"Fill-in-the-Form" Programming†

Lawrence A. Rowe

Computer Science Division, EECS Department
University of California
Berkeley, CA 94720

ABSTRACT

This paper describes a new style of programming, called "fill-in-the-form" programming, for the development of interactive database applications. The applications being developed and the application development environment use the same form and menu interface. High level tools are provided to define interfaces for database query/update and for generating reports or graphs. Our experiences with two systems that are based on this programming paradigm are described.

1. Introduction

A wide variety of computer applications fall into the category of *Interactive Information Systems* (IIS). These applications allow several people to access and update data stored in a database. The applications do not involve much computation but they do involve significant user interaction with the application. The typical interface is a form displayed on a video display through which data can be entered or displayed. Example applications are a software bug report system, a journal submission tracking system, or a personnel management system.

For the past several years we have been developing application development environments (ADE) for writing these applications that use a style of programming we call "fill-in-the-form" programming. An application is composed of a collection of *frames* that contain a *form* where data is entered or displayed and a menu of *operations* the user can execute. The user moves between different frames executing operations to perform whatever action is required. An example frame in a software bug report system is shown in figure 1. The frame has operations that allow a user to retrieve (**Query**) or enter (**Append**) bug reports. To enter a bug report, the user fills in the form and executes the **Append** operation. The system provides built-in commands using a keyboard and/or a pointing device such as a mouse to move to different fields in the form, enter and edit data, and to invoke an operation.

The ADE uses the same "fill-in-the-form" interface as the applications being developed. In other words, the ADE is a collection of frames with operations to define frames, forms, and database relations. The ADE provides frame-types that a programmer can use to construct an application:

1. menu frames for specifying menu interfaces,
2. query/update frames for specifying data browsers,
3. report frames for specifying interfaces to generate reports,

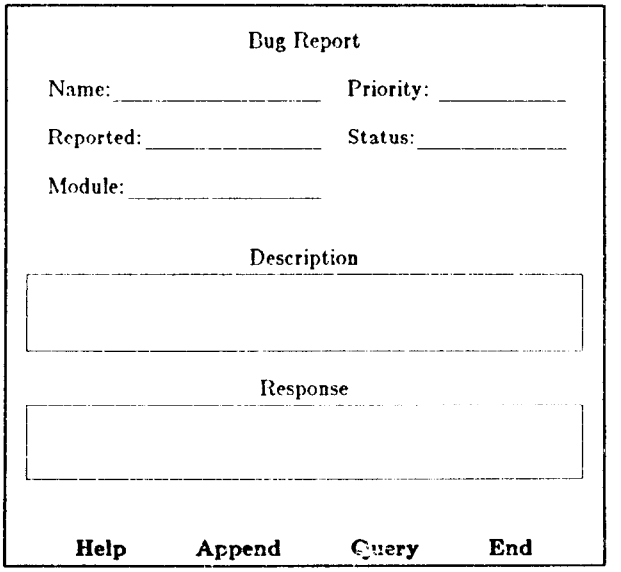

Figure 1. A sample bug report frame.

† This research was supported by the Air Force Office of Scientific Research under Contract 83-0254.

4. graph frames for specifying interfaces to generate graphs, and
5. user-defined frames for specifying interfaces with application specific operations.

Applications are defined by filling in forms that describe how a generic frame should be customized for the specific usage (e.g., for a query/update frame, the form and the mapping between the form and the database). In addition, the ADE allows frames to be tested while they are being defined without requiring that the entire application be compiled and linked.

Two systems have been implemented that are based on the idea of "fill-in-the-form" programming. The Forms Application Development System (FADS) was a prototype system implemented at the University of California, Berkeley to test whether this approach to developing IIS's was viable [RoS82, Sho82]. The prototype was built as a front-end to the INGRES relational database management system [Sta76]. FADS provided only one type of frame (user-defined) but it was readily apparent that applications could be developed quickly and that they were easy to modify. As a result, a local company developed a commercial product, called Application-By-Forms (ABF) [RTI84b], that was based on and extended the ideas in FADS. ABF introduced the notion of other higher level frames (e.g., query/update, report, and graph).

This "fill-in-the-form" approach to developing IIS's can be contrasted with other approaches that are based on extending conventional programming languages or report writer languages. The first approach extends a conventional programming language with constructs to access a database and do screen I/O [HoK84, RoS83, TAN80, RTI84a]. We call such a language a *database/screen programming language*. This approach has several problems. First, programs are much too long because the programmer is forced to specify too much detail. As a result, too much time is required to develop an application and the applications are expensive to maintain and extend. Second, conventional programming languages do not provide support for high-level tools such as reportwriters and database browsers [Cat80, Her80, StK82, RTI84c, Zlo75]. Either these high-level tools must be interfaced to the application program or recoded for each application. Lastly, a database/screen language cannot be used by end-users because too much programming expertise is required.

The second approach to developing IIS's is to extend a report writer language with screen I/O constructs [IBI82, NCS83, Cul83, Cin83, ADR83]. These extended languages are called *Fourth Generation Languages* (4GL). Because a 4GL has an integrated report writer language, they make writing some parts of an application easier. However, the screen I/O constructs are similar to the low-level I/O commands found in the database/screen languages. These systems do not provide high-level commands for specifying operation menus, selecting the code to be executed when a user invokes an operation, specifying help screens, or for specifying other elements of a user dialogue. The user-interface must be coded in the low-level constructs. Consequently, a programmer must write a significant amount of code just to define a simple interface. A second problem with 4GL's is that they are large, monolithic languages which over time have had more and more constructs added to them. The facilities found in high-level tools are typically added to the system by adding constructs to the language. The resulting languages are large and complex and they require considerable training in order to learn how to use them. Another problem is that since these languages evolved rather than being designed, the syntax and semantics are often confusing and inconsistent.

A "fill-in-the-form" ADE is a better approach than these alternatives for several reasons. First, the system has a standard interface. All user-interfaces are defined by frames which standardize how operations are displayed and executed and how data is displayed and entered into the system.

The second advantage of the forms-based approach is that the system supports a collection of high-level tools. For example, tools can be provided for developing data browsers and for defining reports and graphs. These high-level tools make specifying an application easier because the tool has been designed to solve a particular problem (e.g., report writing or graph design). The frame concept is used to integrate these high-level tools into the ADE. In contrast to monolithic languages, a "fill-in-the-form" ADE has several different languages, each customized to a particular usage. Systems composed of several languages typically have two problems: 1) learning the system is difficult and 2) data incompatibility between subsystems. The forms-based interface and online help facilities makes the problem of learning multiple languages less critical. And, the data incompatibility problem does not arise because all data is stored in the database and accessed through a standard query language.

Third, a "fill-in-the-form" ADE allows applications to be developed interactively. The programmer can easily switch between defining an application and running it and he can test partially completed applications. The system also supports a source level debugger that makes it easier to identify and correct bugs.

Another feature of a "fill-in-the-form" ADE is that numerous defaults are provided to simplify the specification of an application. For example, given the definition of a relation, the system will automatically define a form that can be used to query and update data in the relation. Another example is that a default report or graph definition will be generated for a relation or view. By using defaults, applications can be developed very quickly. Moreover, well-designed defaults allow naive users to generate reasonable applications without requiring them to learn the entire system.

The reasons given thus far have focused primarily on prototype applications in which the objective is to get the application running as soon as possible. To achieve this goal, reliability and performance are sacrificed. This trade-off makes sense unless the application will be used in a production environment. For production applications the user-interface must be "bullet-proofed" for naive users and the reports and forms must be fine-tuned to display the data in the best way possible. Moreover, the performance of the application is more critical because it will be run many times or used simultaneously by many users. The ADE's described here have capabilities that allow the programmer to turn a prototype application into a production application. For example, as mentioned above the system allows the programmer to customize a default form or report. Or, if the programmer is willing to bind features of the application, it can be compiled to run more efficiently (e.g., the forms used in an application can be compiled into an executable program rather than being loaded from the database at run-time). Consequently, if the programmer is willing to specify more detail or invest more time compiling the application, he can improve its reliability, performance, and user-interface.

This paper describes the principles behind a "fill-in-the-form" application development environment and presents examples that show how an application is defined. The paper is organized as follows. The second section describes a simple bug report system that will be used to illustrate this style of program development. Section 3 describes the application development environment and shows how different types of frames are defined. Section 4 discusses the productivity improvements possible with a "fill-in-the-form" system. The last section summarizes the paper.

2. A Sample Application

This section describes a sample application. The application is a simple bug report system. The data for this example is stored in one relation with the following schema:

```
BUG( name = text(15),
     priority = (A, B, C),
     reported = date,
     status = (ENTERED, ASSIGNED, FIXED),
     module = text(15),
     description = text(512),
     response = text(512)
   )
```

The *name* attribute is a short name for the bug. The *priority*, *reported*, and *status* attributes describe the relative importance of the bug, the date on which the bug was reported, and the current status of the bug. The *module* attribute identifies to which system component the bug belongs (i.e., an indication of what component has a bug in it). The *description* attribute is a long text description of the bug provided by the person who submits the bug report. The *response* attribute is a long text description filled in by the maintenance programmer after the bug is fixed to indicate how the problem was resolved.

A bug report application is used by many people, including technical support personnel, maintenance programmers, and managers, who are responsible for tracking bugs that are reported and insuring that they are fixed. Figure 2 shows a directed graph that presents an overview of the application. Each node in the graph is a frame and an edge between two nodes indicates that an operation in the frame at which the edge originates calls the frame at which the edge points.

Figure 3 shows the top-level menu frame that is displayed when the application is run. If the user selects the **BrowseBugs** operation, the query/update frame shown in figure 1 is called. The user can browse 'A' priority bugs entered since 1 November 1984 by filling in the form as shown in figure 4 and executing the **Query** operation. Figure 5 shows the screen after the first qualifying bug report has been displayed. Notice that the same form is used to display the data but that a new operation menu is supplied that allows the user to modify or delete the currently displayed bug report or to move to the previous or next bug report. After browsing the bug reports of interest, the user can return to the frame shown in figure 4 by executing the **End** operation. Executing the **End** operation in this frame returns the user to the top-level menu frame.

Figure 6 shows the frame for producing a report that lists bugs in a selected system module that were reported during some time period. This frame is called when the user executes the **BugReport** operation in the menu frame. To generate the report, the user fills in the report parameters and executes the **RunReport** operation. The

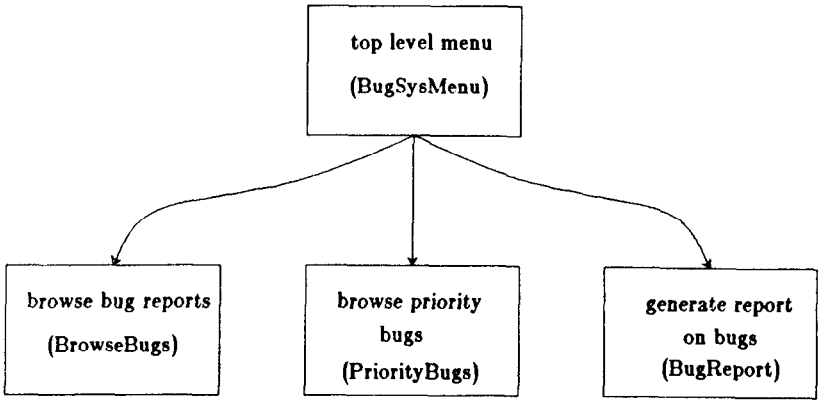

Figure 2. Bug report application overview.

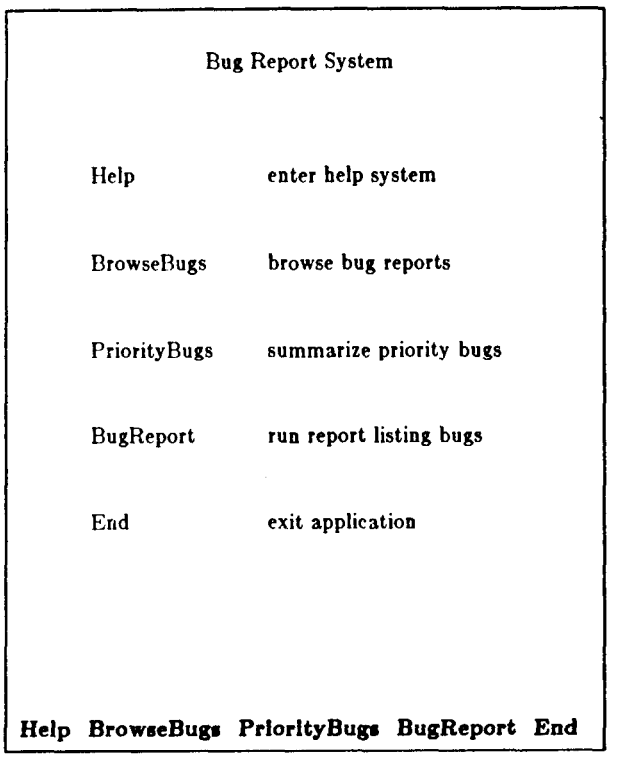

Figure 3. Top level menu.

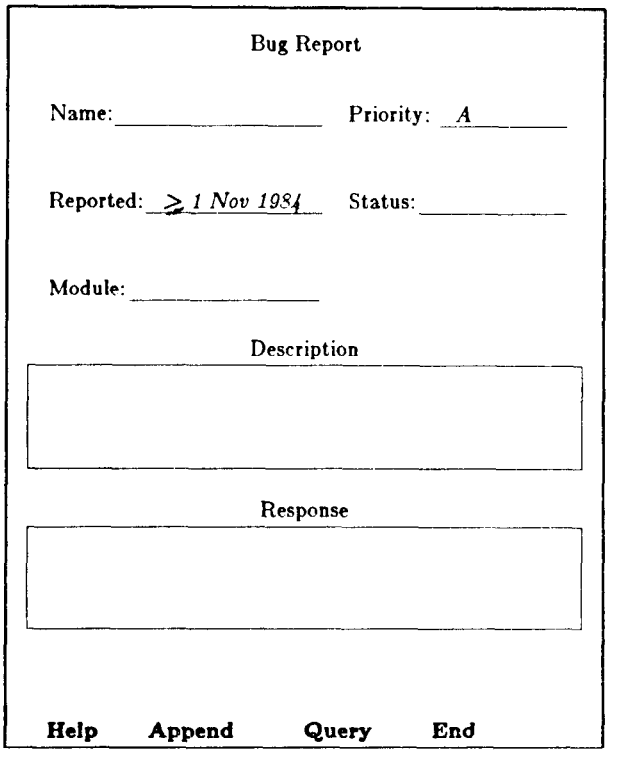

Figure 4. Query specification to browse bug reports.

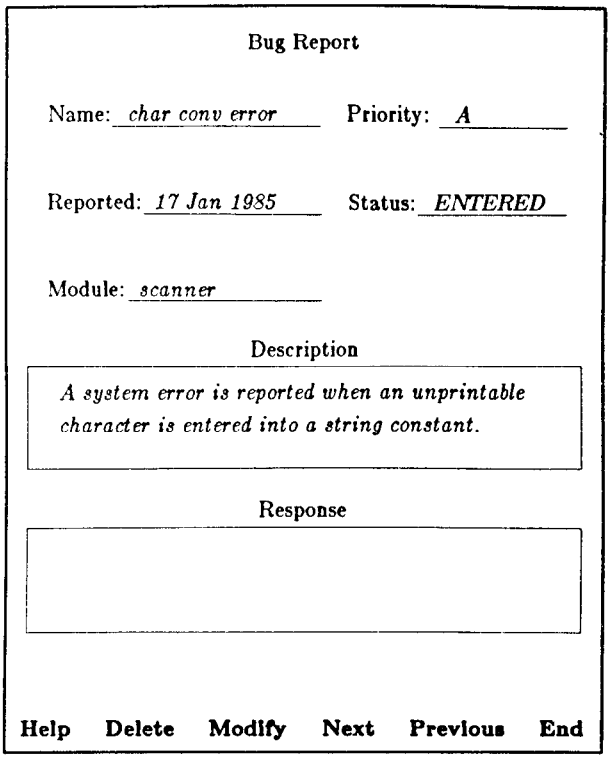

Figure 5. Query/update frame after query is executed.

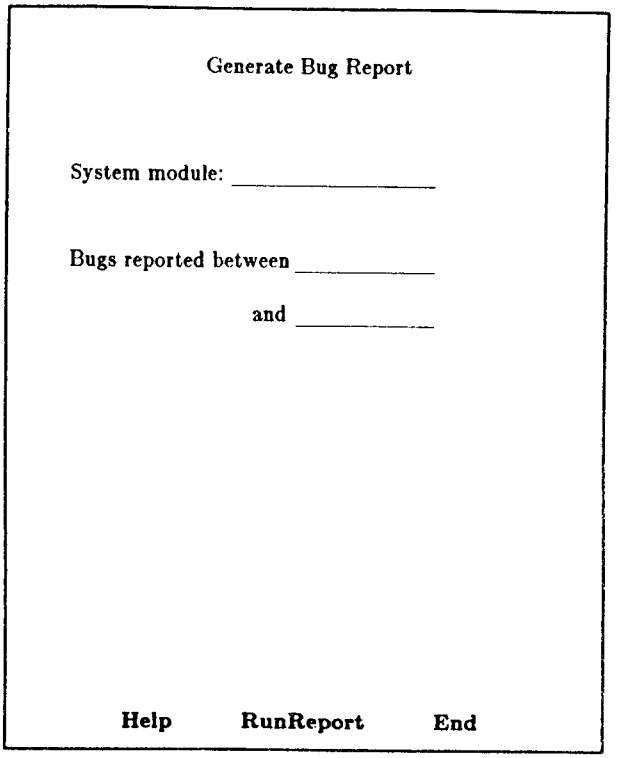

Figure 6. A report frame.

report would be displayed on the user's terminal or spooled to a printer depending on how the application was defined. This frame is an example of a report frame.

This section has described a simple application. The next section shows how this application is defined.

3. The Application Development Environment

This section describes the application development environment and illustrates how frames are defined. Three frame types are discussed: menu, query/update, and user-defined.

The ADE is composed of a collection of frames that the programmer moves between to define the relations, forms, frames, reports, and graphs that make up the application. Figure 7 shows the edit application frame as it would appear after the four frames in the simple bug report application have been defined.

The field labeled *Name* gives the name of the application and is filled in by the programmer when the application is created. The *Creator* field identifies who created the application. This field is automatically filled in by the system when the application is created. The *Created* and *Modified* fields show the dates when the application was created and when it was last modified. These fields are also updated automatically by the system. As will be seen below, this information is kept for every object defined in an application (e.g., frames, forms, reports, etc.). The table in the middle of the frame lists the frames that have been defined for this application. The name and type of each frame is given. A table such as this one is called a *table field* and is actually a window on a larger data set. The user has commands that allow him to scroll forwards and backwards through the table looking at the various entries.

The edit application frame has operations to compile the application for production use (**Bind**), to create and destroy applications (**Create** and **Destroy**), to edit the definition of a frame (**Edit**), to invoke an ad hoc query interface (**Ingres**), and to run the application (**Run**). In addition, the frame includes operations to enter the help system (**Help**) or to exit the current frame (**End**).

The programmer can examine the definition of a frame by selecting the desired frame in the table field and

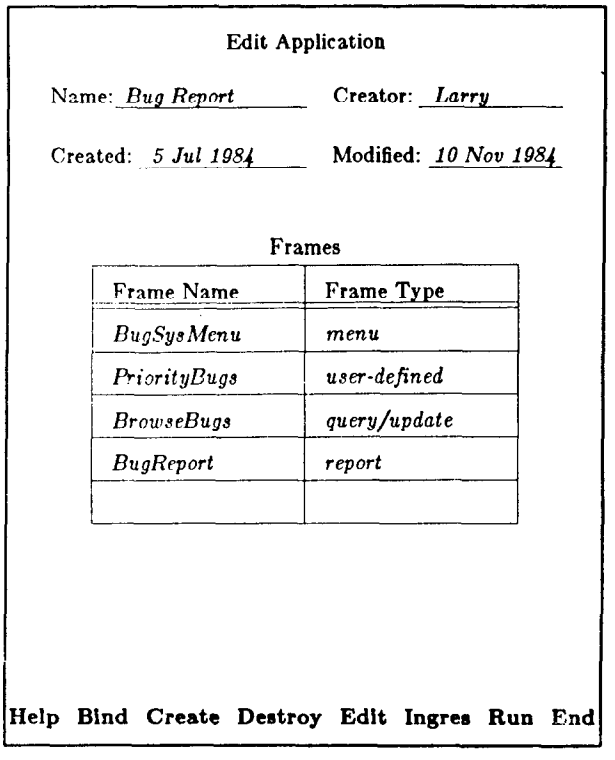

Figure 7. Application definition frame.

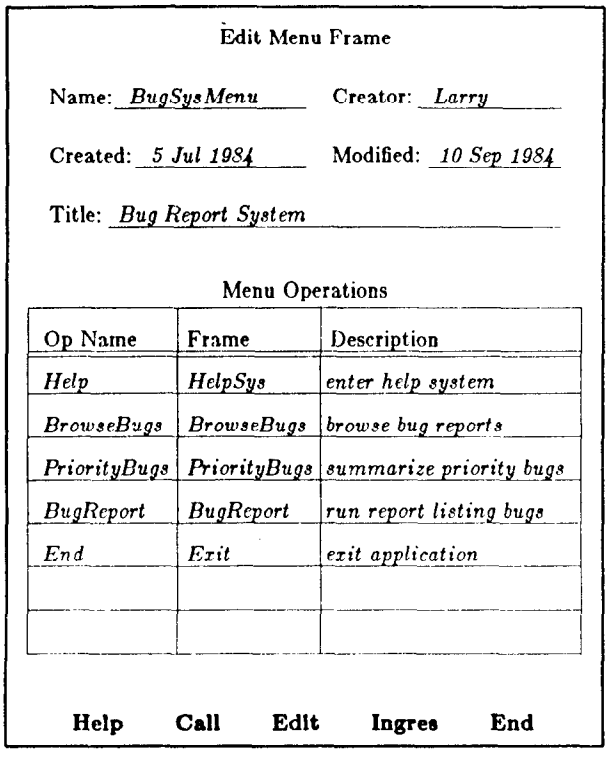

Figure 8. Definition of menu frame.

executing the **Edit** operation. Suppose the programmer selected the top-level menu frame, named *BugSysMenu*, that was shown in figure 3. The definition of this frame is shown in figure 8. Notice that the object information (i.e., frame name, creator, creation date, and modification date) is similar to the information displayed in the edit application frame. Because a menu frame has a predefined structure and operations are limited to calling other frames, the programmer has to specify very little to define the frame. The programmer specifies the title that will be displayed across the top of the frame and, for each operation, he gives the operation name, the frame to call when the operation is executed, and a brief description of the operation. In the menu frame being defined, the operation names are listed across the bottom of the frame and in the middle of the frame along with the descriptions to show the user what each operation does (see figure 3).

The operations provided in this frame allow the programmer to call the frame being defined (**Call**), to edit another object (**Edit**), or to invoke the ad hoc query interface (**Ingres**). The programmer can change the definition of an operation or he can add or delete an operation by modifying the information displayed in the table field that lists the operations.

The structure of a menu frame is fixed by the system. If the programmer does not like this particular structure, he can define a menu as a user-defined frame which allows him to specify the form and to write arbitrary code for the operations. However, the programmer will have to specify more detail to define it.

Query/update frames, like the frames shown in figures 1, 4, and 5, are common interfaces in IIS's. To define a query/update frame, the programmer must specify the form through which the data will be displayed and entered and the mapping between the form and the relations in the database. Figure 9 shows the definition of the *BrowseBugs* frame shown above. The form used in the *BrowseBugs* frame is named *BugForm*. A form is defined or modified by invoking the form editor with the **FormEdit** operation. The form editor is a "what-you-see-is-what-you-get" editor for forms. The programmer can control the definition and placement of fields and descriptive text (e.g., titles, field labels, and other explanatory text) in the form. The form editor also allows the programmer to specify field display enhancements (e.g., inverse video and blinking), edit checks on data entered into a field, and other attributes that control the user interaction (e.g., manadatory fields). If the programmer had not specified a

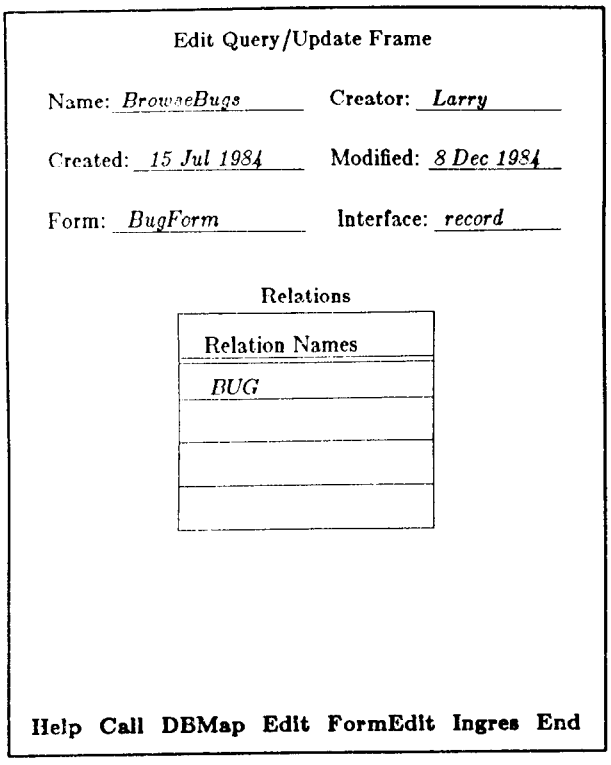

Figure 9. Definition of query/update frame.

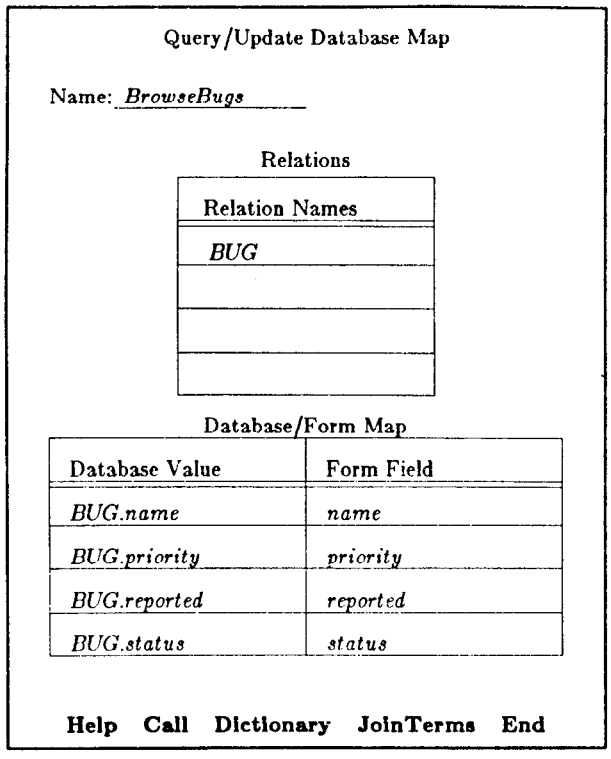

Figure 10. Definition of the database/form map.

form for the query/update frame, the system would automatically generate one for the relations identified in the database mapping.

The *Interface* field defines the style of interface for the frame. In this case, a "record" interface is used which means that only one record in the relation is displayed at a time. And, the mapping between the form and the database is very simple because the data is taken from only one relation, the *BUG* relation.

The mapping is specified in a different frame that is called by executing the **DBMap** operation. The database mapping frame is shown in figure 10. In this simple example, the database values do not involve computation and the relation attribute names and the form field names are the same. By filling in the mapping table field differently, both constraints can be changed. If a more complex mapping between the database and the form is required such as a join between one or more relations, the programmer can execute the **JoinTerms** operation which calls a frame that allows him to specify the mapping.

The system supports two other query/update interfaces: "table" and "master/detail." A "table" interface displays several records through a table field. A "master/detail" interface simultaneously displays one record from one relation through a record interface and several records from a second relation through a table field. For example, suppose there was a second relation that maintained information about the modules in the system with the following definition

```
MODULE(    module=text(15),
           responsible=text(20)
       )
```

Figure 11 shows a "master/detail" interface where the *MODULE* relation is the master and the *BUG* relation is the detail. The mapping between this form and the database must specify that the relations *MODULE* and *BUG* are joined on the *module* attribute and that only the *name*, *reported*, and *priority* attributes in *BUG* are to be displayed. In addition, the programmer must specify whether the "master" record should be deleted when the last "detail" record is deleted.

The problem of specifying complex mappings such as this one is equivalent to the view update problem [Cha75, Day76, Sto75]. ABF solved this problem by constraining what mappings can be specified and allowing the programmer to choose a semantic interpretation. An alter-

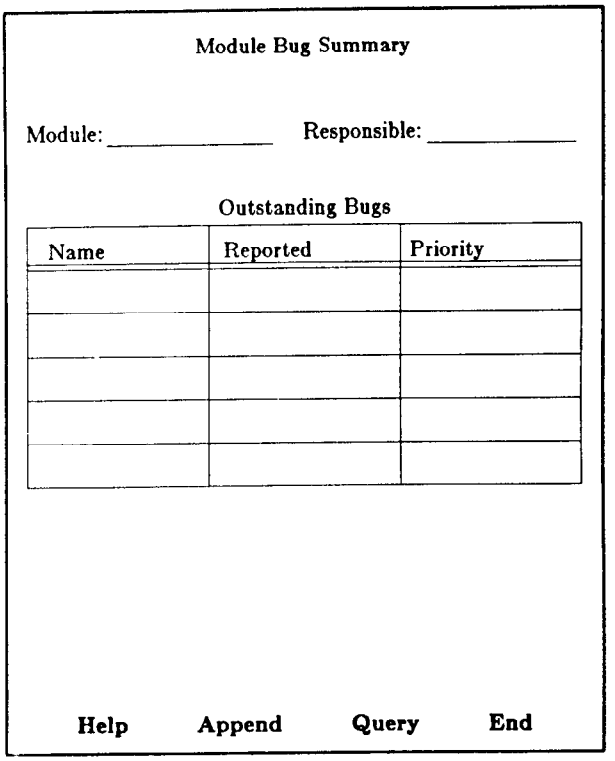

Figure 11. Example of a master/detail interface.

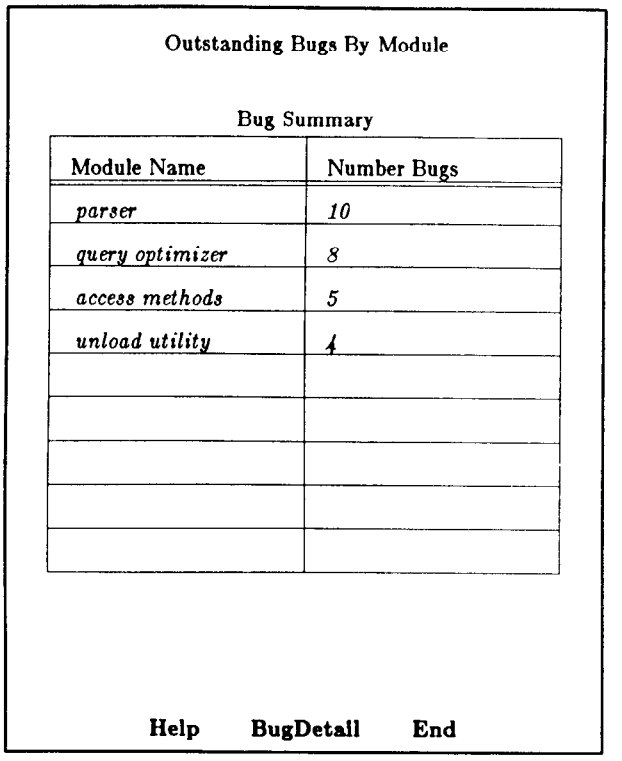

Figure 12. Example of a user-defined frame.

native approach would be to extend the database model so that the correct semantics could be inferred from the database schema.

Query/update frames have proven to be very useful building blocks for IIS's. By providing several choices for these interfaces, query/update frames can be used in more places. The alternative is to force the programmer to specify the interface as a user-defined frame. If a user-defined frame was used, the programmer would have to specify more detail or provide less function at the interface (e.g., the user might not be able to query on arbitrary fields). The "fill-in-the-form" programming environment simplifies the specification of query/update frames and provides the "glue" for integrating them into an application.

The last frame type that will be discussed is a user-defined frame. A user-defined frame gives the programmer complete control over the frame. He specifies the form using the form editor and codes the operations in a high-level programming language, called the *Operation Specification Language* (OSL). Figure 12 shows an example of a user-defined frame. It lists the modules in a software system and a count of the outstanding bugs in each module. The frame has three operations: **Help**, **BugDetail**, and **End**. The **Help** and **End** operations are the standard ones found in most frames.[1] The **BugDetail** operation calls another frame that lists the outstanding bugs in the selected module. Since this frame does not correspond to any generic frame type supported by the ADE, it must be defined by the programmer as a user-defined frame.

The definition frame for a user-defined frame is shown in figure 13. The structure of this frame is similar to the other frames for defining frames. The object name, creator, creation date, and modification date are shown at the top. The frame also shows the name of the form used in the frame and lists the names of the operations that have been defined for the frame. The operation list is displayed through a table field, labeled *Op Names*. Below that field is a text field, labeled *Operation Definition*, that displays

[1] Neither FADS nor ABF made these operations mandatory in every frame. However, they have been included in almost all application frames. An obvious extension would be to include them in the frame model support by the ADE so the programmer would not have to specify them. Frames could be defined that would make it easier to specify help frames and the help text could be stored in the database which would make it easier to manage and allow it to be used in several different contexts (e.g., as on-line help or in manuals).

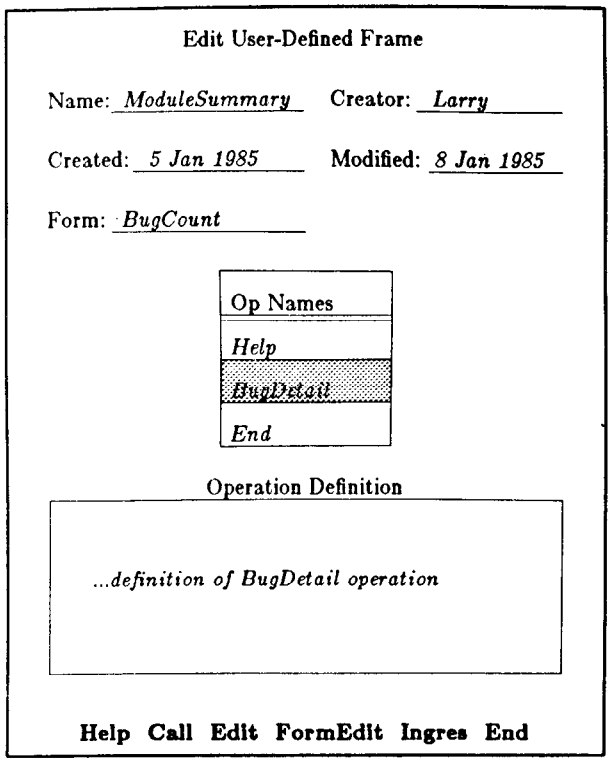

Figure 13. Definition frame for user-defined frames.

the specification for one of the operations. The programmer can edit the specification or select another operation name which causes the definition of that operation to be displayed in the text field. Commands are also provided that allow the programmer to add or delete operations or to change the name of an operation.

Operations are coded in OSL. OSL has constructs for accessing the data in a form or in the database, for calling other frames similar to the way procedures are called in a programming language, and for specifying control-flow for the application. In addition, procedures coded in a conventional programming language can be called so that there is a way to escape if OSL does not provide a required construct. A more detailed description of the features of OSL is given elsewhere [RTI84b].

As powerful as user-defined frames are, the goal of a forms-based programming environment is to use them for only a small percentage of the frames in an application. The reason for this goal is that high-level frames require less specification when they are defined and they allow the programmer to develop his applications in larger "chunks." Consequently, applications can be developed quicker and, to the extent the programmer does not have to wade through code, they can be modified more easily.

This section has shown how menu, query/update, and user-defined frames are defined in a "fill-in-the-form" ADE.

4. Discussion

This section describes the savings that can be achieved when using a "fill-in-the-form" ADE. Following that, future research directions are discussed.

Our experience with FADS and ABF indicates that an application would require 10 to 20 times more lines of code if it had been coded in a database/screen language or a 4GL. This code compaction is due to several factors. First, the frame model for specifying an application subsumes many lines of code that would have to be specified for each frame in an application if it were coded in one of the other languages.

Second, the high-level tools substantially reduce the amount of code the programmer must specify to define part of the application. The high-level tool acts as a program generator that the programmer can parameterize to meet the needs of his specific application. The frame model simplifies the integration of this generated code into the application.

A third reason fewer lines of code are required is that database/screen languages and 4GL's typically have three names for a data value: 1) a name for the field in the form, 2) a name for the attribute in the relation, and 3) a name for a variable that the program manipulates. Many lines of code are used to copy these values between the database, the program, and the form. For example, to display a database value in a form field, two statements are required: one to copy the value from the database to the program variable and one to copy it from the variable to the form field. In contrast, in OSL each field in a form has an implicitly defined variable with the same name as the field. Each time a value is assigned to this variable, the value is automatically displayed to the user through the form. To display a database value in a form field, the attribute is assigned to the form field (i.e., the implicit variable with the same name). Consequently, only one statement is required rather than two. This saving may not seem like much, but if you examine sample applications you will find that is saves many lines of code.

Another way to measure the productivity improvement achieved by a new programming language or system is to quantify the time required to create an application. Using ABF, the simple bug report application described above can be defined in less than 30 minutes. It would take a very sophisticated and experienced programmer to produce this application in the same time with a database/screen language or 4GL.

As we have gotten more experience with these systems it has become clear that there are many different

frame types the ADE might support. We are currently working on a new system that will allow programmers to define their own frame types (i.e., user-defined frame types). With this facility, an organization could customize the frame types for their environment or application. They would get the productivity gains made possible by a "fill-in-the-form" programming environment without having to give up the flexibility found in a general purpose programming language.

Other directions we are pursuing are to define high-level tools for other application domains, such as office automation [Pro85], and to take advantage of new user-interface technologies such as bit-mapped display and mouse interfaces to improve the ADE.

5. Summary

This paper has described a new paradigm for developing interactive database applications. The paradigm is based on the idea of filling in forms to define an application. Two systems have been implemented and experience with them shows this approach to application development to improve programmer productivity and to produce applications that are easier to maintain and extend.

Acknowledgements

I want to thank Joe Cortopassi, Arthur Hochberg, and Kurt Shoens who worked on the development of the ideas presented here and reviewed earlier drafts of this paper.

References

[ADR83] *ADR/IDEAL - Application Development Reference Manual*, S12G-01-00, Applied Data Research, Inc., Aug. 1983.

[Cat80] R. R. G. Cattell, "An Entity-Based User Interface", *Proc. 1980 ACM-SIGMOD Conference on Management of Data*, May 1980.

[Cha75] D. Chamberlin and al, "Views Authorization and Locking in a Relational Database System", *Proc. NCC*, 1975, 425-430.

[Cin83] *Series 80 MANTIS User's Guide*, Release 3.5, Pub. No. P19-0001, Cincom Systems, Inc., 1983.

[Cul83] *Application Development System/Online - Reference Guide*, Release 1.1, Cullinet Software, Aug. 1983.

[Day76] U. Dayal, "On the Updatability of Relational Views", *Proc. 4th VLDB*, 1976, 368-377.

[Her80] C. Herot, "SDMS: A Spatial Data Base System", *Trans. Database Systems*, Dec. 1980.

[HoK84] E. Horowitz and A. Kemper, High-Level Input/Output Facilities in a Database Programming Language, Unpublished manuscript, Univ. South. Calif., June 1984.

[IBI82] *Focus Users Manual*, Information Builders, Inc., New York, NY, 1982.

[NCS83] *NOMAD2 Reference Manual*, National CSS, Wilton, CT, Jan. 1983.

[Pro85] R. Probst, *BOISE: An INGRES-based Office Information System*, MS Report, U.C. Berkeley, May 1985.

[RoS82] L. A. Rowe and K. A. Shoens, "FADS - A Forms Application Development System", *ACM SIGMOD 1982 Int. Conf. on Mgt of Data*, June 1982.

[RoS83] L. A. Rowe and K. A. Shoens, "Programming Language Constructs for Screen Definition", *IEEE Trans. on Software Eng. TSE-9*, 1 (Jan. 1983).

[RTI84a] *EQUEL/C User's Guide*, Version 3.0, VAX/VMS, Relational Technology, Inc., Berkeley, CA, May 1984.

[RTI84b] *INGRES ABF (Applications By Forms) User's Guide*, Version 3.0, VAX/VMS, Relational Technology, Inc., Berkeley, CA, May 1984.

[RTI84c] *INGRES QBF (Query By Forms) User's Guide*, Version 3.0, VAX/VMS, Relational Technology, Inc., Berkeley, CA, May 1984.

[Sho82] K. A. Shoens, *A Form Application Development System*, PhD Thesis, U.C. Berkeley, Nov. 1982.

[Sto75] M. R. Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification", *Proc. SIGMOD* , 1975, 65-78..

[Sta76] M. R. Stonebraker and al, "The Design and Implementation of INGRES", *ACM Trans. Database Systems* , Sep. 1976, 189-222.

[StK82] M. Stonebraker and J. Kalash, "TIMBER: A Sophisticated Relation Browser", *Proc. 8th Very Large Data Base Conference*, Sep. 1982.

[TAN80] "Tandem 16 Pathway Reference Manual", 82041, Tandem Computers Inc., Feb. 1980.

[Zlo75] M. M. Zloof, "Query by Example", *Proc. NCC 44* (1975).