Freeform: A User-Adaptable Form Management System

Roger King
Michael Novak

Department of Computer Science
University of Colorado
Boulder, Colorado 80309

Abstract

Freeform is an office form management system based on the belief that a form should easily adapt to the needs of each of its users. Freeform uses a high-level data model in order to make relationships between items on a form more visible to the user. The description of a form, as well as the actual data, is stored in Cactis, an object-oriented database. By building more meaning into a form, Freeform makes it possible for a user to make all modifications to a form design by interactively editing the form. This includes modifications that affect the underlying database schema. Freeform thus allows users with little database knowledge to create and maintain private versions of a form while automatically maintaining integrity and consistency between versions.

1. Introduction

Freeform is a menu-based form management system running on a Sun workstation under the UNIX operating system. Freeform provides a highly interactive interface for both design operations and data operations on forms, however, the emphasis is on designing forms. A novel aspect of Freeform is that it allows end users to easily adapt a form to their individual needs. For this reason, we introduce the concept of *form families*. A form family is a set of different versions of the same form. End users may create a private version of the form and add it to the family. Constraints on the form family are specified by the owner of the master version and automatically enforced by the system. A mail system is also provided to allow communication between form users. The mail system may be used to send mail manually and to specify events or classes of events that cause mail to be sent automatically. In order to make form descriptions easier to understand and work with, an object-oriented data model is used to represent forms and an object-oriented database is used for the storage of form descriptions and data.

For many years, forms have been an important method of communication in the office environment. It has even been suggested that forms are the most natural form of interaction between a user and data [Lef79, SLT82]. A form can be thought of as a structured input/output medium. Traditionally, a designer created a paper form and the users filled in the form by putting data in the appropriate areas. Data storage consisted of putting the paper form in a file cabinet. Recent research has led to a large increase in the use of electronic form systems, where the electronic form (displayed on a computer terminal) is used as the input/output medium and a database is used to store data. We have also seen systems such as COUSIN [HaS83], a form-based interactive command interface, several application development systems [Row85, Shu85] that use form-based interfaces, and some database languages such as QBE [Zlo75] that use a form-like interface for manipulating a database. These systems merely use a form layout as a tool and are not meant for managing forms. Our interest is in *form management systems* which are used to create, modify, and store descriptions of forms and to do data operations such as filling in and reading forms. We refer to the description of a form as a *form design* and a particular set of data for a form design as a *form instance*. A form design involves both the appearance of the form and information about how objects on the form relate to database schema objects. We will refer to the relationships between form and database schema objects as the *link* between the the form design and the database.

There are two general categories of operations in most form management systems, *modifying* a form design and *using* a form. Using a form refers to data operations such as filling out a form or looking at a form instance. Using a form generally requires no database knowledge, since the form design has already been created and linked to the database. Therefore, using a form is very straightforward on most current systems. Many existing form management systems [HuW84, KGM84, RoS82, Shi84, Tsi80, YHS84, Zlo82] provide a screen-oriented interface for using forms. It is for this reason that our work does not concentrate on using forms.

Although most of the current form management systems [CzE84, HuW84, KGM84, RoS82, Shi84, Tsi80, YHS84, Zlo82] use a relational database, some also allow higher level data structures than just tables (the form equivalent of relations and tuples) for describing a form. SOFTFORM [HuW84] and FORMANAGER [YHS84] use a hierarchically structured data model for forms. Although a form in these systems does contain tables, it need not be just a flat collection of tables. Instead, the form in these systems is tree structured. This allows a form to contain more semantic information than a pure relational model. Even in these systems, modifying a form design often forces the designer to deal directly with the relational database. This is due to the fact that modifying a form design often involves modifying the link between the form and the database. When a form object is added or modified, there is a high probability that the database schema will be affected. For example, if a new form object has no corresponding schema object in the

This work was supported by ONR under contract N00014-86-K-0054.

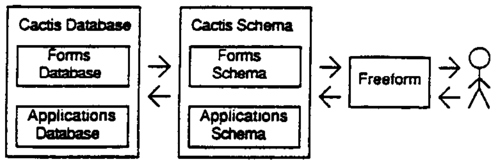


Figure 1

database, one must be created. This is why form design modifications, other than those involving only the physical layout of the design, require some relational database knowledge. For the same reason, some systems [HuW84, RoS82, Tsi82, YHS84] explicitly distinguish between a form designer and a form user.

We feel that form users should be able to modify the form design they are using, including modifications that affect the link between the design and the database schema. To achieve this, Freeform uses a similar object-oriented data model for both the form and the underlying database. This model supports methods and mechanisms for constructing complex objects using semantic database constructs. Since a semantic model conveys more semantic information than the relational model (see [HuK85, KiM85] for a general discussion on semantic database models), this results in a form that is easy for the user to understand. It also means that Freeform can link the form and database schema in such a manner that the user never needs to work directly with the database, even when doing operations that modify the link. This is done by deriving the names of form objects from corresponding database schema objects and by having relationships between form objects mimic the relationships between the corresponding database schema objects. Although this imposes some limitations on how a form design may look, it causes the form design and the underlying database schema to look very similar. This allows Freeform to manipulate the database schema in response to user manipulation of the form design.

Since Freeform provides a highly interactive, graphics-based interface, the user need not be an expert on object-oriented or semantic data modeling. Freeform also differs from systems such as ISIS [GGK85], SKI [KiM84], and SNAP [BrH86]. These systems all provide a graphical interface to a semantic database, while Freeform provides a graphical interface to a form management system. The method used to manipulate forms eliminates the Freeform user's need for a direct user interface to the underlying database.

Since manipulation of the database schema is done automatically, the user may edit a form design by manipulating only the form design. This is true even for operations that involve interaction with the database schema. For example, if a user wishes to add a form object, the corresponding database schema object must be found and the fact that these objects correspond to each other must be stored somewhere. In most systems this involves working directly with the database schema. In Freeform the user may "expand" form objects to see what other objects have a relationship with them. This lets the user view objects in the underlying database schema and add them to the form. Changes in the form-database link are done automatically by Freeform. This makes editing a form design possible even for the user with very little database knowledge.

In order to ensure consistency between the different versions of a form design, Freeform uses form families. The designer of a form creates a master version and specifies what kind of modifications are allowed. For example, the designer may specify that certain objects on the form design may not be modified or deleted, or that an object's value domain must be in a certain range. Other users of the form may create private versions. Freeform makes sure that the versions they create are consistent with the specifications given by the designer. This allows each user to design a version that they are comfortable with and Freeform makes sure it is compatible with everyone else's versions.

The rest of this paper is organized as follows. Section 2 talks about our data model; section 3 describes the user interface; section 4 talks about form families; section 5 describes the implementation; and section 6 discusses future research.

2. The Data Model

Freeform uses an object-oriented data model to represent forms and an object-oriented DBMS called Cactis to store forms [HuK86, HuK87, Kin84]. Freeform users communicate only with

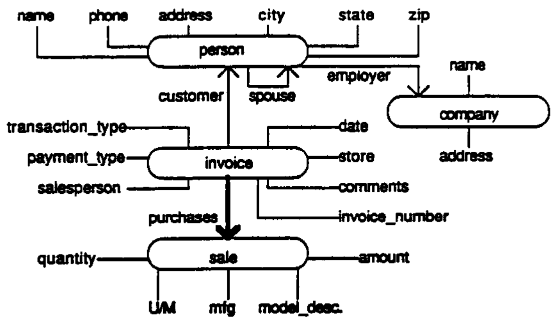
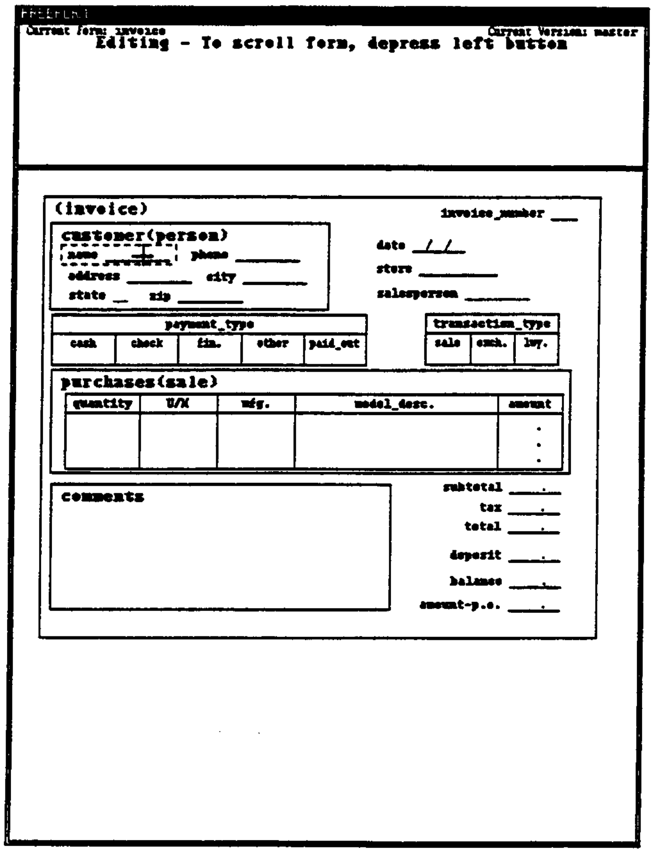


Figure 2



Editing - To scroll form, depress left button

Figure 3

Freeform; Freeform communicates with a Cactis database by using a Cactis schema as its interface (see Figure 1). The forms schema defines how form descriptions are stored and the forms database stores them. A form description contains information about how objects on the form are related to objects in the application database and visual information about the form.

Cactis views an application environment as a collection of *constructed objects*. An object may have *attributes* and *relationships*. Objects and relationships are typed. A constructed object's type is determined by two things: its attribute structure and its *connectors*. A connector allows a relationship to be applied to a certain object. An attribute is an atomic property of a constructed object. These atomic properties may be of any C data type, except pointer. A relationship is a directional mapping from one constructed object to one or more constructed objects. Restrictions such as non-null or unique may also be put on a relationship.

For example, a constructed object called person may have the attributes name, social_security_number, and age, which are all atomic and single-valued. It may also have a connector called children and a connector called parent. The directed relationships my_children and my_parent can be used to connect people to their immediate family. Relationships may be used to pass attributes from one object to another, in order to calculate derived attributes. In this way, the social security number of a person could be passed to a child over the my_children relationship, and used as the value of an attribute called my_parent's_social_security_number.

The Freeform data model is a modified version of the data model used by Cactis. The basic Freeform object type is *form*, which consists of a form name and one or more *form versions*. It is the form version that corresponds to what most people think of as a "form". A form version consists of a version name and a *form object* that describes the version. Often, we will talk about a form version and this form object interchangeably. It is this form object which corresponds to some object in the Cactis schema. For example, the form version in Figure 3 (please ignore the dashed border around the object name at this time) corresponds to the Cactis schema in Figure 2 (Figure 2 is *not* a Freeform screen image). The form shown has more white space then is generally seen on a paper form. Since an electronic form does not have the space limitations of a paper form, we feel there is no need to compress it, but the user may modify the layout if a more compressed form is desired. The enclosed text in the schema drawing represents constructed objects, the unenclosed text represents attributes, thin lines represent one-to-one relationships, and thick lines represent one-to-many relationships. Other form versions could also correspond to the same schema.

The form object is the basic building block in Freeform. A form object may be *simple* or *compound*. A simple form object may be a *field* or a *table*. A field corresponds to a Cactis attribute and may be of type integer, real, boolean, character, dollar, date, unformated text, ordinal or comment. For example, the ordinal field payment_type in Figure 3 corresponds to the attribute payment_type in Figure 2. A comment field has no value associated with it and therefore no corresponding Cactis object, and an ordinal field is merely a field that has a finite number of possible values. A table is a collection of columns, each of which corresponds to a Cactis attribute. The table itself corresponds to a Cactis constructed type that is the range of some one-to-many relationship. For example, the table in Figure 3 (with columns quantity, U/M, mfg., model_desc., and amount) corresponds to the constructed type sale in Figure 2.

Also associated with each field, ordinal type and table column is some information that keeps track of whether it is used for input or output. We will refer to this as its I/O class. A compound form object is a collection of other form objects and may may be considered the Freeform equivalent of a Cactis constructed object.

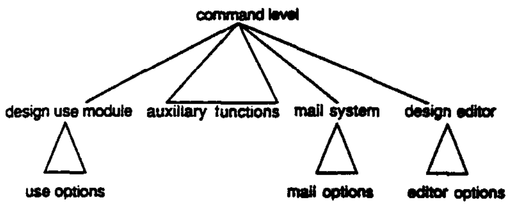


Figure 4

However, rather than using the Cactis notion of relationships, Freeform uses the idea of ownership. Therefore, a compound form object owns all the form objects within it (in Figure 3, invoice owns all the other form objects). Like Cactis attributes, simple form objects may be functionally-defined; as in Cactis, the description of how an object is defined is encapsulated within that object. For example, the total field and all the fields beneath it in Figure 3 are functionally-defined, and maintained by Cactis.

The Freeform model also has some restrictions not found in the Cactis model. First, only the types defined above are used in Freeform. Second, the way Freeform object are connected is more restrictive than the way Cactis objects are connected, since the only kind of "relationship" allowed in Freeform is ownership. The Freeform data model thus has a hierarchical structure, rather than the graph structure of the Cactis data model.

3. The User Interface

A popular line of thought is that there is necessarily a tradeoff between learnability and usability. Several recent studies argue just the opposite [RoM83, WJL85]; the two seem to be congruent. By borrowing some techniques from human factors research, we wish to produce a user interface that is both easy to learn and to use. The high usability and wide acceptance of systems such as the Star [SIK82] and the Macintosh [Rub84] have shown the viability and flexibility of highly interactive, graphics oriented user interfaces. This combined with the increasing availability of graphics hardware have led us to create a menu-driven graphical interface for Freeform. By carefully designing this interface, we hope to minimize two general problems associated with user interfaces [WJL85]. One of these problems is the difficulty involved in navigating within the interface. The second is inconsistency at different levels of the interface.

In order to make navigation within the Freeform user interface straightforward, the interface is organized as the tree seen in Figure 4 (Figure 4 is *not* a Freeform screen image). The user starts in the command level (Figure 5) and chooses an option with one of the command buttons. The option chosen moves the user down a level into the appropriate subtree. For example, selecting the **Edit Current Version** button takes the user into the design editor. When done working at the current level, the user moves back to the previous level. To minimize user confusion, the tree is never more than three levels deep. We also allow no horizontal moves such as going

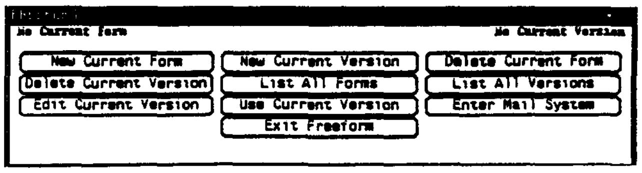


Figure 5

directly from the design editor to the design usage module. In addition to making navigation as simple as possible, this ensures that users complete a task before moving to the next one.

In order to create a consistent interface, interaction is through menus whenever possible. Internal nodes (command level, form editor, form usage module) always use menus, however, leaf nodes (auxiliary functions, edit options, use options) may prompt the user for input, since some tasks are impossible to do with menus alone. For example, at the command level, choices are made with menu buttons. If the **New Current Version** option is selected, a leaf level interface is entered. Now the user is prompted to type in a version name and when this is completed the command level is reentered.

The interface uses two adjoining windows. The top window is used for command buttons, displaying messages and entering text (form names, version names, etc.). The bottom window is used by the design editor, the design use module, and the mail system to graphically display and interact with a form design. In order to avoid confusion, the user interacts with only one window at a time (even though both windows are used by some operations).

When Freeform is run, the top of the screen appears as shown in Figure 5. The buttons other than **Edit Current Version, Use Current Version,** and **Enter Mail System** are for executing the auxiliary functions. They are provided for doing the tasks that do not require interactively working with a specific form design. All the auxiliary functions either require no input or prompt for text input.

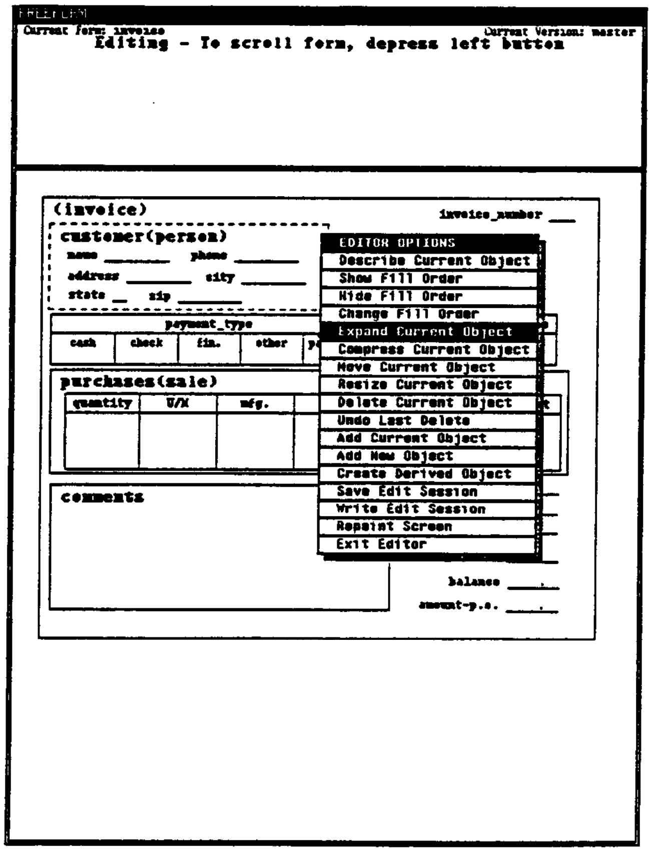
The design editor, design use module, and mail system all provide an interactive interface utilizing pop-up menus and using the principles of direct manipulation [HHN, LeL83] whenever possible. The design use module is used for data entry and retrieval. It allows users to create new form instances and to work with existing form instances (browse, modify, etc.). The mail system is designed to allow the users of a form to communicate with each other. Users may specify events that trigger mail messages to themselves or to other users and they may send mail manually. The mail system is also used by Freeform to notify users of events [ElB82, StR86], such as when a form version is no longer up to date. The design editor interface is where the thrust of our work lies and it will be discussed in a separate section. Since both the design use module and mail system interfaces are consistent with the design editor interface, they will be not discussed further.

3.1. The Design Editor

The design editor is used for creating private versions of a form design. The main features of the design editor are:

- It uses an interactive interface utilizing pop-up menus.
- The user needs very little database knowledge.
- Consistency between versions is automatically enforced.
- Both the appearance and content of a version may be edited.

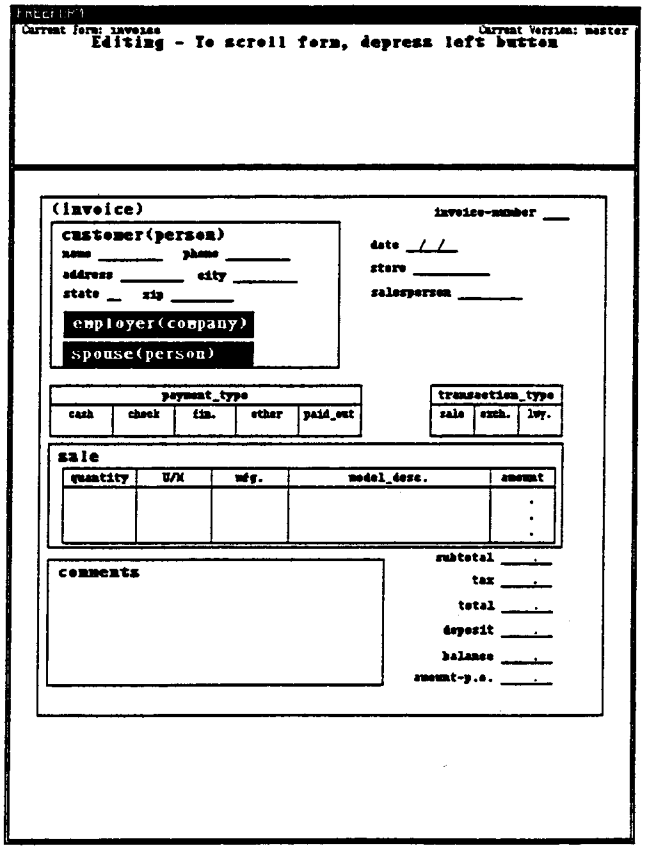
The design being edited is graphically displayed (Figure 3) in a manner that corresponds directly to the way it is stored. Everything owned by a compound form object is displayed inside that object. An example of this is that the form object person owns the form objects name, phone, address, city, state, and zip. The system keeps track of the lowest level object that the cursor is within. This object



Current form: invoice   Current Version: master

Editing - To scroll form, depress left button

Figure 6



Current form: invoice   Current Version: master

Editing - To scroll form, depress left button

Figure 7

is the current object and is surrounded with a dashed border (in Figure 3, name is the current object).

In order to provide the capability for modifying the content of a form design (adding and deleting form objects), the editor must allow the user to manipulate the database schema through the editor interface. Since each compound form object corresponds to an constructed object in the database schema we can find every schema object that has a relationship with this constructed object. **Expand Current Object** does this for the current form object, then creates a corresponding *temporary* form object for each schema object located. A temporary form object is one that does not really exist in the current version. Those form objects that do really exist in the current version are referred to as *permanent*. Temporary form objects are displayed in reverse video and are treated just like permanent form objects while the design is being worked with. These temporary objects may also be expanded. However, **expand** only works for compound forms objects, since simple form objects do not correspond to any constructed object in the database. **Compress Current Object** removes all the temporary form objects within the current form object. **Compress** also works only for compound form objects, except in the following special case. If the current form object is a simple temporary form object, **compress** removes that object. This is not as drastic as it sounds, since removing a temporary object has no effect other than making it disappear from the user's view. The object can also be made to reappear by doing an **expand** on its parent object. By using **expand** and **compress**, the user may browse the database schema as if it is just an extension of the form design being worked on.

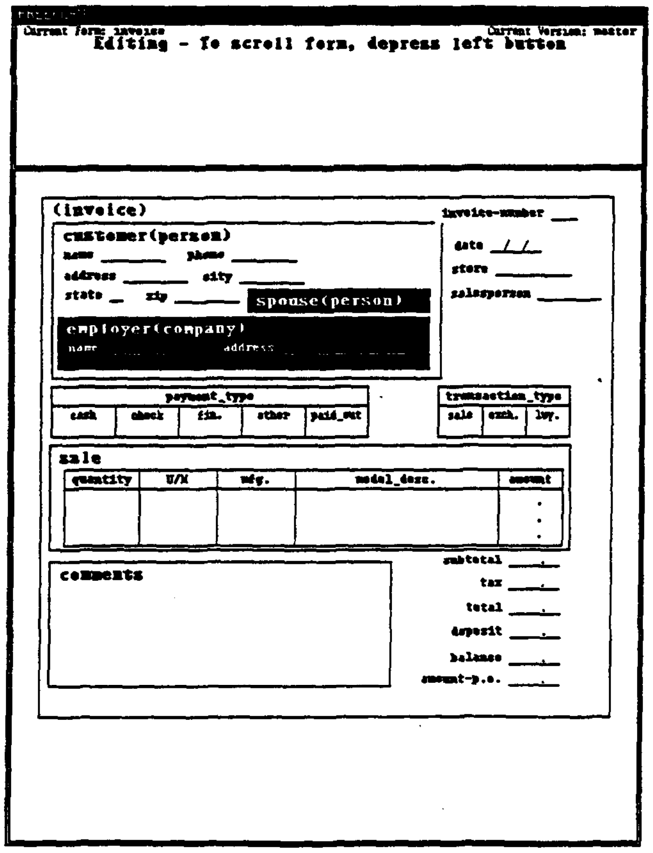

Current form: invoice Current Version: master
Editing - To scroll form, depress left button

Figure 8

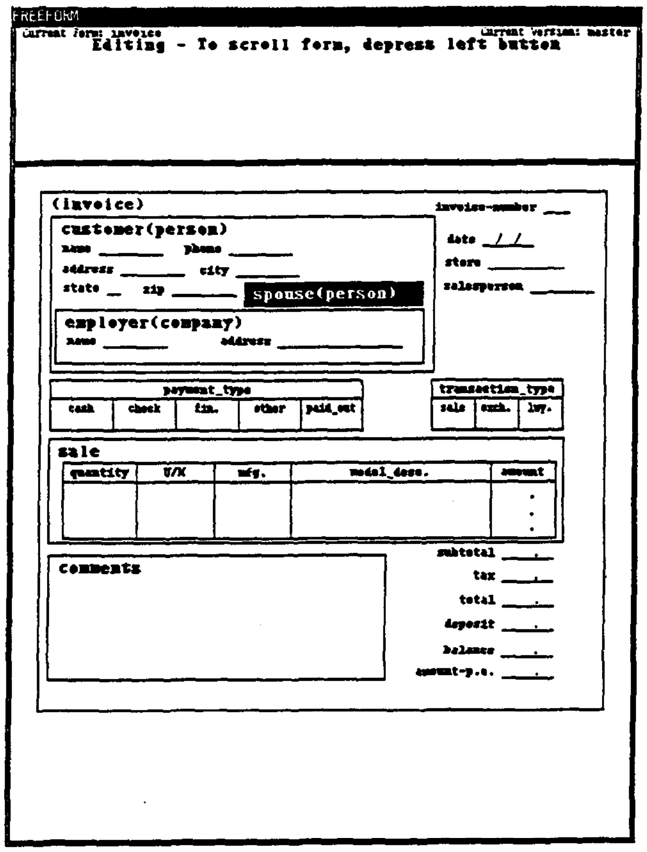

Current form: invoice Current Version: master
Editing - To scroll form, depress left button

Figure 9

The way **expand** and **compress** work is shown in Figures 6 through 8. Given the version in Figure 6 and the corresponding database schema in Figure 2, the **expand** being done in Figure 6 will result in the version shown in Figure 7. Doing an **expand** on this version, with the current form object being employer(company), results in Figure 8. Now doing a **compress** with the current form object still being employer(company) results in Figure 7 again. Freeform automatically creates a new form layout when a form object is expanded or compressed. This layout may be changed by the user.

In addition to being able to look at the schema, the user must be able to incorporate new objects into the form design. This is done by browsing the database schema (using **expand** and **compress**) till the desired object is found, then making it permanent (adding it to the design) with the **Add Current Object** command. This is the preferred method for adding form objects and is shown in the following example. To add employer(company) in Figure 8 to the form design, it is selected as the current object and the **Add Current Object** option is chosen (using the same menu seen in Figure 6). The result can be seen in Figure 9. The form object that owns the object being added is automatically added to the form version (if it is not already there) in order to preserve the hierarchy of the version. When a new form object is added the user is also prompted for certain information such as whether it is an input or output field, restrictions on its value, etc., however, its name, type, etc. are determined from the corresponding type in the database schema. Restrictions on the value of a form object are not allowed to conflict with restrictions that exist for the corresponding database schema object. The way new objects are incorporated into the form design ensures consistency

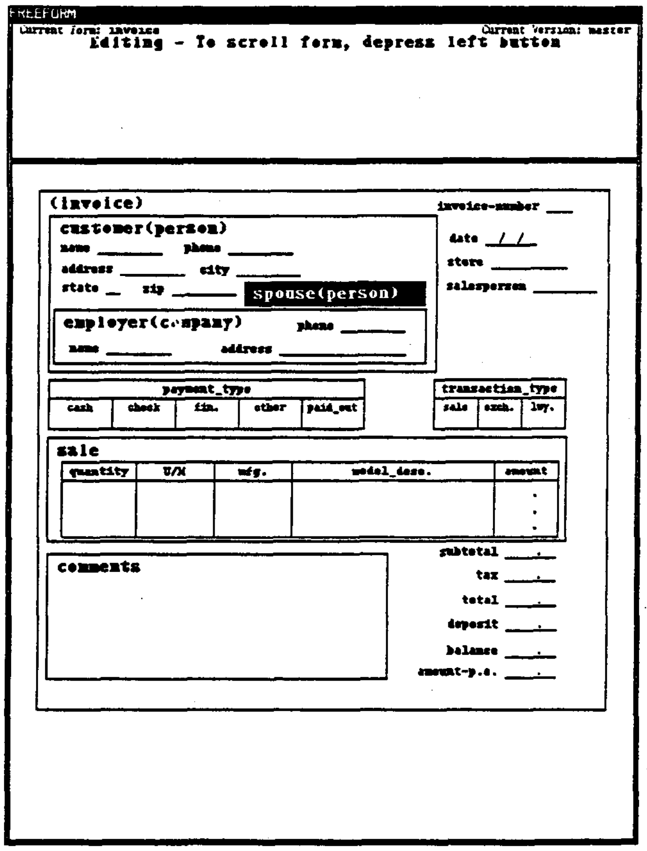


Figure 10

between different versions using the same object.

In the event that the user cannot find the desired object, a new object may be created using **Add New Object**. This command prompts the user for some information about the object (name, type, etc.) and puts the object in a location specified by the user. A corresponding Cactis type and appropriate Cactis relationship is also created. This is done automatically, since Freeform knows where in the database schema the new object(s) must be placed by where the user has located the object on the form design. An example of this can be seen by looking at the version being edited in Figure 9 (this version corresponds to the Cactis schema in Figure 2). To add a new object, the **Add New Object** option is chosen from the same menu seen in Figure 6. When the field phone is added (see Figure 10), the underlying schema changes to look like Figure 11 (Figure 11 is *not* a Freeform screen image). Since **Add New Object** modifies the database schema, its use is discouraged by forcing the user to confirm that this is what is really desired.

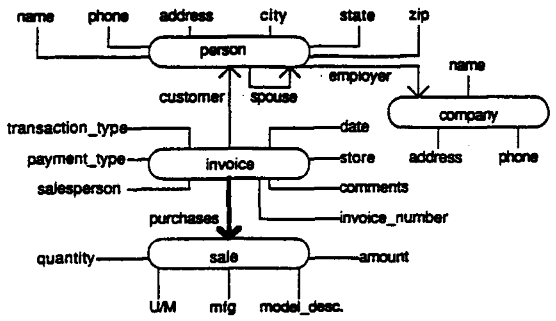


Figure 11

Freeform also provides the user with the ability to interactively create form objects whose values are functionally derived from other objects on the form (see [Shi84] for another form system with this capability). The form objects used may be permanent or temporary. Allowing the use of temporary form objects allows the user to create a form object whose value is based on values of Cactis objects that have no corresponding form object. For example, Cactis may contain the age of each person a department employs while the user may only want to put the average age of a department's employees on his form version.

The editor also provides commands for deleting form objects and commands for changing form layout, fill order, etc. None of these commands affect the corresponding database schema.

4. Form Families

It is through form families that all the different versions of a form design are tied together. A form family is born when someone creates a new form design (the master version) and grows as other users create new versions of it. These new versions are created by starting with an existing version and using the design editor to modify it. The user need not start with the master version, even though the master version determines what is and is not legal on the other versions. It is for this reason that the creator of the master version is also the administrator of the corresponding form family.

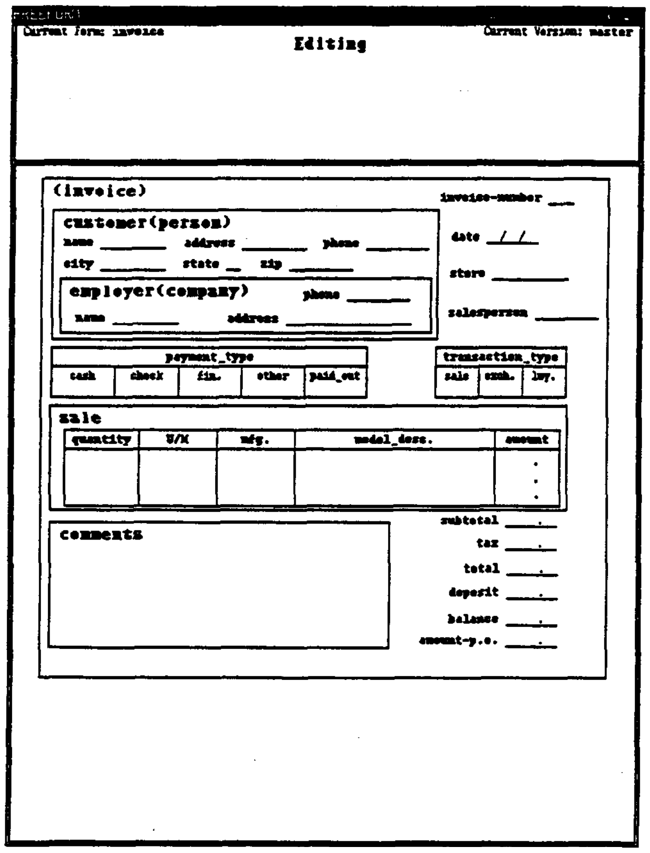


Figure 12

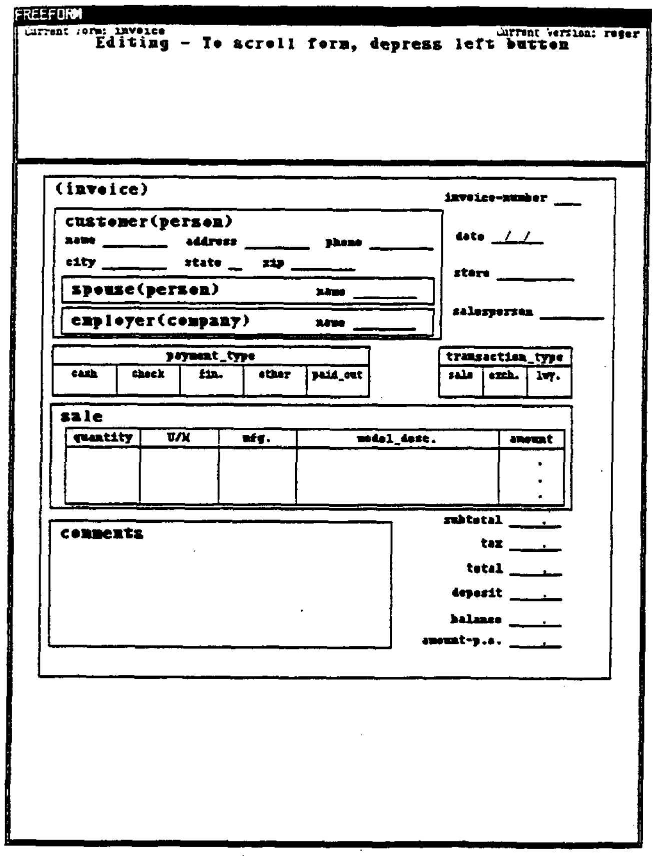


Figure 13

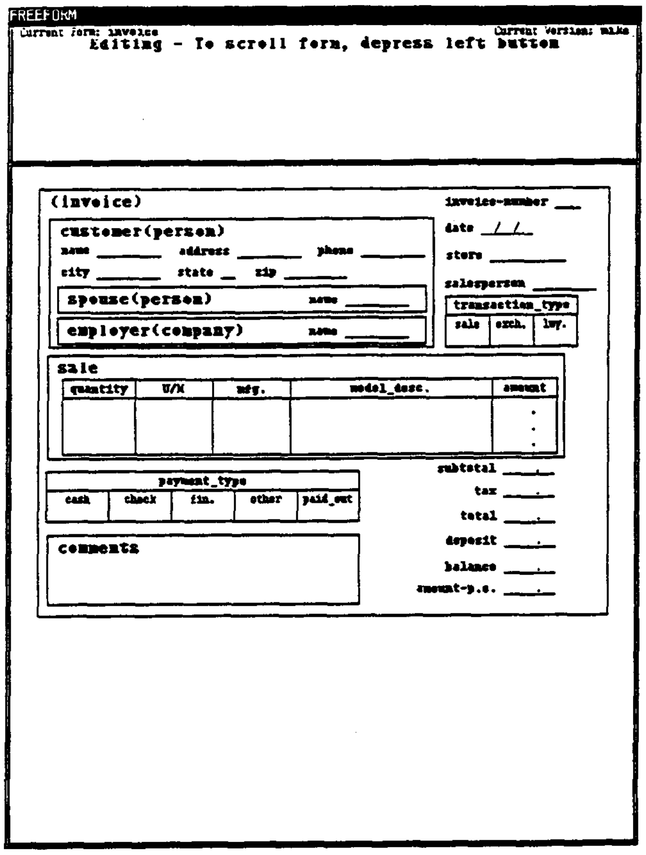


Figure 14

While creating a master version with the design editor, the creator may specify whether a form object is required or optional, what value ranges are legal for a form object when the form is being used, etc. When a user creates a private version these specifications are automatically enforced by the design editor. The administrator may also modify the master version later. The versions that are no longer legal are automatically modified to make them legal. If, during experimental use of Freeform, we discover that automatic modification proves to be too drastic, a method that involves some dialogue between the administrator and the users will be substituted.

How form families grow can best be seen with an example. Suppose that the form administrator creates the form invoice, shown in Figure 12, with all objects except for comments and employer(company) required. Since employer(company) is optional, all the objects within it are also optional. Not all objects within a required object must be required. To prevent careless mistakes while filling out the amount column of the table, the administrator has specified that its value must be greater than 0. The administrator also wishes to be automatically notified when a user adds a new object to any version of the form. This is done with the mail system.

Now Roger is going to create a private version of invoice by starting with the master. The version Roger creates (Figure 13) has no phone and address within the employer(company) object. Also, spouse(person) has been added to the form design, because Roger would like to be able to ask about the customer's spouse by name the next time they come into the store. Adding a new object is always legal because it never causes a form design to contain less information then the master version. Since spouse is not even on the master version it automatically becomes an optional object. This addition causes a mail message to be sent to the form administrator.

Mike is also going to create a private version of invoice by starting with Roger's version. Since comments is optional, there are no restrictions on modifying it and it has been resized on Mike's version (Figure 14). Since it is always legal to tighten an already existing restriction, Mike will require any value entered in the amount column of the table to be greater than or equal to 10, since he sells nothing priced less then $10. The physical layout of Mike's version is also different from that of the other two versions. This is always legal because it has no effect on the content of the form.

The three versions that have been created make up a form family. All three of these forms are slightly different, yet all ensure that when a form is being filled out, all the information that the administrator desired will be provided.

5. Implementation

Freeform is written in C and runs on a Sun workstation under the UNIX operating system. The Sunviews window package, an interrupt driven, window based system, is used to generate all the graphics and for all user interaction. Freeform is also interrupt driven, but only certain interrupts are processed at any given time. This makes Freeform a bit more restrictive than the general Sunviews environment and avoids some of the problems caused by a user being able to do too many things at once. Aside from the mail system, the implementation of Freeform is nearing completion.

When storing form descriptions, we take advantage of the fact that Freeform objects are very similar to Cactis objects and define some new Cactis constructed types and relationships to simulate the

Freeform hierarchy. These new constructs allow Cactis to conveniently store Freeform objects. This lets us store forms without having to rearrange their natural structure. It also allows both form descriptions and data to be stored in the same database.

6. Future Directions

There are a couple of form design features we would like to add to Freeform. The first of these is the ability to create derived form objects using simple predicates or arbitrary C functions. This will be very simple, since Cactis supports both of these methods of creating derived attributes. The second is a facility similar to the **expand** command that allows the user to look at the related objects (including derived objects) on other versions of the current form or even other forms and to add them to the form design being edited.

There are also some features we would like to add that would make filling out a form easier for the user. The ability to include system data [HuW84] such as time, date, etc. is one of these features. It would also be desirable to have certain form objects only appear when other form objects have a certain value when filling out a form [HuW84, Shi84, YHS84, Zlo82]. For example, information about a customer's spouse may be requested only if the customer is married.

Acknowledgements
We would like to thank the Freeform implementation team: Bruce Barker, Elke Duttlinger, Joe Frank, Brenda Howell, Kurt Nagel, Godwill Nelson, Sharon Smith, and Gary Vanderlinden.

References

[BrH86] D. Bryce and R. Hull, "SNAP: A Graphics-based Schema Manager", *IEEE Conference On Data Engineering*, 1986, 151-164.

[CzE84] B. Czejdo and D. W. Embley, "Office Form Definition and Processing Using a Relational Data Model", *SIGOA Conference Proceedings*, June 1984, 123-131.

[ElB82] C. A. Ellis and M. Bernal, "Officetalk-D: An Experimental Office Information System", *SIGOA Conference Proceedings*, June 1982, 131-140.

[GGK85] K. J. Goldman, S. A. Goldman, P. C. Kanellakis and S. B. Zdonik, "ISIS: Interface for a Semantic Information System", *SIGMOD Conference Proceedings* , May 1985, 328-342.

[HaS83] P. J. Hayes and P. A. Szekely, "Graceful Interaction Through The COUSIN Command Interface", *International Journal of Man-Machine Studies 19*, 3 (Sept. 1983), 285-306.

[HuW84] K. T. Huang and C. C. Wang, "SOFTFORM - A Two Dimensional Interactive Form Design Language", *IEEE Workshop on Languages for Automation*, 1984, 229-234.

[HuK86] S. Hudson and R. King, "CACTIS: A Database System for Specifying Functionally-Defined Databases", *Proceedings of the Workshop on Object-Oriented Databases*, Sept. 1986, 26-37.

[HuK87] S. Hudson and R. King, "Object-Oriented Database Support for Software Environments", *SIGMOD Conference Proceedings*, May 1987.

[HuK85] R. Hull and R. King, "Semantic Database Modeling: Survey, Applications, and Research Issues", *USC Technical Report Tech. Rep.-86-201* (April 1985).

[HHN] E. L. Hutchins, J. D. Hollan and D. A. Norman, "Direct Manipulation Interfaces", in *User Centered System Design* , 87-124 .

[Kin84] R. King, "Sembase: A Semantic DBMS", *Proceedings of 1st Int'l Workshop on Expert Database Systems*, Kiawah Island, South Carolina, Oct. 1984, 151-171.

[KiM84] R. King and S. Melville, "Ski: A Semantic-Knowledgeable Interface", *VLDB Conference Proceedings*, Singapore, August 1984.

[KiM85] R. King and D. McLeod, "Semantic Database Models", in *Database Design*, S. B. Yao (editor), Prentice Hall, 1985.

[KGM84] H. Kitagawa, M. Gotoh, S. Misaka and M. Azuma, "Forms Document Management System SPECDOQ - Its Architecture and Implementation", *SIGOA Conference Proceedings*, June 1984, 132-142.

[LeL83] A. Lee and F. H. Lochovsky, "Enhancing The Usability of an Office Information System Through Direct Manipulation", *CHI Conference Proceedings*, 1983, 130-134.

[Lef79] H. C. Lefkovitz, "A Status Report on the Activities of the CODASYL End User Facilities Committee (EUFC)", *SIGMOD Record 10* (August 1979).

[RoM83] T. L. Roberts and T. P. Moran, "The Evaluation of Text Editors: Methodology and Empirical Results", *Communications of the ACM 26*, 4 (April 1983), 265-283.

[RoS82] L. Rowe and K. Shoens, "A Form Application Development System", *SIGMOD Conference Proceedings*, Orlando, 1982, 28-38.

[Row85] L. Rowe, "Fill-in-the-Form Programming", *VLDB Conference Proceedings* , 1985.

[Rub84] C. Rubin, "Macintosh: Apple's Powerful New Computer", *Personal Computing 8*, 2 (Feb. 1984), 56-61, 65-69, 72-75, 79, 81, 85, 199.

[Shi84] Z. Shi, "Design and Implementation of FORMS", *Proceedings of the IEEE International Conference on Computers and Applications*, 1984, 31-36.

[SLT82] N. C. Shu, V. Y. Lum, F. C. Tung and C. L. Chang, "Specification of Forms Processing and Business Procedures for Office Automation", *IEEE Transactions on Software Engineering SE-8*, 5 (Sept. 1982), 499-512.

[Shu85] N. C. Shu, "FORMAL: A Forms Oriented, Visual Directed Application Development System", *Computer*, Aug. 1985, 38-49.

[SIK82] D. C. S. Smith, C. Irby, R. Kimball, B. Verplank and E. Harlem, "Designing the Star User Interface", *BYTE 7*, 4 (April 1982), 242-282.

[StR86] M. Stonebraker and L. A. Rowe, "The Design of Postgres", *SIGMOD Conference Proceedings*, May 1986, 340-355.

[Tsi80] D. Tsichritzis, "OFS: An Integrated Form Management System", *VLDB Conference Proceedings*, 1980, 161-166.

[Tsi82] D. Tsichritzis, "Form Management", *Communications of The ACM 25*, 7 (July 1982), 453-478.

[WJL85] J. Whiteside, S. Jones, P. Levy and D. Wixon, "Performance with Command, Menu, and Iconic Interfaces", *CHI Conference Proceedings*, 1985, 185-191.

[YHS84] S. B. Yao, A. R. Hevner, Z. Shi and S. Luo, "FORMANAGER: An Office Forms Management System", *ACM Trans. on Office Inf. Syst. 2*, 3 (July 1984), 235-262.

[Zlo75] M. M. Zloof, "Query By Example", *Proceedings of the National Computer Conference 44* (1975), 431-438.

[Zlo82] M. Zloof, "Office-By-Example: A Business Language that Unifies Data and Word Processing and Electronic Mail", *IBM Systems Journal 21*, 3 (1982), 272-304.