A FORM APPLICATION DEVELOPMENT SYSTEM

by

Lawrence A. Rowe and Kurt A. Shoens

Memorandum No. UCB/ERL M82/38

2 April 1982

ELECTRONICS RESEARCH LABORATORY

To be presented at 1982 SIGMOD Conference

A FORM APPLICATION DEVELOPMENT SYSTEM†

Laurence A. Rowe and Kurt A. Shoens

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

ABSTRACT

This paper describes **FADS** – a Form Application Development System which is an interactive system for the development of form-based database applications. **FADS** reduces the amount of work required to implement a forms application by suppressing much of the detail which would be required by conventional tools (e.g., a screen definition system, a database system, and a programming language). **FADS** provides direct access to a relational database, a standard model of the user interface, built-in form constructs, and an integrated development and debugging environment. Using **FADS**, form applications can be developed quickly and the resulting systems are easy to modify.

A prototype implementation of the **FADS** kernel has been completed.

1. INTRODUCTION

A form application is one in which users interact with a computer through a form displayed on a video terminal. These applications often involve several people sending and receiving forms to communicate and accomplish some goal. An on-line inventory control system, a student enrollment system, a software bug report system, and a purchase order system are examples of form applications.

The Form Application Development System (**FADS**) is an interactive system for developing form applications. It provides built-in facilities to display and enter data through forms and to execute high-level operations coded in a set-oriented query language. These high-level operations correspond to the actions that an end-user performs (e.g., *hire* an employee) when using the application. **FADS** applications are defined by filling in forms using a similar interface to that seen by a user of an application. An integrated development environment is provided that includes an application editor, a screen layout editor, a relation editor, and a debugger.

The goal of **FADS** is to shorten the time required to develop form applications that involve several users. This goal is accomplished by suppressing the detail that an application designer must specify, by having the system automatically fill-in defaults (e.g., screen layout), and by providing an interactive development environment similar to that provided for **LISP** [Teitelman 81].

Previous work on tools for developing form applications can be divided into screen definition systems [Tandem 80, Hewlett-Packard 79] and office automation systems [DeJong 80, Luo 81, Zloof 81]. Screen definition systems such as **Screen COBOL** [Tandem 80] provide embedded language constructs to define screen layouts (i.e., position of fields on a screen) and display attributes (e.g., labels and integrity checks) and to do screen input/output. However, applications are still coded in a conventional programming language and the screen definitions do not access the data definitions stored in the data dictionary even though the same data is being described and manipulated. Consequently, changing an application often requires changes to the screen definition, the database definition, and the program which can be time-consuming and error prone.

Interactive screen definition systems, such as **VIEW/3000** [Hewlett-Packard 79] allow the application programmer to design and edit screen layouts while they are displayed on the terminal. These systems provide a better tool to define screens than embedded languages because the designer can change the screen while looking at it rather than having to compile, link, and run a program to see what the screen looks like. Nevertheless, they have the same problems as embedded languages because they deal only with screen definition. They are

† This work was supported by the National Science Foundation under grant MCS-8006329.

not integrated with the database system and, in some cases, the interface between the program and the screen run-time system is not type checked.

Office automation systems address the same kinds of applications as **FADS** but they either do not provide a complete application development environment or were designed with a different idea of how applications will be developed. **FOBE** [Luo 81] is a nonprocedural query language based on the form data model [Shu 81] that was designed to specify office procedures. In contrast to **FADS**, it does not have an interactive development environment and, as described in [Luo 81], it can not be used to produce applications that are used interactively by an end-user.

SBA [DeJong 80] and **OBE** [Zloof 81] are programming systems for end-users who gradually automate their activities. **FADS** is designed for programmers who develop interactive applications for many users. As with **FOBE**, **SBA** and **OBE** do not provide a complete programming environment.

The remainder of this paper is organized as follows. Section 2 describes the primitives on which the **FADS** system is based. Section 3 presents an overview of the application development environment, describes the interface that the user of a **FADS** application sees, and presents an example of application definition. Section 4 describes the application debugging tools. Section 5 describes the current implementation and extensions to the system. Finally, section 6 summarizes the paper.

The examples below are taken from a bug report system that keeps track of the bug reports handled by a software support group. A relational database design for the system is shown in figure 1. The *BUGS* relation contains a tuple for each reported bug. *Bug* is a short name for the bug and is the logical key of the relation. The *PEOPLE* relation contains information about the programmers and the *BUGASSIGN* relation represents the relationship of bugs assigned to programmers.

The users of the system include a secretary, a manager, and the maintenance programmers. The secretary screens incoming bug reports and enters data about each report into the system (e.g., who submitted the report and a description of the problem). The manager assigns a report to a programmer and sets a date by which the bug should be fixed. If the programmer does not complete the investigation by the due date, first the programmer and later the manager will be notified. After the bug has been fixed, the report is sent back to the manager. The manager then passes it back to the secretary who sends a response to the person who submitted the report.

BUGS (*bug*, program, version, status, date_submitted, submitter, affiliation, date_fixed, description, response)

PEOPLE (*name*, job_title, ...)

BUGASSIGN (*bug*, *name*, date_due, date_assigned)

Figure 1. Bug report system database design.

2. FADS PRIMITIVES

This section describes the primitives used to define a **FADS** application, including: frames, forms, data types, operations, and roles. The primitive objects that comprise an application are stored in the data dictionary.

A **FADS** application consists of a collection of *frames* that the users of the application interact with and move between. A *frame* is what a user sees on the terminal display. It is composed of one or more forms in which data can be entered or displayed and a list of operations that a user can execute. Figure 2 shows a frame that contains one form (a bug report) and five operations (listed at the bottom of the frame).

A user of this frame can enter a bug report into the system by entering data into the fields in the form and executing the *enter* operation. A conventional data entry facility is provided in **FADS** for entering data into a form. An operation is invoked by typing an escape character and a unique prefix of the operation name.

FADS supports two built-in kinds of forms: tuple and relation. A *tuple form* contains one or more fields that can display a tuple in a relation. The form in figure 2 is a tuple form with a field for each attribute in the *BUGS* relation. This form corresponds to a paper form and is called the bug report form.

A *relation form* can display a relation. The tuples in the relation are displayed in a table format. Each row in the table corresponds to a tuple in the relation. Figure 3 shows a second frame that contains two forms: a relation form (labeled "Bug summary") that displays a summary of a collection of bug reports and a tuple form with only one field (labeled "bugs").

Tuple and relation forms are primitives that can be used to define more complex forms. For example, a master-detail form, such as a purchase order, contains a tuple form that displays the data about the purchaser and a relation form that displays the items purchased. This example illustrates how non-normalized data can be displayed.

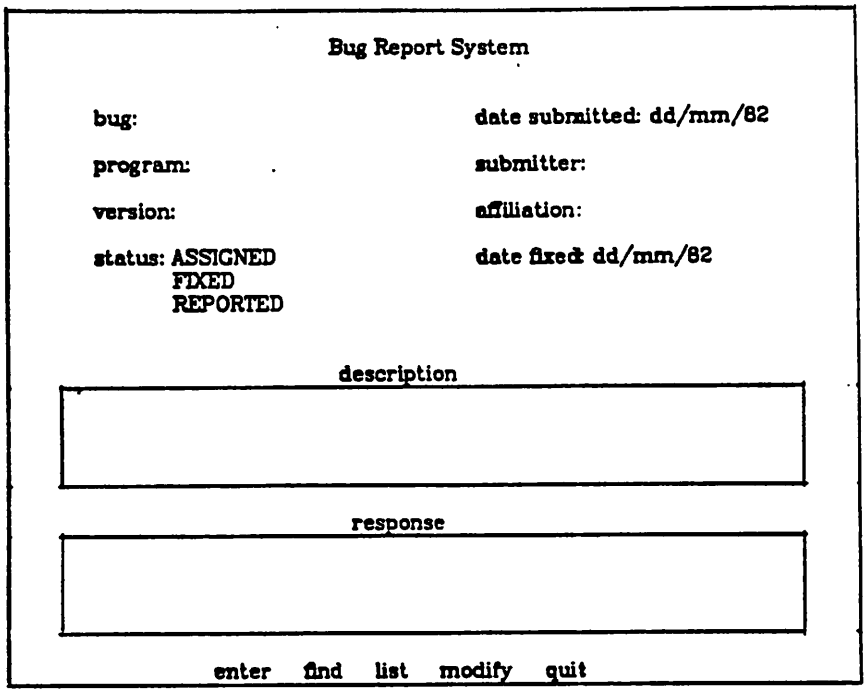

Figure 2. A frame.

Each field in a tuple form and column in a relation form has a data type and display attributes. FADS supports a variety of built-in *data types* including: integers, reals, fixed and variable length strings, dates, times, and enumerated types.[1] Display attributes describe how the data should be displayed (e.g., the label, output format, and display enhancement) and entered (e.g., edit checks, default values, data formats, and input masks such as "dd/mm/82" for dates).[2]

In addition to the simple data types described above, FADS supports two structured data types: tuple and relation. These types describe the values that a tuple and relation form can hold. In a typical FADS application, a tuple type is defined for each entity in the application (e.g., a bug report). This type is then used to define a relation to store entities in the database and forms to display the entities. A tuple form is defined to display a single entity and a relation form is defined to display several entities. If there is a single type definition that is shared by the relation and form definitions, the application designer can add an attribute to the type definition and FADS will automatically extend the relation to include the attribute and the form definition to include an extra field.

Operations listed at the bottom of a frame are defined in an extended set-oriented query language. The language allows a query to access data entered into a form or to display in a form data retrieved from a database. The language also provides statements to call and return from another frame. Arguments such as the output of a query can be passed to a called frame. Lastly, the language includes statements to alter control-flow (e.g., a conditional statement).

Form applications often involve several people sending forms to each other. The extended query language does not provide an explicit statement to send data and a form from one user to another. Communication is accomplished by sharing access to a database. Data is entered into the database by the sender and a condition is set which causes the receiver to retrieve the data. For example, the secretary enters a bug report with status

[1] An enumerated type has a fixed number of values that are represented by identifiers. For example, the *status* attribute in the *BUGS* relation is an enumerated type with values: *REPORTED, ASSIGNED,* and *FIXED.*

[2] In addition to field level edit checks, FADS supports form and frame level integrity checks (e.g., cross field checks).

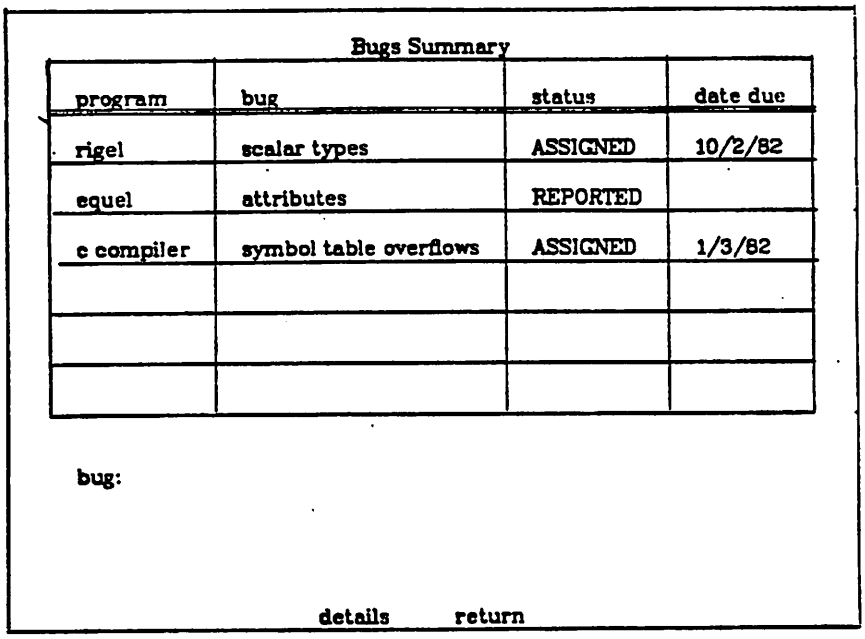

Figure 3. A frame that contains a relation form.

REPORTED. The manager retrieves all *REPORTED* bug reports which effectively sends it from the secretary to the manager.

Using the database to implement communication has several advantages. First, only one copy of the bug report data is maintained which reduces the problem of maintaining consistent copies. Second, a database system provides crash recovery and concurrency control which would have to be implemented for an explicit send statement. And lastly, by storing all data in the database system other tools, such as an ad hoc query language and report writer can be used to access all data controlled by an application.

Different users of a form application typically require access to different data and operations. The data and operations that a particular user requires are defined by their role in the organization. For example, the secretary needs access to the details of all bug reports and operations to enter and correct reports. In contrast, the manager is interested only in summary information (e.g., the number of bug reports assigned to each programmer) and details about selected reports (e.g., bugs which have been reported but have not been assigned to a programmer).

Each user of a **FADS** application has a *role* that determines what frame is displayed when they run the application. This initial frame is called the user's *home frame*. Figure 2 is the secretary's home frame for the bug report system.

This section described the primitives from which **FADS** applications are built. An application is composed of frames which allow users to enter and display data stored in a database. Each frame has a list of operations, specified in an extended query language, that a user can execute. These operations correspond to the actions that the user performs as defined by their role in the organization.

3. APPLICATION DEVELOPMENT

This section describes how a **FADS** application is developed. An overview of the application development environment is given in section 3.1. Section 3.2 shows how a frame is defined and describes the extended query language that is used to define operations. Section 3.3 shows how forms are defined.

3.1. Application Development Overview

An application is defined by moving between frames, filling in forms, and executing operations to define the *objects* (i.e., frames, forms, relations, roles, and data

types) that comprise the application. Figure 4 shows the frames in the application development environment (the boxes) and how the designer can move among them to define an application (the arrows).

The frame titled *application editor* is the top-level description of an application (i.e., it is the home frame for application designers). This frame allows the designer to enter the definition of a new **FADS** application or to modify the definition of an existing application. It has operations to list the names of objects in this application, to edit an object, or to enter the debugger to test the application.

The *list objects* frame displays the names of all existing objects. It helps the designer to recall what objects are defined and it can be used to look up the name of an object.

The frames provided to edit the different kinds of objects (i.e., *edit frame*, *edit form*, *edit data types* including relations, tuples, and enumerated types, *edit relations*, and *edit user roles*) contain forms to define or modify a specific object. For example, the frame for editing frames is used to define the secretary's home frame shown in figure 2. Examples of defining frames and forms are presented in sections 3.2 and 3.3.

Since frames contain forms and forms are defined in terms of the data types they can display, the designer can move directly from defining a frame to defining a form, and from there to defining a data type. Objects can be defined in any order and the system accepts partial definitions. For example, if the designer is defining a frame and specifies that it contains a form which has not been defined yet, he can either complete the frame definition and then define the form or suspend work on the frame definition, define the form, and then complete the frame definition. A *check* operation, described in section 4, is provided so the designer can check the consistency of an application (e.g., that all forms contained in a frame are defined).

From the *application editor* frame, the designer can also call the relation editor or enter the **FADS** debugger. The relation editor is a simple form-based query system in which the designer can examine, modify, enter, and delete tuples in relations to verify the operations of an application. The relation editor is also used to set up test data for an application. The debugger allows the designer to set breakpoints, to run an application (i.e., call an application frame), to examine the state of a running application, and to switch back and forth between editing the application and running it. The debugger is described in more detail in section 4.

Application definitions are stored in system catalogs in the database. This representation enhances the modularity of an application. For example, type definitions stored in the system catalogs are used throughout the application definition (e.g., to define the attributes in a relation and data types of fields in a form). This representation also encourages the designer

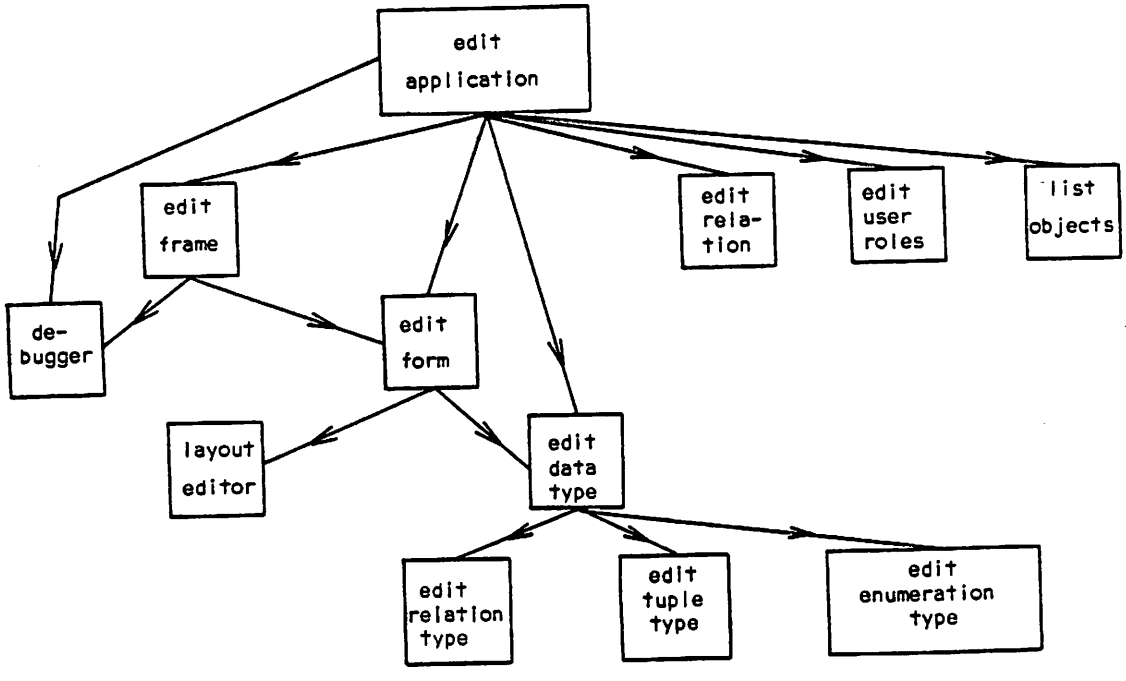

Figure 4. Application development environment.

to reuse pieces of an application (e.g., frames and forms) which provides a more consistent user interface because information is displayed and entered in the same way throughout the application. If frames and forms are reused the size of an application is reduced because fewer objects are defined.

This subsection presented an overview of the frames in the application development environment. They allow a designer to move easily between defining and testing the frames that comprise an application.

3.2. Frame Definition

This subsection describes how frames are defined. It also describes the extended query language used to define the operations in a frame.

A frame is defined by specifying the title, the forms, and the names and definitions of operations in the frame. Figure 5 shows an *edit frame* filled in to define the bug report frame shown in figure 2. The name of the frame is *bugReport*. The frame title is "Bug Report System" which is centered on the first line of the *bugReport* frame. The frame contains one form, named *bugForm*, which is defined in the next subsection.

Operation definitions are coded in an extended query language based on **QUEL**[Held 75]. The language includes statements to retrieve and update values in relations, a notation to reference values in forms, and statements to alter the control-flow of an application (e.g., to call and return from a frame, to execute a statement if a condition is true, and to exit an application).

Edit Frame

Name: bugReport

Title: Bug Report System

Forms

bugForm

Operations

name	definition
enter	**append** BUGS (bugForm.**all**)
find	bugForm = **retrieve** (BUGS.**all**) **where** BUGS.bug = bugForm.bug
list	**call** bugSum(SumForm = (**retrieve** (BUGS.program, BUGS.bug, BUGS.status, BUGASSIGN.date_due) **where** BUGS.status **not**= FIXED **and** BUGS.bug = BUGASSIGN.bug))
modify	**replace** BUGS (bugForm.**all**) **where** BUGS.bug = bug.Form.bug
quit	**quit**

debug delete enter find form layout modify return

Figure 5. Definition of bug report frame.

The remainder of this subsection illustrates the features of the language by describing the definitions of the *bugReport* frame operations.

The first operation shows how data values in a form are referenced. The *enter* operation takes the data entered into the *bugForm* form and appends it to the *BUGS* relation. The expression "bugForm.all" references the values entered into the form. Each form holds the value which has been entered or displayed through it. Tuple forms hold tuple values and relation forms hold relation values. Individual fields in a tuple form can be referenced by using the selection operator (e.g., "formName.fieldName").

In this example, the fields in *bugForm* have the same names as the attributes in the *BUGS* relation so the system can determine which field should be assigned to each attribute in the relation. If the names had been different, explicit assignments would be required. For example, suppose the field in the form which contained the name of the bug was named *bugName*. The assignment of a field in a form to an attribute in a relation could be specified as follows

append BUGS(bug = bugForm.bugName,...)

The second operation shows how queries are parameterized and how the result of a query is displayed in a form. The *find* operation takes the value entered into the *bug* field and retrieves that bug report. The *retrieve*-expression on the right-hand side of the assignment returns a tuple which is assigned to, hence displayed in, the form. The relation attributes are assigned to the form fields with the same names and are automatically converted to the appropriate data type and edited for output as specified for the form field. If the attribute names and fields were different, explicit assignments would be required.

In this example, only one tuple could be returned by the query because *bug* is the logical key of the *BUGS* relation. If the query were different and more than one tuple were returned, **FADS** prints a message to that effect and allows the user to step through the values. Because this facility is built into **FADS**, the application designer does not have to write code to test whether more than one value was returned and to step through them.

The third operation shows how a frame is called and a value is passed to a form in the called frame. The *list* operation calls the frame shown in figure 3 and passes to it the bug reports that have not yet been fixed. The definition of the operation shown in figure 5 calls the frame named *bugSum* and passes to it the bug report data retrieved from the database. The data passed to the frame is displayed in the form, named *sumForm*, which is the relation form in the *bugSum* frame. Notice that the retrieve statement includes a join and that data from both relations is displayed.

The *modify* operation shows an example of a **replace**-statement that updates a bug report and the *quit* operation shows the **quit**-statement that exits the bug report application.

This subsection showed how a frame is defined and described the extended query language for defining operations in frames.

3.3. Form Definition

This section shows how forms are defined. The definition of the bug report form (*bugForm*) contained in the *bugReport* frame defined in section 3.2 is used as an example.

A form is defined by specifying the title, the kind of form it is (i.e., tuple or relation), the data type that can be entered or displayed through it, and information specific to each kind of form.

An *edit form* frame filled in with the definition of *bugForm* is shown in figure 6. This form is a tuple form defined for the data type *bugType*. *BugType* is used to define the relation *BUGS* and this form.

The form layout (i.e., the placement and order of the fields on the terminal display) is defined by a layout format. In this example, the format is "2 columns" which indicates that the fields should be arranged into two columns as shown in figure 2. Other formats provided for tuple forms are: "packed" (pack as many fields on a line as possible) and "tabbed" (separate fields by tabs). Relation forms have only one layout format – the table format shown in figure 3.

Layout formats are useful when building a prototype frame or when a frame will only be used a few times because the forms are easy to specify. A layout editor is provided that gives complete control over the form layout to the application designer. It displays a form or frame as it would appear on a terminal display and allows the designer to move fields around and to change labels and other display attributes. This combination of layout formats and a layout editor allows the designer to get a frame running quickly and to have complete layout control.

The data type *bugType* that is used to define *bugForm* is a tuple type. A tuple type is defined by

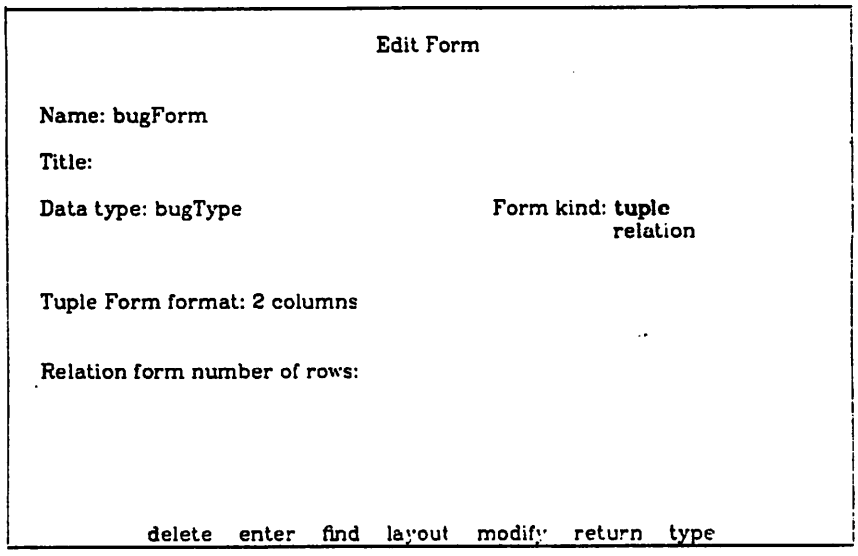

Figure 6. Definition of the *bugForm* form.

specifying the name, type, and display attributes of each of its fields. Data types are defined by filling in forms in the same way frames and forms are defined.

Figures 5 and 6 and the *bugType* data type define the bug report frame shown in figure 2. *BugType* is also used to define the *BUGS* relation and, in the complete definition of the bug report system, *bugForm* would be used in several different frames.

Section 3 has described the application development environment for **FADS**. The development environment includes tools to edit an application definition and the database on which the application runs.

Application definitions are specified by filling in frames that describe the frames, forms, data types, operations, relations, and user roles that make up an application. Using a form-based interface to define applications makes defining an application easy because the form indicates to the designer what information is required and input integrity checks can detect errors when the definition is entered.

FADS applications can be defined quickly because entering data (i.e., an application definition) into a form is fast and because the system provides reasonable defaults for much of the detail that would be required if the application were coded in a programming language. For example, default display attributes (e.g., labels, formats, and input integrity checks) are supplied for fields based on the name and type of the field and default screen layouts are supplied for forms.

FADS provides a standard frame organization (e.g., operations listed at the bottom) and built-in facilities for recognizing which operation was requested and for calling the appropriate operation definition. These definitions are coded in a high-level, nonprocedural query language that does not require redundant declarations of the data types, forms, and relations used in an operation. Because the number of lines of code required to define an operation is reduced and because the data types in the form fields are checked against the database data dictionary, the operation definitions are quicker to write, less prone to errors, and easier to debug when an error is made.

The next section describes the debugging tools for **FADS** applications.

4. APPLICATION DEBUGGING

This section describes the *check* operation that checks the semantic consistency of a **FADS** application and the debugging frame which is called by the designer to test an application or by the system when a run-time error is encountered.

The *check* operation performs many consistency checks on an application. It ensures that

- all called frames exist,
- all forms used in frames are defined,
- operations are legal (e.g., form, field, and attribute names used in an operation are valid), and
- data types used to define forms are defined and valid.

The *check* operation warns the designer of possible

problems, but does not stop him from running the application. The **FADS** system is designed to allow applications to be developed incrementally. The purpose of the *check* operation is to remind the designer of incomplete or inconsistent specifications.

If a **FADS** application is running and an error is encountered (e.g., a frame called in an operation is not defined), the system takes different actions depending on whether the person running the application is a user or the designer. If it is a user, the system prints an appropriate error message (e.g., "Error in the operation definition."), and returns to a consistent state (e.g, returns control to the user).

If the user is the application designer, the *break* frame is called (figure 7). The *break* frame displays the error message that caused the break and the list of active frames. The *break* frame contains operations to control subsequent execution of the application, including breakpoint control; operations to examine or modify the values in any active frame; and an operation to edit an active frame with the application editor described in section 3.

Breakpoints are controlled through a separate frame, called the *breakpoint* frame, which contains a relation form with the breakpoint list. Breakpoints can be set on the execution of a frame or procedure, or on the access or modification of a relation. A breakpoint is set by entering a new row in the breakpoint table and removed by deleting an existing row.

One of the frames in the active list can be selected by entering its name in the "Frame name" field. Then, the *display* operation will display the selected frame and allow the values inside to be modified using the standard form entry editing functions.

The definition of an active frame can be examined or modified with the application editor by selecting one of the frames as for *display* and executing the *edit* operation.

The *query* operation calls the relation editor described in section 3 so that the application designer can examine or modify the contents of the database.

The *continue* operation resumes the application after a breakpoint. The *restart* operation restarts the application from scratch.

These application debugging facilities allow the designer to develop an application incrementally. It is not necessary to write the entire application before testing a portion of it. A piece as small as a single-frame can be written separately and tested. If the frame references an undefined object, the definition of the object can be specified using the *edit* operation of the *break* frame, and the application resumed. This section

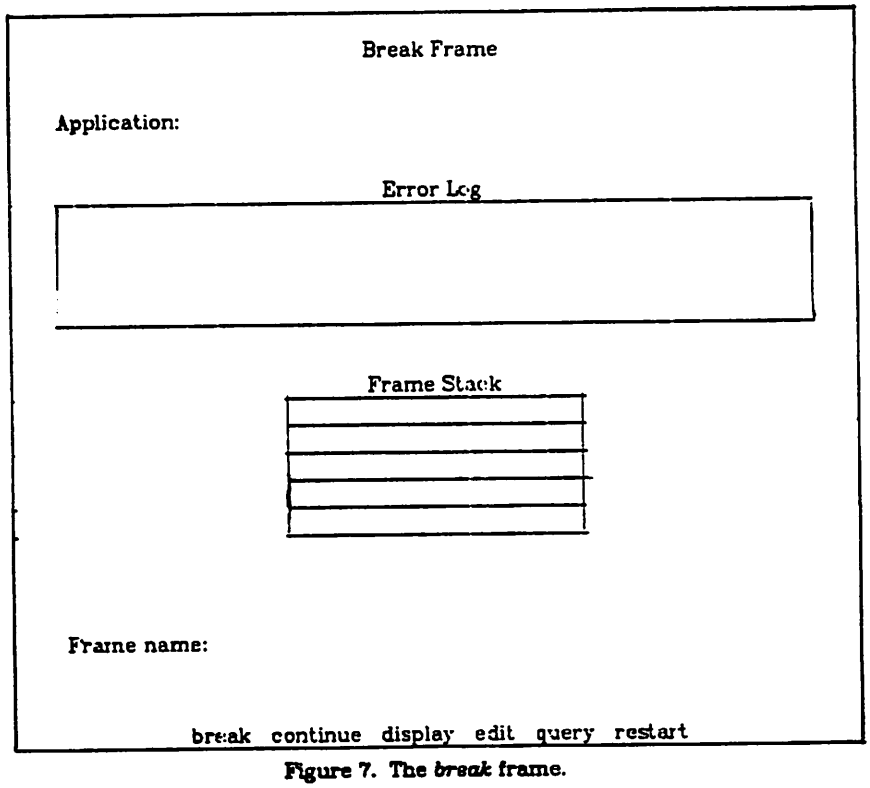

Figure 7. The *break* frame.

described the facilities provided for testing and debugging a **FADS** application. These facilities provide convenient access to all the tools a designer will need: a relation editor, an application editor, a break point package, and access to the intermediate values in the application.

5. CURRENT IMPLEMENTATION AND FUTURE EXTENSIONS

This section briefly describes the implementation of **FADS**, the current status of the system, and the extensions we plan to make to it.

The **FADS** system runs as two processes: the database system (**INGRES**) and the **FADS** kernel. The kernel has three components: the frame driver, the executive, and the frame cache. The *frame driver* uses a terminal independent abstraction [Arnold 81] which handles all input/output between a user and a terminal. This terminal abstraction allows **FADS** applications to run on any alphanumeric terminal with cursor addressing. The frame driver displays frames and the values in the forms inside, accepts input into forms, and passes operation requests to the **FADS** executive.

The *FADS executive* controls a running application. It loads and runs frames, executes queries, calls compiled procedures, and other control flow statments. The *frame cache* maintains an in-core copy of object definitions from the database to improve system performance.

The system is coded in C (approximately 15,000 lines) and runs on DEC VAX-11's [DEC 76] under the Virtual Memory UNIX[3] operating system [Babaoglu 79]. The INGRES interface is coded in EQUEL [Allman 76]

The current system has implemented the features described in the previous sections for running **FADS** applications. These features include: entering and displaying data through forms (tuple and relation forms), executing frames and operations, and executing the internal representation of an operation (running parameterized queries, displaying the output of a query in a form, and calling frames and procedures with arguments). The application development frames have been implemented and we are currently experimenting with the system.

The current system can be extended and improved in three general directions: adding new kinds of forms, using a personal computer for the user-interface, and providing more application development tools. These directions are discussed in the remainder of this section.

[3] UNIX is a trademark of Bell Laboratories.

Many applications have forms which change depending on the data entered into a field. A simple example is an employee form with a marital status field. Depending on the marital status, different information is required (e.g., for married employees the spouse's name may be required while for divorced employees only the children's names, if any, are required). Popup forms which are displayed only if a specified condition is met (e.g., status is married) provide good feedback to the application user indicating what fields require values. The problem with popup forms is that the frame definitions which contain them are harder to understand because the forms displayed vary. Nevertheless, we plan to experiment with popup forms and frames for displaying their definitions.

Another kind of form which is currently being added to the system is graphical forms. These forms allow graphs, bar and pie charts, and scatter plots to be displayed. We are using a color graphics terminal so we will also be looking at display enhancements based on color.

The second set of extensions involves using a personal computer with a bit-mapped display and a mouse for the user interface. Besides using the mouse to select operations, values in enumerated type fields, and tuples in relation forms, we want to allow more user control of the amount of screen space used by a field or form. We also want to move the frame driver component, and possibly the entire **FADS** kernel described above, into the personal computer.

The last area to explore is in the development of more application development tools. Because the application description is stored in the data dictionary in the database, it will be very easy to develop tools which show the application at varying levels of detail. For example, a graph showing what frames call which other frames can be displayed. Or, the designer could ask questions like "what frames have operations which can update relation *R*?"

Another possibility is to display in real-time where each entity (e.g., bug report) is in the system. These higher level abstractions will make applications easier to write and maintain because the developers will not have to wade through as much detailed code to discover what the system is doing.

6. SUMMARY

This paper has described the Form Application Development System and shown how it can be used to develop interactive, form-based database applications

quickly. **FADS** provides an integrated development environment that includes an application editor, a screen layout editor, a relation editor, and an interactive debugger.

Acknowledgements

We would like to thank Joe Cortopassi, Tom Morgan, John Ousterhout, and Mike Stonebraker for their comments on an earlier draft of this paper.

7. REFERENCES

[Allman 76] E. Allman, M. Stonebraker, and G. Held, "Embedding a Relational Data Sublanguage in a General Purpose Programming Language," *Proc. of a Conf. on Data: Abstraction, Definition, and Structure*, pp. 25-35 SIGPLAN Notices, (March 1976). Special issue

[Arnold 81] K. Arnold, *Screen updating and Cursor Movement Optimization: A Library Package*, Department Electrical Engineering and Computer Sciences, U. C. Berkeley (1981).

[Babaoglu 79] O. Babaoglu, W. Joy, and J. Porcarj, "Design and implementation of the Berkeley virtual memory extensions to the UNIX operating system," in *UNIX programmer's manual, seventh edition*, U.C. Berkeley (December 1979).

[DEC 76] DEC, *VAX-11/780 Architecture Handbook*, Digital Equipment Corporation (1976).

[DeJong 80] S. P. DeJong, "The System for Business Automation (SBA): A Unified Application Development System," *Informatin Processing*, (1980).

[Held 75] G. Held, M. Stonebraker, and E. Wong, "INGRES — A Relational Data Base System," *Proc. AFIPS 1975 NCC* 44 pp. 409-416 (1975).

[Hewlett-Packard 79] Hewlett-Packard, "HP 3000 Computer System VIEW/3000 Reference Manual," 32209-90001,
Hewlett-Packard (1979).

[Luo 81] D. Luo and S. B. Yao, "Form Operation by Example — a Language for Office Information Processing," *Proc. SIGMOD Conf.*, pp. 212-223 (June 1981).

[Shu 81] N. C. Shu, V. Y. Lum, F. C. Tung, and C. L. Chang, "Specification of Forms Processing and Business Procedures for Office Automation," Research Report RJ3040, IBM Research Laboratory, San Jose (February 1981).

[Tandem 80] Tandem, "Tandem 16 Pathway Reference Manual," 82041, Tandem Computers Inc. (February 1980).

[Teitelman 81] W. Teitelman and L. Masinter, "The Interlisp Programming Environment," *Computer Magazine* 14(4)(April 1981).

[Zloof 81] M. M. Zloof, "QBE/OBE: A language for office and business automation," *Computer Magazine* 14(5) pp. 13-22 (May 1981).