as 'today', 'tomorrow', and weekdays
               such as 'monday', 'tuesday' and their
               abbreviations such as 'mon', 'tue'.
               }d
          }q
          {q ~time
          Time (h:m):
          }q
          {q : (~content != ~null) ~content
          Content of the reminder:
          }q
          {a (~date != ~null && ~content != ~null)
          add "~content" "~date" "~time"
          }a
     }c
}f
```

Fig. 8—The Adder frame of the Reminder Service.

AUTHOR

Kiem-Phong Vo, M.A., 1977 (Applied Mathematics); Ph.D., 1981 (Mathematics), University of California at San Diego; AT&T Bell Laboratories, 1981—. Mr. Vo is a Member of Technical Staff in the Advanced Software Department. His research interests include combinatorial structures and algorithms, and efficiency issues in software development. Member, ACM, AMS, SIAM.