High Level Form Definition in Office Information Systems

N. H. Gehani
Bell Laboratories, Murray Hill, New Jersey 07974, USA

Several office information systems based on the concept of forms have been built. However, none of these systems provides a high level language for defining forms. In this paper, I will discuss the advantages of a high level form definition mechanism, state the requirements that should be satisfied by such a mechanism, and then propose a high level form definition language based on the concept of abstract data types in programming languages. The proposed mechanism can be used in a form manipulation system. Alternatively, it can be incorporated into an existing language or be the core of a specially designed form manipulation language. The form definition language is illustrated by two exemplary form definitions.

1. INTRODUCTION

Automated office information systems based on the concept of electronic* forms have many advantages over other kinds of systems.[1] Dealing with forms is a major activity of manual office systems. Consequently, one important advantage of form based office information systems is that they allow for a smooth and natural transition from manual office systems to automated systems. Some other advantages of form based systems are:

(i) forms allow logically related data to be treated as an entity
(ii) forms retain many of the properties of paper forms on which most manual systems are based
(iii) forms can be automatically traced, and routed, and
(iv) access to form fields and operations on forms can be restricted to specific classes of users.

Electronic forms combine the virtues of paper forms with the power of the computer and can be made intelligent. They can guide the user in filling out the required information and prevent the user from inserting incorrect information.

Electronic forms can be used as a vehicle to integrate all the different facilities and services offered by office information systems. Theoretical models of office information systems based on forms will aid in the design and analysis of these systems. Such models could be used to analyze the flow of forms, bottlenecks, the work load at a node in the system, forms that flow in infinite loops, and so on. Form flow models have been proposed and research in this direction is continuing.[2,3]

Several office information systems either based on the concept of forms, e.g. OFS[4] and Office Talk,[5] or having forms as important objects, e.g. SBA[6–8] have been built. However, none of these systems provide for a high level form definition language. Only SBA comes close to providing a high level form definition facility.

* I shall use *form* to mean an *electronic form* and shall therefore omit the adjective *electronic* unless there is some ambiguity or an emphasis is intended.

Office Talk provides the user with a forms editor which allows for the specification of the graphical design of the form and the style of each field in the form. The forms editor requires that newly designed forms satisfy certain conditions such as no overlapping fields. It also permits certain fields to be designated as *signature* fields.

OFS provides no mechanism for defining new forms. The system administrator may incorporate new forms into OFS by manually modifying and including code for them.[9]

1.1 Advantage of a high level form definition language

Every office information system based on forms should provide a high level language for defining forms. There are numerous advantages in having a high level form definition language:

(i) It will be easy to define new forms.
(ii) All information associated with a form definition will be in one place.
(iii) Modifying a form can be accomplished easily by changing its high level definition instead of modifying code. Only one module, the form definition, has to be modified instead of several modules implementing the definition as in present systems. Forms can thus be easily corrected and customized.
  (a) The high level form definition will be easy to read and understand compared to reading code defining a form as in present systems.
  (b) An efficient implementation can be produced by a good form definition translator.
  (c) Porting a form from one office information system to another is simply a matter of moving the form definition. No code has to be moved (code is notoriously unportable).

1.2 Forms and abstract data types

A *type* is a set of values plus a set of operations that may be performed on these values. An *abstract data type* is a user defined type. A *form type* is a set of values

CCC–0010–4620/83/0026–0052 $04.00

(corresponding to all the ways in which a form can be filled correctly) plus the operations that can be performed on these values (e.g. edit, check for completeness, copy). A form is an instance of a form type. The high level form definition language I am proposing is based on the idea of abstract types in programming languages. It would allow the programmer to define forms in much the same way one defines abstract data types via *packages* in Ada[10] or *clusters* in CLU.[11]

1.3 Forms manipulation and forms processing

The form definition mechanism can be used by itself in conjunction with a form manipulation system (Fig. 1).

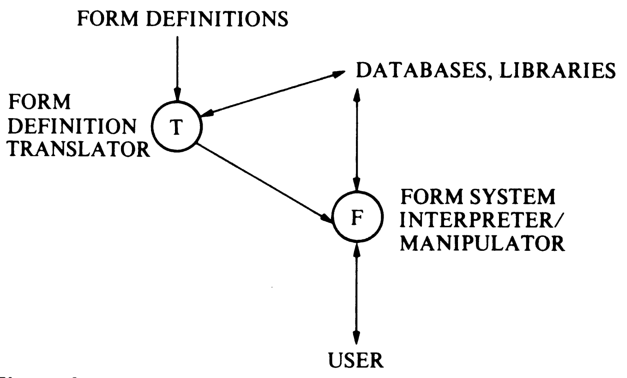

FORM DEFINITIONS
DATABASES, LIBRARIES
FORM DEFINITION TRANSLATOR
T
F
FORM SYSTEM INTERPRETER/ MANIPULATOR
USER

Figure 1

Alternatively, it can be incorporated into an existing programming language or become the core of a specially designed forms manipulation language. The first approach is simpler but has the disadvantage that the form definition and the form manipulation system are written in different languages. Also application programs to process forms are written in a language different from the form definition language. The second approach is more complex but provides an integrated language for defining, manipulating and processing forms.

Business programming languages such as SBA,[6,8] OBE[12] and BDL[13] provide for the processing of documents such as forms. However, like the form based office information systems, they do not provide a high level notation for describing forms.

2. REQUIREMENTS FOR THE FORM DEFINITION LANGUAGE

I had recently proposed the idea of a high level form definition language.[1] I then built a prototype electronic form system to determine the feasibility of a form definition language and to get a better idea of the facilities that should be included in such a language.[14] The prototype relied on hand translated form definitions. A translator for the form definition language was not built since it was not clear what facilities should be included in the form definition language. Construction of the translator was left for the next stage—after the form definition language was specified based on the experience gained with the prototype.

Having built the prototype, I now feel that a form definition language should allow for the specification of all the information of a form type in one module. This encapsulation of all information related to a form in a module makes this information available in one place leading to form definitions that can be easily understood, modified, and so on. This facility would be similar to packages in Ada which allow logically related information to be specified in one place, e.g. the specification of abstract data types, a collection of subprograms and other groups of logically related entities.

It should also be possible to define objects of type form so that the form definition mechanism can be easily incorporated into a programming language to provide for the convenient processing forms. Forms would be full fledged objects like any other object in the language, e.g. integers, queues.

Specifically, a form definition language should provide for:

1. *Specification of the textual information that is to be displayed* to help the user in filling out a form. Heuristic algorithms can then be used to display this information appropriately on different types of terminals.
2. *Specification of the types and other attributes of the fields in a form.* Types appropriate for form definitions should be provided, e.g. date, signature, lock, order, variant. These attributes will be used by the forms system to ensure that the user supplies the right information in the manner specified. For example, a valid date must be filled in a field requiring the birth date, but only after the user has filled in the name field.
3. *Definition of all the operations that can be performed on the form type being defined.* Some form operations come with all forms, e.g. edit, mail. The user should be able to define any other operations appropriate to the type of form being defined.
4. *Association of access rights with the fields.* User rights to operate on fields may be restricted. For example, only the staff of the treasurer's office should be able to fill in fields reserved for *treasury use*.
5. *Association of access rights with the operations.* User rights to operate on forms may be restricted. For example, not everybody should be allowed to destroy forms. (The above access rights could be further refined by specifying the time periods during which users are allowed to update fields and perform form operations and the specific work stations they may use for this purpose.)
6. *Association of pre-conditions and post-conditions with the fields.* A pre-condition is a constraint that must be satisfied before filling a field. A post-condition is a constraint that must be satisfied after a field is filled. These conditions help specify the complex relationships that exist between the various fields in a form. In case of paper forms these conditions are specified as a set of instructions to the user. Verification that correct information has been filled in a paper form is carried out manually some time after the form has been filled out. In case of electronic forms, specification of these conditions will enable the verification to take place automatically and will happen while the form is being filled out.
7. *Association of pre-actions and post-actions with fields.*

Two sets of pre-actions can be associated with every field. A *pre-action* is an action that is performed just before a field is filled. One set of pre-actions is performed if the associated pre-condition is satisfied. Otherwise, the alternate set of pre-actions is performed.
Similarly, two sets of post-actions can be associated with every field. A *post-action* is an action that is performed just after a field is filled. One set of post-actions is performed if the associated post-condition is satisfied. Otherwise, the alternate set of post-actions is performed.

8. *Specification of external routines and names that are being used in the form definition.* For example, the name of a package (i.e. module) containing routines to access an employee data base could be specified indicating to the reader and the form system that it is needed by the form definition.
9. *Definition of local data and routines.* The data and routines used by the rest of the form definition are specified here. These are not available to any user.
10. *Forms processing.* It should be possible for the form definitions to interface with a forms processing language. For example, it would then be possible to write a routine that takes a set of *ExpenseVoucher* forms and determines how many employees have claimed reimbursement for dinners over $30.00.

2.1 SBA and the requirements for the form definition language

SBA provides the user with a mechanism for defining forms based on Query-By-Example.[15] As mentioned before it is the only system that provides a form definition facility that comes close to being high level. However, it does not satisfy many of the requirements listed. The major deficiencies of SBA with respect to the above requirements are

1. There is no notion of field access rights.
2. There is no notion of access rights for operations.
3. There is no separation of condition into pre-condition and post-condition and actions into pre-actions and post-actions.
4. Only limited field types (e.g. no lock, ordered, signature, or variant fields).
5. Same operations are provided for all form types. There is no way to define new form operations.
6. Forms are not treated in the same manner as the data types provided in SBA. For example, instances cannot be passed as parameters like integer objects.

On the plus side, it provides for the inheritance of form definitions (in the manner of Simula classes). For example, one can define a specialized form *book purchase order* based on the definition of a general type of *purchase order*.

3. THE FORM DEFINITION MECHANISM

Based on the requirements for a form definition language, I propose notation of Fig. 2 to describe forms.

```
form xxx is
  imports

    all external procedures and functions used
      in the form definition

  display

    The display text of the form along with the field
      names. The position of the field
      names indicates the positions of the field when
      the form is displayed. The attributes of the fields
      are described below.

  fields

    field attributes, constraints, and actions
      associated with the fields

  operations

    operations that can be performed on forms of this type

  access rights

      fields

        which users can update which fields

      operations

        which users can perform which operations

  local data and routines

    data and routines that are local to the form definition

  end xxx
```

Figure 2

I shall now describe each part of the form definition mechanism in detail,* illustrating its use by developing the definition of the *TuitionReimbursement* form shown in Fig. 3. The *TuitionReimbursement* is used by company employees to have the company reimburse them for the cost of taking job related courses after working hours at a local university. After the form has been filled out by the employee, it must be approved by the employee's project leader and then by the manager of the project.

I shall use an Ada-like[10] notation to describe my ideas. Comments in the form definition (none allowed in the *display* part) begin with '– –' and are terminated by an end of line. Keywords of the form definition language will be depicted in bold font, e.g. **lock, after**.

3.1 Form type

The first line of a form definition specifies the name of the form type, e.g.

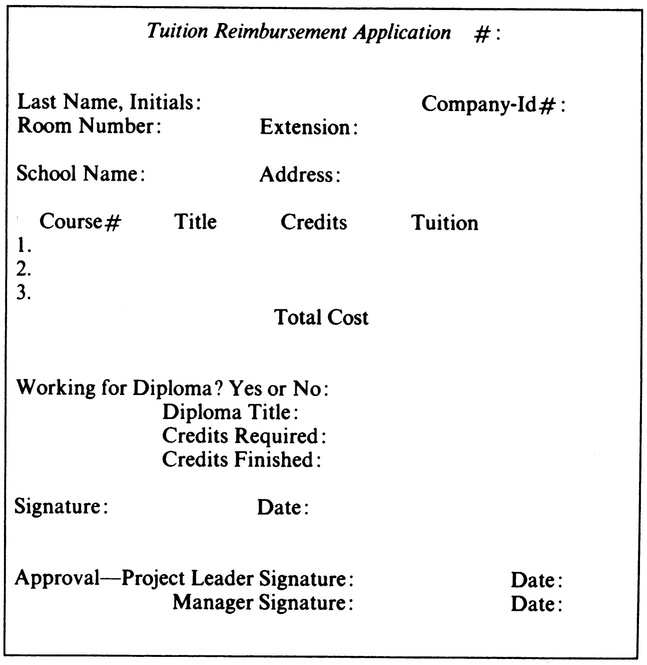
form TuitionReimbursement **is**

3.2 Imports section

The *imports section* contains the names of the packages (Ada terminology used—also known as modules) that will be used in the form definition but are defined outside the form definition.

The *TuitionReimbursement* form definition uses the package EMPLOYEE which provides an interface to

* The exact details of the notation that will be used for the high level form definition language are not important. The important thing is the idea of a high level form definition language.



Tuition Reimbursement Application #:

Last Name, Initials: Company-Id#:
Room Number: Extension:

School Name: Address:

Course#	Title	Credits	Tuition
1.			
2.			
3.			
		Total Cost	

Working for Diploma? Yes or No:
Diploma Title:
Credits Required:
Credits Finished:

Signature: Date:

Approval—Project Leader Signature: Date:
Manager Signature: Date:

Figure 3

the company database. The specification of the package EMPLOYEE is

```
package EMPLOYEE is
  function EMP_NAME(ID:INTEGER) return STRING;
  function EMP_EXT(ID:INTEGER) return STRING;
  function EMP_ROOM(ID:INTEGER) return STRING;
  function EMP_ORG(ID:INTEGER) return STRING;
  function VALID_SIGNATURE(S:SIGNATURE) return BOOLEAN;
           -- validates the signature as being that of the user
           -- logged on to the system
  function DESIGNATION return RANK;
           -- returns the designation of the user
end EMPLOYEE;
```

The declaration of each procedure or function completely specifies the types of the formal parameters and the result so that type checking can be done at compile time.

The fact that the form definition *TuitionReimbursement* uses the package EMPLOYEE is specified by means of the *with* clause as

with EMPLOYEE;

The subprograms inside the package can now be accessed in the form definition by qualifying them with the package name, e.g.

EMPLOYEE.VALID_SIGNATURE($Sig1)

The need for this qualification can be eliminated by means of the *use* clause, e.g.

use EMPLOYEE;

which makes the components of the package EMPLOYEE *directly* visible, i.e. the subprograms inside the package can now be accessed without any qualification (provided there is no ambiguity), e.g.

VALID_SIGNATURE($Sig1)

The *imports* section of the *TuitionReimbursement* form is

```
imports
  with EMPLOYEE;
  use EMPLOYEE;
```

3.3 Display section

Every form has some pre-printed text on it. This text is specified in the *display* section and is interspersed with field names of the form *$identifier*. The text is provided to help the user in filling the form. It also provides semantic information to the reader of the form to allow proper interpretation of the user supplied information. The pre-printed text is specified in the format in which it should appear on the terminal screen. Heuristics will have to be used if the information does not fit on a screen.

The location of the field names (separated by blanks, tabs or new lines to help in recognition) identifies the location of the form fields in the display text. It is at these positions the user will be able to insert information corresponding the fields. These field names are not displayed on the screen. The field names are used (in the next section) to associate information about the type of the fields, the constraint and actions associated with them. The display portion of the *TuitionReimbursement* is defined as:

```
Display
            Tuition Reimbursement Application   #: $No


  Last Name, Initials: $Name              Company-id#: $Id
  Room Number: $Room                    Extension: $Ext


  School Name: $Sname                   Address: $Add


   Course#        Title      Credits      Tuition
  1. $Cn1         $T1        $C1          $Tu1
  2. $Cn2         $T2        $C2          $Tu2
  3. $Cn3         $T3        $C3          $Tu3
                         Total Cost       $Tu4


Working for Diploma? Yes or No: $D
                     Diploma Title: $Dt
                     Credits Required: $Cr
                     Credits Finished: $Cf

Signature: $Sig1                     Date: $Date1


Approval—Project Leader Signature: $Sig2        Date: $Date2
                 Manager Signature: $Sig3       Date: $Date3
```

3.4 Fields

The *fields* section of a form definition contains the type and other attributes of a field, conditions that must be satisfied before a field is filled, actions that must be performed if these pre-conditions are satisfied, conditions that must be satisfied after the field is filled, and actions that must be performed after these post-conditions are satisfied.

3.4.1 Field attributes. A field is specified as having a type in much the same way variables in programming languages have types. These types are CHARACTER, STRING, INTEGER, FLOAT, BOOLEAN, enumeration types, SIGNATURE and DATE.

A signature field is like a string field, except that the sequence of characters inserted by the user are not displayed. Instead the name of the person signing (who must be the person logged on to the system) is displayed.

The system itself will not do any checking. This must be done in the post-condition part of a field as will be shown later. A date field is one which accepts all the usual denotations of dates. In addition to the type attribute, a field may have the following additional attributes (and combinations of these when it makes sense).

(i) **required**—specifies that this field must be given a value for a form to be considered complete.

(ii) **virtual**—specifies that this field will have the value specified by the expression given. This field cannot be filled in by the user. This expression is re-evaluated every time a field it depends upon is changed by the user.

(iii) **unchangeable**—specifies that the field cannot be changed after it has been filled. The field can only be filled if it has the null initial value.

(iv) **tag** and **variant**—the value of this field determines which list of fields, called a *variant* can be filled in by the user. The syntax of the tag and variant fields is

```
case F of
   when v1 = > FL1;
   .
   .
   .
   when vn = > FLn;
end case;
```

where F is the tag field and FL_i are field lists. Each field is described as defined later. If the field F is given the value v_i then the variant FL_i is the active one.

(v) **ordered**—specifies that the field can be filled in only after some other field has been filled in. The syntax used, along with the different options, is

```
after field_name
before field_name
```

(vi) **lock**—specifies that certain fields cannot be changed after this field has been filled. The syntax used is

```
lock field_name_list
lock all above
lock all except field_name_list
```

All fields to be filled in by the user have appropriate null values initially. Thus a form is blank when displayed.

3.4.2 Field definition. All fields are described using the following syntax.

$$\textbf{pre}\{\alpha = > al_{true}, al_{false}\}\ \text{field\_name\_list attributes}\quad \textbf{post}\{\beta = > bl_{true}, bl_{false}\}$$

where α and β are the *pre* and *post* conditions, al_{true} and bl_{true} are lists of procedure calls specifying the actions to be performed if the corresponding pre-conditions and post-conditions are satisfied, and al_{false} and bl_{false} are lists of procedure calls specifying the actions to be performed if the corresponding pre-conditions and post-conditions are not satisfied. The procedure calls in each list are separated by semi-colons.

If the pre-condition is not satisfied then the user cannot fill the field; if the post-condition is not satisfied then the field reverts to its previous value. The user will be informed that a correct value was not supplied for the field. Elaborate error messages can be given to the user by means of procedure calls performed when the conditions are not satisfied.

The = > symbol may be left out if there are no actions to be performed. The **pre**{...} part may be left out if the field is to be filled without any pre-condition and pre-actions. Similarly for the **post**{...} part.

For example, the pre-condition of an amount field in a bank withdrawal form would be amount > 0 and the post-condition will be amount ≥ 0. These conditions would be specified as

pre{amount > 0.0} and **post**{amount ≥ 0.0}

The fields portion of the *TuitionReimbursement* is defined as follows:

```
fields
 $No: INTEGER virtual NEXT();
      - - every form instance gets a unique identification number
      - - got by calling the local routine NEXT
 $Name: STRING(1..30) required;
 $Id: INTEGER range 0..99999 required post{$NAME = EMP_NAME($Id)};
         - - The post-condition specifies that
         - - the name associated in the data base with the Id
         - - supplied must match the name supplied

 $Room: STRING(1..5) virtual EMP_ROOM($Id);
 $Ext: STRING(1..4) virtual EMP_EXT($Id);

         - - The values are determined automatically for the user
 $Sname: STRING(1..25) required;
 $Add: STRING(1..35) required, after $Sname;
         - - School name must be filled in before its address

 $Cn1: STRING(1..5) required;
 $T1: STRING(1..15) required, after $Cn1;
 $C1: FLOAT after T1 post{$C1 > = 0};
 $Tu1: FLOAT required, after $C1 post{$Tu1 > = 0.0};
         - - at least one course must be filled in the form
         - - tuition must be positive

 $Cn2: STRING(1..5) after $Cn1;
 $T2: STRING(1..15) after $Cn2;
 $C2: FLOAT after T2 post{$C2 > = 0};
 $Tu2: FLOAT after $C2 post{$Tu2 > = 0.0};

 $Cn3: STRING(1..5) after $Cn2;
 $T3: STRING(1..15) after $Cn3;
 $C3: FLOAT after T3 post{$C3 > = 0};
 $Tu3: FLOAT after $C3 post{$Tu3 > = 0.0};

 $Tu4: FLOAT virtual $Tu1 + $Tu2 + $Tu3;
         - - the total tuition is computed automatically
         - - to be the sum of the tuition for each course

case $D: (Yes, No) required of
           - - $D is defined to have either Yes or No as a value
    when Yes = > $Dt: STRING(1..20);
                 $Cr: FLOAT post{$Cr > = 0.0};
                 $Cf: FLOAT post{$Cf ≤ $Cr};
    when No = > null;
end case;
      - - only if the user is working for a diploma (i.e., $D = Yes)
      - - is the user allowed to fill the fields $Dt (diploma title),
      - - $Cr (credits required) and $Cf (credits finished)
      - - Credits required must be greater than credits finished
      - - Inapplicable fields will be skipped automatically

. $Sig1: SIGNATURE (1..6) lock all above, required
          post{VALID_SIGNATURE($Sig1)};
              - - if the signature is not valid, an error
              - - will be indicated and the old value
              - - restored
 $Date1: DATE after $Sig1, required;

 $Sig2: SIGNATURE(1..6) after $Date1 post{VALID_SIGNATURE($Sig2)
                  and DESIGNATION () = PROJECT_LEADER};
 $Date2: DATE after $Sig2, required;

 $Sig3: SIGNATURE(1..6) after $Date1 post{VALID_SIGNATURE($Sig3)
                  and DESIGNATION () = MANAGER};
 $Date3: DATE after $Sig3, required;
```

Although in this example the pre-condition part has not been used, its use will be necessary in situations when a field is to be filled only after certain conditions have been satisfied. An example of such a situation is when the amount in bank account must be > 0 prior to withdrawing money from it. Of course, after the withdrawal, the account balance should not be negative. This is expressed as

```
pre{BALANCE($Name) > 0} $Withdrawal: FLOAT post{BALANCE($Name)
                                   — $Withdrawal ≥ 0.0};
```

3.5 Operations

The forms system interpreter provides a number of operations that come with every form definition. Users will be allowed to perform only authorized operations on a form type. This will be in accordance with the access rights specified in the form definition. Some operations that come with every form definition (provided by the forms system) are:

Edit: Used to fill a form.
Create: Used to create a new instance of a form type. An unique identifier for the form is given to the user by the interpreter.
Mail: Mails the specified form to another user.
Find: Returns a list (possibly null) of form identifiers that match the desired characteristics, e.g. a specific string pattern.
Complete: A form is complete if all the required fields have been filled.
Destroy: The specified form is destroyed. In practice it will probably be made inaccessible to the user.
Display: The specified form is displayed.

It must be noted that his mechanism is being designed for use with a system that supports the form definition mechanism, i.e. an interpreter or a language. For additional operations the user can write additional procedures or functions. Inside such a procedure or a function, each form is like a record and its fields are accessed as such.

```
function TOTAL_AMOUNT (F: in TuitionReimbursement) return INTEGER is
  begin
    return F.$Tu4;
  end TOTAL_AMOUNT;
```

The *TuitionReimbursement* form has no operations defined in its *operations* section. The only operations that can be performed on this type of forms are the ones that are provided for all types of forms (stated above).

3.6 Access rights

3.6.1 Fields. This part of the form definition specifies which users can change which fields.

```
case DESIGNATION () is
   when TECHNICAL = > update all except $Sig2, $Date2, $Sig3, $Date3;
   when PROJECT_LEADER = > update $Sig2, $Date2;
   when MANAGER = > update $Sig3, $Date3;
end case;
```

The function DESIGNATION returns the RANK of the user in the organization which is one of the values

(TECHNICAL, PROJECT_LEADER, MANAGER, DIRECTOR, SUPERVISOR, TREASURER, SECRETARY)

In this example, a TECHNICAL employee can update all the fields in a form except the fields which contain the signatures and dates filled in by the project leader and the manager.

3.6.2 Operations. This part of the form definition specifies which users can perform which of the operations available for forms of this type. For example, in this case a manager can perform any operation defined for the *TuitionReimbursement* forms whereas persons classified as other than technical, project leader, or a manager can only display the forms.

```
case DESIGNATION is
   when TECHNICAL = > all except destroy;
   when PROJECT_LEADER = > all except destroy;
   when MANAGER = > all;
   when others = > display;
end case;
```

3.7 Local data and routines

In this section are specified data and subprograms local to the form definition. These are not available outside the form definition. For example, each instance of the *TuitionReimbursement* form must be assigned a unique identification number. The local function NEXT supplies a series of unique identification numbers starting with 1.

```
N: INTEGER := 0;

function NEXT return INTEGER is
   N := N + 1;
   return N;
end NEXT;
```

3.8 End form type

end *TuitionReimbursement*

This just signals the end of the form definition.

The Appendix contains the complete definition of another example form type, the *Cash Advance and Ticket Requisition* form.

4. CONCLUSIONS

The definition and modifications of forms will be simplified by providing a high level form definition language in form based office information systems. High level form definitions will be easy to understand and check for correctness.

The high level form definition language I have proposed was designed on the basis of the experience gained from the prototype electronic form system[14] in which a preliminary version of the form definition language was used. The form definitions in this prototype were hand translated. The next step is the construction of the form definition translator.

Office information systems are inherently distributed systems. There should be no difficulty in implementing the form definition language on a distributed system provided each node in the system has access to the form definitions and the packages imported by the form definition.

The form definition language proposed is quite powerful and can be used to describe a wide variety of forms. It appears that the proposed form definition mechanism will blend easily with a language like Ada or Pascal extended for forms processing because each form instance can be treated as a record. Research in this direction is in progress.

Considerable amount of work research needs to be done to adapt the proposed high level form definition language to the needs of real offices. For example, how would one define that a field X in a form A has the same value as a field in another form B which must be filled to determine the value of X? Or, how would one allow fields to be *unlocked* to allow modification?*

Acknowledgements

I very much appreciate the constructive comments and suggestions of my colleagues C. D. Blewett, J-M. Chang, J. O. Limb, P. Zave and C. S. Wetherell.

* Signature fields are often used to lock a set of fields. One possible solution to the unlocking problem would be that they could be unlocked by the person who signed the form by re-signing the form. This would result in the signature field having a null value and can therefore be signed afresh after modifying the fields that were unlocked.

REFERENCES

1. N. H. Gehani, The potential of forms in office automation. *IEEE Transactions on Communications,* **COM-30**. (1), (January 1982). (Special Issue on *Communications in the Automated Office.*)
2. I. Ladd and D. Tsichritzis, An office form model. *Proceedings NCC,* Anaheim (1980).
3. D. Tsichritzis, Forms management. In *Omega Alpha* ed. by D. Tsichritzis, Technical Report CSRG-127 (March 1981).
4. C. K. Cheung and J. Z. Kornatowski, *The OFS User's Manual.* Computer Systems Research Group, University of Toronto, Ontario, Canada (March 1980).
5. C. A. Ellis and G. J. Nutt, Office information systems and computer science. *Computing Surveys* **12** (1), (March 1980).
6. S. P. de Jong, The system for business automation (SBA): a unified application development system. *Proceedings of IFIP 80* (October 1980).
7. S. P. de Jong and R. J. Byrd, Intelligent forms creation in the system for business automation (SBA). *Research Report RC 8529,* Computer Science Dept. IBM T. J. Research Center, Yorktown Heights, New York 10598 (October 1980).
8. M. M. Zloof and S. P. de Jong, The system for business automation (SBA): programming language. *Communications of the ACM* **20** (6), (June 1977).
9. D. Tsichritzis, private communication. 24 February (1981).
10. *Reference Manual for the Ada Programming Language.* United States Department of Defense (July 1980).
11. B. Liskov *et al., CLU Reference Manual.* MIT/LCS/TR-225, Laboratory for Computer Science, MIT (October 1979).
12. M. M. Zloof, A language for office and business automation. *Office Automation Conference 1980 Digest,* Atlanta, Georgia (March 1980).
13. M. Hammer *et al.,* A very high level programming language for data processing applications. *Communications of the ACM* **20** (11), (November 1977).
14. N. H. Gehani, An electronic form system: an experience in prototyping. Submitted for publication.
15. M. M. Zloof, Query-by-example. *AFIPS Proceedings National Conference* **44**, 431–438 (1975).

FURTHER READING

J. Hogg, O. M. Nierstrasz and D. C. Tsichritzis. Form procedures. In *Omega Alpha* ed. by D. Tsichritzis, Technical Report CSRG-127 (March 1981).

V. Y. Lum *et al.,* Automating business procedures with form processing. *Research Report RJ3050,* IBM Research Laboratory, San Jose, California 95193 (March 1981).

G. M. Rader, C. D. Blewett and D. W. Shaklee, private communication (September 1980).

N. C. Shu *et al.,* Specification of forms processing and business procedures for office automation. *Research Report RJ3040,* IBM Research Laboratory, San Jose, California 95193 (March 1981).

D. Tsichritzis, A form manipulation system. In *A Panache of DBMS Ideas III* edited by F. H. Lochovsky. TR CSRG-101, pp. 53–71, Computer Science Research Group, University of Toronto, Canada (1979).

D. Tsichritzis, OFS: an integrated form management system. *Proceedings of the ACM International Conference on Very Large Data Bases* (1980).

M. Zisman, Representation, specification and automation of office procedures. *PhD Thesis,* Wharton School, University of Pennsylvania, Philadelphia, Pennsylvania (1977).

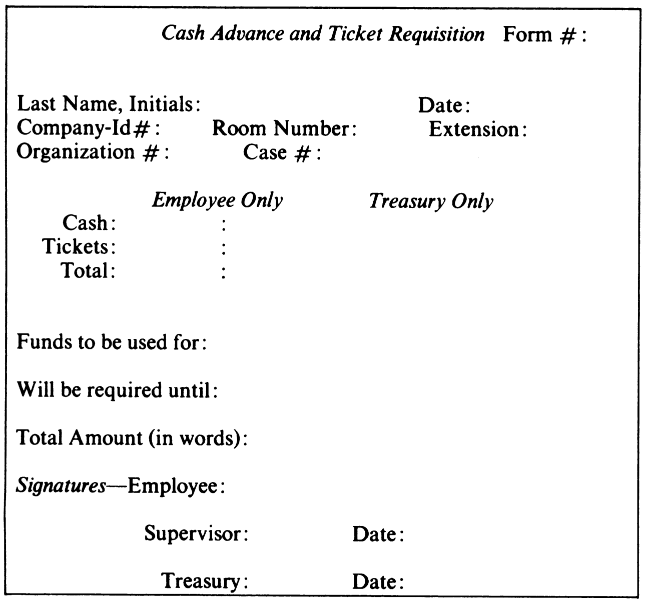
Received March 1982

APPENDIX

The *Cash Advance and Ticket Requisition* form (Fig. 4) is used by an employee to request an advance (at most $1500.00) for traveling on business. An employee requests some money to be given in cash and some money to cover the cost of an airline ticket. This advance will be vouchered later after the employee has completed the business travel.

The employee fills in the relevant fields in the form and mails it to the supervisor. The form is mailed to treasury by the supervisor after signing and dating it (assuming the supervisor approved the travel). The supervisor keeps a copy of the form.

Treasury checks the total on the form, making adjustments to the cash advance to reflect the actual cost of the ticket. The total as computed by the employee and the treasury must be the same.



Cash Advance and Ticket Requisition Form #:

Last Name, Initials: Date:
Company-Id#: Room Number: Extension:
Organization #: Case #:

	Employee Only	*Treasury Only*
Cash:	:	
Tickets:	:	
Total:	:	

Funds to be used for:

Will be required until:

Total Amount (in words):

Signatures—Employee:

Supervisor: Date:

Treasury: Date:

Figure 4

The form is defined in the form definition language as shown in Fig. 5.

```
form CashAdvance is
  imports

   with EMPLOYEE;
   use EMPLOYEE;

  display
            Cash Advance and Ticket Requisition Form #: $No

    Last Name, Initials: $Name                  Date: $Date_Emp
    Company-Id#: $Id   Room Number: $Room   Extension: $Ext
    Organization #: $Org         Case #: $Case

              Employee Only              Treasury Only
          Cash: $Cash                  : $Treas_Cash
       Tickets: $Ticket_Price          : $Treas_Ticket_Price
         Total: $Total                 : $Treas_Total

    Funds to be used for: $Purpose

    Will be required until: $Until

    Total Amount (in words): $Amount

    Signatures—Employee: $Emp_Sig

               Supervisor: $Sup_Sig        Date: $Date_Sup

               Treasury: $Treas_Sig        Date: $Date_Treas

fields
    $No: INTEGER virtual NEXT();
    $Name: STRING(1..30) required;
    $Date_Emp: $Date required;
    $Id: INTEGER range 0..99999 required post¦$Name = EMP_NAME($Id)¦;
    $Room: STRING(1..5) virtual EMP_ROOM($Id):
    $Ext: STRING(1..4) virtual EMP_EXT($Id);
    $Org: STRING(1..4) virtual EMP_ORG($Id);
    $Case: STRING(1..10) required;
    $Cash: FLOAT required post¦$Cash > = 0.0¦;
    $Treas_Cash: FLOAT required post¦$Treas_Cash > = 0.0¦;
    $Ticket_Price: FLOAT required post¦$Ticket_Price > = 0.0¦;
    $Treas_Ticket_Price: FLOAT required post¦$Treas_Ticket_Price > = 0.0¦;
    $Total: FLOAT virtual $Cash + $Ticket_Price post¦$Total < = 1500.0¦;
    $Treas_Total: FLOAT virtual $Treas_Cash + $Treas_Ticket_Price
                              post¦$Treas_Total = $Total¦;
    $Purpose: STRING(1..100) required;
    $Until: DATE required;
    $Amount: STRING(1..100) required;
    $Emp_Sig: SIGNATURE(1..6) required, lock all above
                 post¦VALID_SIGNATURE($Emp_Sig)¦;
    $Sup_Sig: STRING(1..100) required, after $Emp_Sig
                 post¦VALID_SIGNATURE($Sup_Sig)¦;
    $Date_Sup: DATE after $Sup_Sig;
    $Treas_Sig: STRING(1..100) required post¦VALID_SIGNATURE($Treas_Sig)¦;
    $Date_Treas: DATE after $Treas_Sig;
operations
  — — no additional form operations specified
access rights
  fields
    case DESIGNATION() is
      when TECHNICAL = > update all except $Treas_Cash, $Treas_Ticket_Price,
                                $Sig_Sup, $Date_Sup, $Treas_Sig, $Date_Treas;
      when SUPERVISOR = > update $Sig_Sup, $Date_Sup;
      when TREASURER = > update $Treas_Cash, $Treas_Ticket_Price,
                                $Sig_Treas, $Date_Treas;
    end case;
  operations
    case DESIGNATION() is
      when TECHNICAL | SUPERVISOR | TREASURER = >
                              all except destroy;
      when others = > display;
    end case;
local data and routines

   N: INTEGER := 0;
   function NEXT return INTEGER is
      N := N + 1;
      return N;
   end NEXT;
end CashAdvance;
```

Figure 5