V. Y. Lum
N. C. Shu
B. C. Housel

# A General Methodology for Data Conversion and Restructuring

**Abstract:** This paper presents a methodology and a model for data conversion or translation. The model assumes that both source and target systems are available and that conversion interfaces may be required to interact between these systems and the conversion system. To achieve data conversion or translation using this approach, two languages are needed: 1) a language to describe the data structures, and 2) a language to specify the mapping between source and target data. This paper describes these two languages, DEFINE and CONVERT and gives numerous examples to show the capabilities of these languages and how they can be used in data conversion and restructuring. Both languages are high level and nonprocedural and have the power to deal with most situations encountered in data conversion processes. In addition, the paper also describes some of the facilities in the languages specifically designed for data checking in a data conversion process.

## Introduction

In recent years applications of data base systems have grown very rapidly. While the use of data base systems relieves users of the task of having to know much of the implementation details, it has at the same time made data conversion a necessity because of various reasons. In general, data conversion is a complex problem requiring more of our attention than it has received in the past. This paper proposes a solution applicable to a broad class of logical data conversion problems.

Relatively little work has been done to find a solution making data conversion easier [1-11]. All investigations so far are preliminary. Only few individuals are actively involved. The most comprehensive work is done by members of the Stored Data Definition and Translation Task Group under CODASYL's System Committee, which attempts to develop a general method for defining data structures, storage structures, their relationship, and translation from one structure to another. Similar work goes on at the University of Michigan and to a lesser extent elsewhere (see references). The paper of Sibley and Taylor [11] gives a good account of some of these related works.

As reported in reference [12], the authors initiated a similar project at IBM. This project was established to investigate and develop a methodology for application conversion and migration. Application conversion is defined to include the movement of both data and programs from one system (or one form) to another. After studying the problem for some time, it became clear that current technology is inadequate in solving the general problem. Our initial attack is to solve first the problem of data conversion. This approach not only provides us with a more fundamental understanding of the problem but it actually is a necessary first step since we must understand what is needed for data conversion before we know what is to be done in the programs. Attention is paid, however, to the larger problem so that the results obtained can be used as a foundation in the solution of total application migration.

At present data conversion is done infrequently because of its complexity. In spite of changes in requirements, users are reluctant to change their data structures. It is believed that conversions will take place more frequently when better techniques are known, when automatic or semi-automatic aids are available, and when greater data independence is achieved.

## Problem environment

A study of current works revealed that current approaches to data conversion are either too broad and general, as in the case of CODASYL Task Group or Smith's and Taylor's work [5, 6], or too narrow in application as in Lin ahd Heller [13]. In the first case an economically feasible solution requires much more research and, therefore, appears distant. In the second case, a narrow approach is not really solving the main problem and, therefore, will provide benefits to only a small subset of computer users. The approach we have adopted is a compromise which will provide help to a broad class of users in the near future.

The approach assumes that the conversion system will run under the operating system of either the source or the target system. We also assume that an interface on the source system is available to transform the source data into an intermediate form acceptable to the translator, and an interface on the target system is available to take the output of the translator and transform it into the target data. In this way the translator is shielded from many of the physical incompatabilities of the source and target data such as parity schemes, etc. Specific details of our model are discussed later.

A basic assumption in our approach is that it is generally impossible to perform data conversion without the users' help. It is therefore visualized that it is the users' responsibility to describe the data structures for both the source and target data and to define the mappings between them. It is possible, however, to have an advanced system which may provide some prompting through interaction.

Two languages have been defined for this purpose: 1) DEFINE a language to define data structures, and 2) CONVERT, a language to specify mappings between source and target data, each of which may contain multiple logical record types and logical views. This paper discusses at some length these two languages. For a complete discussion, readers should refer to [14, 15].

In designing these languages we assumed that the users are skilled programmers. The programmers are familiar with their data's content, not in the sense of how many screws and nuts are in a parts' file, but in the sense of knowing that there exists a field for describing a part and that this field may contain blanks if no description exists. They know the semantics of their data and its structure at a logical level and what they want to be done in the mapping process. These aspects are quite different from the assumptions of the designers of data base systems who frequently consider their users to be casual users with little knowledge of the underlying data structure.

Assuming that the users are sophisticated and know their data, they do not know, however, the implementation details of their data structure, nor do they want to be burdened with the details of how to accomplish the whole conversion process. Another assumption is that the users are willing to follow some simple syntactic rules of the languages, but are unwilling to learn another complex language comparable to, say, COBOL or PL/1. We have also assumed that these users are not mathematically oriented and they do not appreciate semantics in mathematical terms. As a result we set out at the beginning to make our languages high level, nonprocedural, easy to learn, and simple to use.

The above aspects cannot be achieved without some expense. As opposed to a general language like PL/1, our languages are simple only because we tailored them to a specific purpose, namely, data conversion and in certain cases we traded capabilities for simplicity. Our philosophy is to provide a language to handle a great majority of the cases encountered frequently in data conversion and let the remaining small number of cases be handled by the computer's procedural languages. In any case, the languages have been so structured that additional capabilities can be included without much difficulty.

## The conversion model

Figure 1 illustrates the overall conversion process in our model. The source systems which originally process the source data is used to access it and interacts with the conversion interface module to produce a nearly system independent source data called linearized source files. As the name implies, linearized files are sequential files. (More is said about them in a subsequent section). These files become the input to the converter/translator. The output from the converter/translator is another set of linearized files called linearized target files, which are changed into physical target files with the use of the conversion interface and the target system.

Generally speaking, data conversion can be divided into two basic categories: 1. from files to data base, and 2. from data base to data base. These two categories have some basic differences. Several points are salient in the first case. 1) Data is generally not well organized. It contains much redundancy and much of the data description is carried implicitly in the procedures. In fact, frequently additional information is contained there. For example, a census file may be separated into two parts such that the first part contains information about males and the second part about females, but this separation is not stated explicitly when the data structure for this file is defined. In our system all this descriptive information is made explicit. 2) These source files are sequential files. Since the real world at this time has a preponderance of sequential files to be converted to data bases, we have attempted to define in our data definition language a capability that can describe most of these files instead of imposing severe limitations on the formats of linearized files. 3) The COBOL files deserve further attention because a great majority of commercial users are COBOL oriented. Hence, our data definition language has been designed to have a strong COBOL flavor and the capability to describe the common COBOL files. Thus, we define a linearized file to be a file belonging to that subset of sequential files describable by our data definition language. It may have a flat or hierarchical record structure. It may contain self-defining data, terminators of different kinds, multiple record types within the same

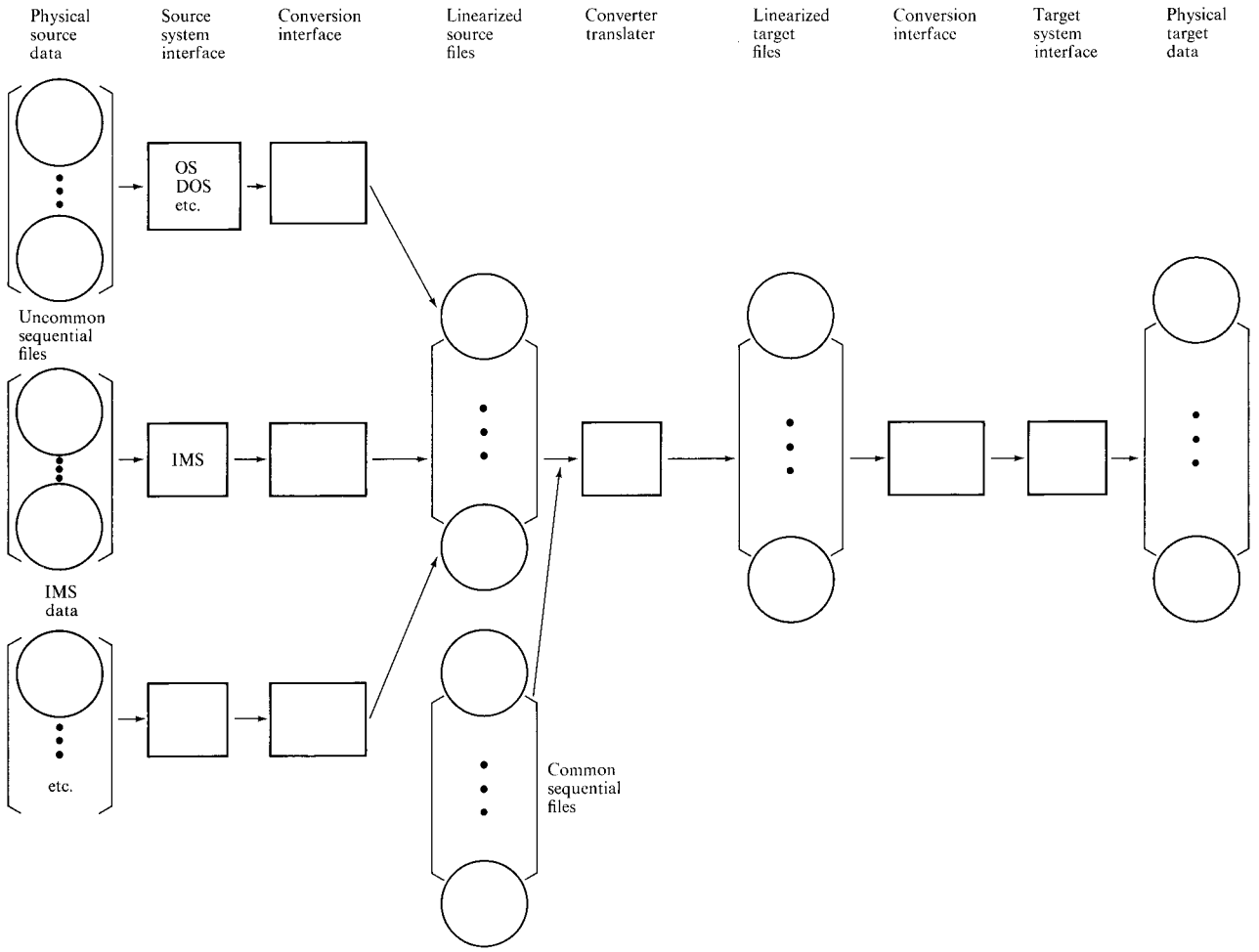



**Figure 1** Overall view of conversion.

file, and repeating or non-repeating groups belonging to the same logical record but appearing in separate physical records. It does not, however, contain any system-dependent accessing or alignment information. Direct addressing, if present, must be replaced by symbolic addressing. In this manner, common sequential files can be directly used as input to our conversion model.

The second category of data base to data base conversion is different from that of file to data base conversion. Here, source data is much better defined and its structure can be quite complicated. In this case we expect the users to use the source systems' support, including utilities, to decompose the data base into linearized files, stripping all the unneeded control information and replacing physical or direct pointers by symbolic or key pointers (the process of linearization can be automated by creating conversion interfaces for individual data base systems). For a given data structure, the linearized files can be expected to be different depending on the user. For example, consider the case as defined in Fig. 2a. One can create the linearized files for this data base as given in Fig. 2b where each physical data base becomes one linearized file. Alternatively one may want to create a set of linearized files as given in Fig. 2c. It appears that there are an arbitrary number of ways to define linearized files. In reality, however, the limitations existing in a system and its support and the naturalness of the resulting files dictate the choice. This choice has only a very small impact to the conversion process because using the translation language, one can alter this choice easily.

In a similar manner the target structure is defined in terms of linearized files. These files should be defined in such a way that easy loading is possible. As in the case of source data, a given data structure in a target system may have several versions of linearized target files.

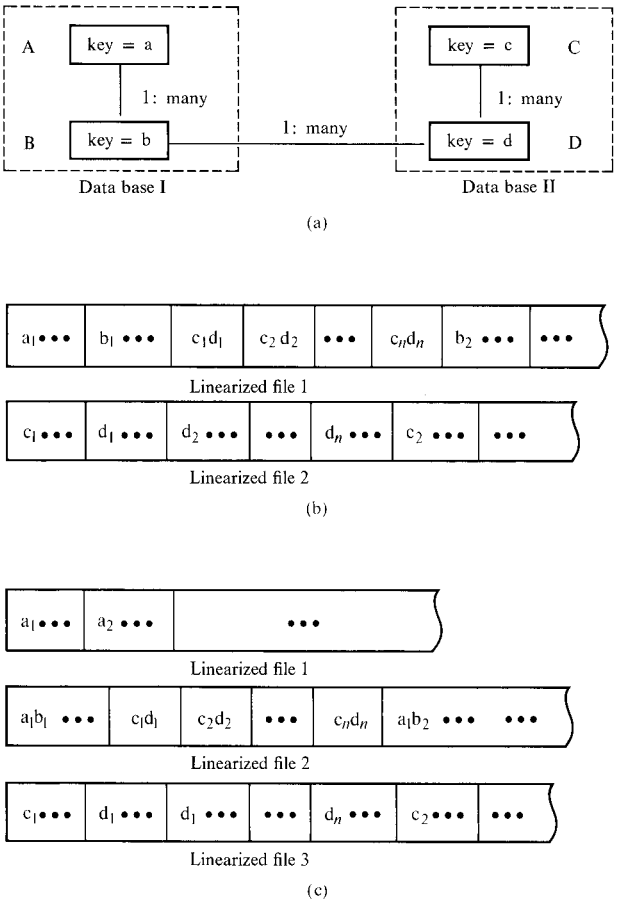



**Figure 2** Linearization of a Data Base.

Let us expand the central portion of the conversion process to present more details as shown in Fig. 3. The data definition analyst writes the description of the linearized source and target files. The translation system Reader extracts from this description the logical view of these files and strips off much of the encoding information in the source and target file description which is not useful for restructuring. Examples for such information include the length of the fields, the kind of data representation, and the way a repeating group may be terminated. The data translation analyst, who may or may not be the same person as the data definition analyst, then writes data mapping specifications to indicate the movement of data from source to target. During actual conversion, the linearized source files are changed into a standard system internal form, i.e. IF(S1) and IF(S2)$\cdots$, suitable for reorganization. Data in this form will go through restructuring as specified by the mapping statements. The end result from the restructurer component is data in a standard form i.e., IF(T1), IF(T2), $\cdots$, closely corresponding to the target logical view. The writer then changes this data format into linearized target data.

## Data description language—DEFINE

Having described the model for data translation (Fig. 3), we see that a data description language is needed for two purposes. The first is to provide means for describing a wide spectrum of hierarchically structured linear files to enable them to be converted by the Reader (Writer) to (from) the conversion system's internal form. The second is to provide the basis for extracting logical views of the files suitable for use in the restructuring process. At this time it is appropriate to discuss briefly why we felt the introduction of a new data description language was necessary.

First, we examined the data definition facilities of COBOL and PL/1, since they are well-known and broadly used by the computing community. We found they lacked sufficient capability for describing files with characteristics such as variable length fields, optional data, and self-defining data. For example, in COBOL files, frequently one file contains multiple record types distinguishable by a prefix. The procedural portion of the program tests the prefix and using the REDEFINE feature applies the appropriate data definition to the remainder of the record depending on the prefix value. It is felt that these kinds of semantics should be included in the data description itself where possible.

Second, we investigated other data definition and translation languages which were thought to be potential candidates for our purposes. Specifically, we studied the Stored Data Definition Language (SDDL) developed at the Univeristy of Michigan [11], and the Data Definition Language and Data Manipulation Language (DDL/DML) developed for data conversion at the University of Pennsylvania [7, 16]. The Michigan SDDL is very general. In contrast, the Pennsylvania language is defined for one file to one file conversion only, but relies on the use of PL/1 procedures. Both languages include facilities for describing not only data structures, but also their corresponding storage structures, and the mappings between them. Further, the Michigan language attempts to treat every level of description from device-media and system specific storage and data structures to the high-level data structure classes and schemas. While the above capabilities are required to attain the goal of complete data description at all levels, they are not required for our needs.

The data description language, called DEFINE, is developed specifically for describing the linearized source and target data structures for our conversion system which, as mentioned before, uses this information to

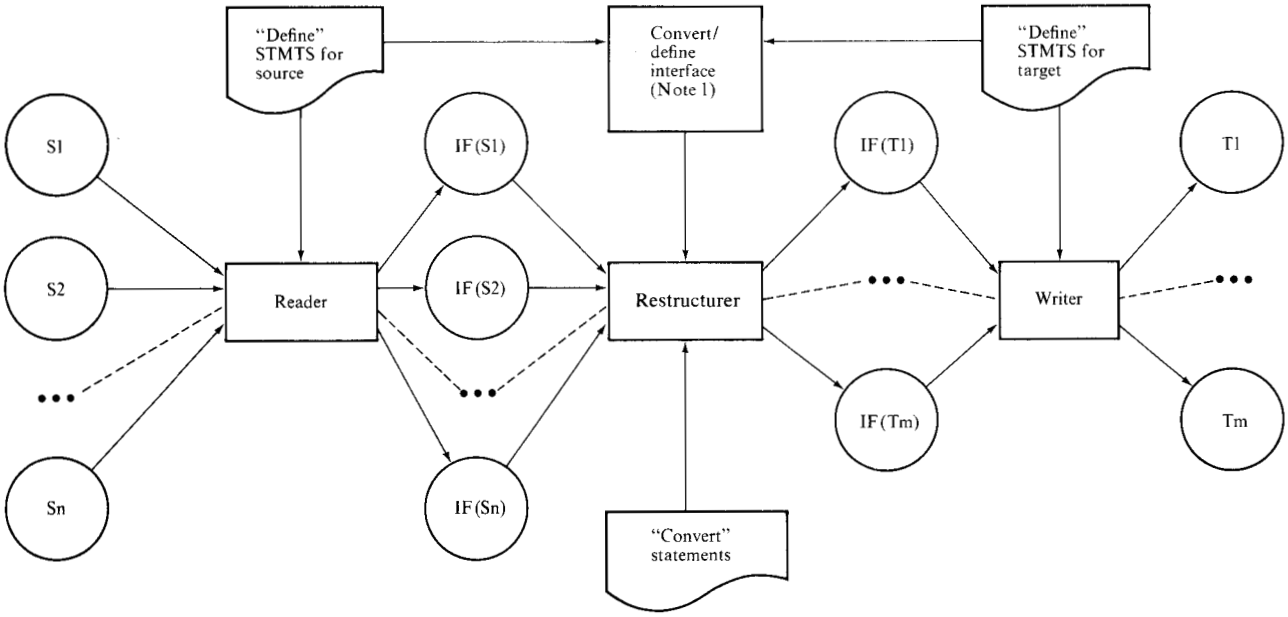



**Figure 3** Model of translation.
Note: Logical views of the files are extracted from the define specifications.

parse data in the read and write steps of the translation process and to extract the logical views to be presented to the translation analyst.

As in all data description languages, DEFINE provides constructs for describing the usual encoding characteristics of data. However, the unique feature of the language lies in its rich facilities for explicitly describing the characteristics of files generated by the REDEFINE feature that is used very frequently in COBOL. Figure 4 illustrates some of these common record formats where REDEFINE may have been used. A complete DEFINE program is a structured description of one or more source and target files as outlined in Figure 5.

The data description block (A block is a general term used for designating a group of statements.) (DDB) applies to the entire DEFINE program. A DDB may contain one or more *file description blocks* (FDB). The keywords SOURCE and TARGET designate whether the file is to be parsed (input to translator) or generated (output by translator), respectively. The "declarations section" for the DDB specifies characteristics (e.g., character codes) which are true for all subsequent FDBs except those which are specifically overridden in the declaration section of FDBs. Similarly, an FDB declaration, for example, specifying character string justification, applies to all the relevant data entities in the given file unless it is overridden in a particular data specification. If the declarations section is absent in the DDB or the FDB, installation specified defaults are assumed.

Following the declarations section in the file description block are descriptions which specify the schema and encoding characteristics of the file data structures in the order of their occurrence. There are two generic data constructs, groups and items. An item is a named unit of elementary data and a group is an ordered (named or unnamed) sequence of items or groups. The term data object is used to denote either one. Multilevel hierarchical data structures can be defined by recursive definition of groups. Figure 6 illustrates a skeletal structure of a file description.

As shown in the diagram the description of a file is made up of group and item descriptors (GD and ID respectively). Repeating groups can have a variable number of instances as indicated by the "occurs" clause.

All descriptors can be either unconditional or conditional. The example in Figure 6 illustrates the unconditional descriptors. A conditional descriptor is specified like an unconditional one except that a conditional expression is specified following the data object name. If the conditional expression is evaluated as true, the given descriptor will be used to parse the source data or generate target data. If it is evaluated as false, the next descriptor will be used.

In some cases, the structure of a record depends on the content of a particular field. COBOL files with multiple record types are examples of this category. DEFINE has a GROUPCASE descriptor to provide it with the capability of dynamically selecting the appro-

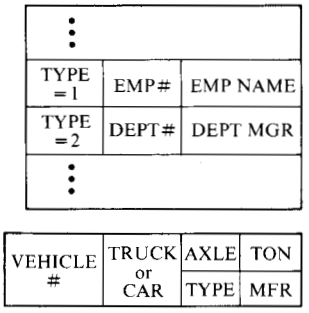
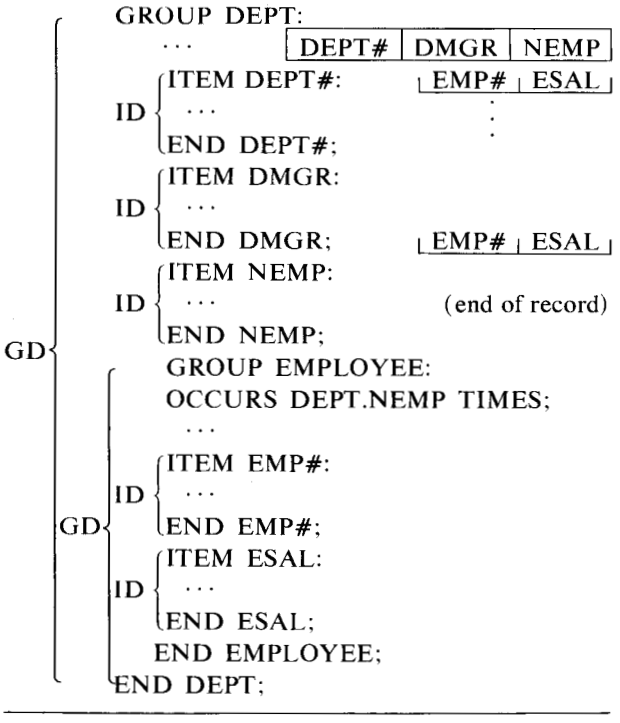



**Figure 4** Examples of self-describing common file formats.

```
DATA DESCRIPTION:
  [optional data description block (DDB) declarations]
  SOURCE FILE DESCRIPTION( filea ):
    [optional file description block (FDB) declarations]
    data descriptors for "filea"
  END FILE DESCRIPTION;
  TARGET FILE DESCRIPTION ( fileb ):
    [optional FDB declarations]
    data descriptors for "fileb"
  END FILE DESCRIPTION;
END DATA DESCRIPTION;
```

**Figure 5** DEFINE program structure.

**Figure 6** Skeletal DEFINE structure for DEPT file.




priate descriptor. The example in Figure 7 illustrates the use of the GROUPCASE descriptor. In this example, each DEPT record is followed by a number of EMPLOYEE and/or PROJECT records. If the first field, TYPE, in a record contains E, the record is an EMPLOYEE record; if it is P, the record is a project record. Otherwise, it is a DEPT record. These conditions are reflected in the specification of the conditional descriptors, EMPLOYEE and PROJECT. The GROUPCASE descriptor has no effect if none of the conditions in its member descriptors is satisfied.

Frequently, arithmetic and conditional expressions are required in writing a data description. In general, a conditional expression is a logical factor or a sequence of logical factors separated by AND's and OR's. A logical factor is a predicate, comparison of arithmetic expressions, or a parenthesized conditional expression and the operators allowed in the arithmetic expressions in DEFINE are +, −, * and /. For example A < B and C * D = E are the two logical factors in (A < B OR C * D = E). Conditional expressions are primarily used in conditional descriptors.

Currently only one predicate has been defined in the DEFINE language: CONFORM(x). This predicate returns true if the referenced yet untranslated data object agrees with its description, and false otherwise. For example, suppose that a record is to be interpreted differently depending on the last character's content. If the last character is numeric, the record takes one form; if it is alphabetic, the record takes another form. The description for this might be as follows:

```
GROUPCASE GF:
  GROUP G(CONFORM(LAST)):
    :
    ITEM LAST:
      CHAR(PICTURE IS '9');;
  END;
  GROUP F(CONFORM(LAST)):
    :
    ITEM LAST:
      CHAR(PICTURE IS 'A');;
  END;
END;
```

Note that in this example, the conditional descriptor approach from Figure 7 is not applicable. The difference between these two examples by our convention is that in Fig. 7, the value for testing is that one which has just been parsed; while in this example, the value to be used for testing has yet to be defined. Thus, the predicate CONFORM essentially provides us with the capability of looking ahead in processing the input data.

In the above example the specification of data representation by means of the "picture" clause, which is a generalization of COBOL's "picture" clause was introduced. The details for this appear in reference [14].

This completes the essential description of the DEFINE language. Following is an example of a DEFINE description.

Consider a parts-supplier (PTS) file with variable length records whose fields are P#, DES, S#, CN, and UC, standing for part number, description, supplier number, company name, and unit cost, respectively. The last three fields are grouped together to form a supplier repeating group. Further, each of the fields is fixed length with the following structure:

P# – character string of length 4 with first two characters alphabetic and the next two numeric. Key for record.
DES – 5 alphabetic characters, left justified, padded with blanks.
S# – 4 numeric, packed-decimal characters. Key for repeating group.
CN – 10 alphanumeric characters
UC – 15 bits binary integer

In addition, assume that the file is so structured that each logical record (Fig. 8(a)) ends with a $ sign. A DEFINE description for this file is given in Fig. 8.

The example in Fig. 8 illustrates the general flavor of DEFINE and reveals some of its typical constructs, including some shorthand descriptions. For example, instead of ending items by 'END item-name;', we have used ';' as a substitute. This substitution is possible in other places as well (e.g., in 'END group-name;'). The description is self-explanatory.

```
GROUP DEPT-EMP-PROJ:
 ...
 GROUP DEPT:
   FOLLOWED BY EOR; /* end or record */
   ITEM DEPT#: ...;
   ITEM DMGR: ...;
   ITEM NEMP: ...;
 END DEPT;
 GROUP EMP-PROJ-RCD:
   OCCURS FROM 0 TIMES, TERMINATES
     WHEN TYPE NOT IN ('E', 'P');
   ITEM TYPE: CHAR(1); END TYPE;
   GROUPCASE PE:
     GROUP EMPLOYEE(TYPE = 'E'):
       ...
       description for EMPLOYEE
       ...
     END EMPLOYEE;
     GROUP PROJECT(TYPE = 'P'):
       ...
       description for PROJECT
       ...
     END PROJECT;
   END PE;
 END EMP-PROJ-RCD;
END DEPT-EMP-PROJ;
```

**Figure 7** "GROUPCASE" Example.

### Conceptual data representation

To understand further the translation process one must understand the representation of the data as viewed by the translation analysts and the conversion system's Data Restructurer. It is paramount to define a representation or form that is simple, capable of representing different data structures and familiar to data conversion analysts. While there are many candidates for this role, most of them can be eliminated because they are too restrictive or require too much learning. Our final choice was the partially filled tabular form for hierarchically structured data. All data structures will be transformed into this tabular form, which from now on will be referred to simply as the Form. This choice biases toward the source and target data organized in flat files, hierarchical, and relational structures.

To illustrate the tabular form, or simply the Form in Fig. 9 (a) the schema of a hierarchical data structure is represented. The actual data may be organized as COBOL records, in which case the information on education, skill, or child may be repeating groups, or the data may be organized as a tree as in some data base systems (e.g., IMS). In either case, the data can be represented by the Form as shown in Fig. 9 (b). Note that in this representation, caution is required not to produce false information that does not exist in the original. For example, although information on education and child exist in the same row in the table, there does not exist any relationship between them except that they are both related to the same person. Hence these tables or Forms are not relational tables [17–21], but perhaps somewhat akin to Bracchi's unnormalized relations [22–24]. Note that additional visual aids can be included in the Form's layout if so desired by the user, e.g. double lines to separate repeating groups.

The use of the Forms to represent data and its structure is believed to possess many advantages. Included among these are that the Forms are easy to visualize, they can represent other data structures readily and they are easy to manipulate. One can define a very simple and powerful translation language to operate on these Forms as shown in a subsequent section.

```
DATA DESCRIPTION:
  DECLARATIONS:
    CHAR CODE IS EBCDIC;
    NUMERIC ENCODING IS IBM 370;
    CONSTANT BLANK IS HEX '40';
  END DECLARATIONS;
  SOURCE FILE DESCRIPTION (PARTS-SUPPLIER):
    GROUP PTS:
      OCCURS FROM 1 TIMES,
        FOLLOWED BY END-OF-FILE;
      KEY IS P# WITHIN PTS;
      ITEM P#:
        CHAR (PICTURE IS 'AA99');;
      ITEM DES:
        CHAR (PICTURE IS 'A(5)',
          JUST IS LEFT
          AND PAD CHAR IS BLANK);;
      GROUP S:
        OCCURS FROM 1 TIMES,
          FOLLOWED BY '$';
        KEY IS S# WITHIN S;
        ITEM S#:
          DEC (PICTURE IS '9(4)');;
        ITEM CN:
          CHAR (10);;
        ITEM UC:
          BINARY (15);;
      END S;
    END PTS;
  END FILE DESCRIPTION;
END DATA DESCRIPTION;
```

**Figure 8** DEFINE description of PTS file.

### Translation definition language—CONVERT

From the logical views extracted from the DEFINE description of data structures, a translation analyst can visualize his data in view of the Form (Fig. 9) and proceed to write mapping statements for manipulating his data into the format required in the target description. In order to achieve this goal, a translation language must have the following capabilities:

1. Combining components of different linearized files to form new files according to some specified criteria.
2. Decomposing a linearized file to form different files according to some specified criteria.
3. Rearranging the order of data instances (occurrences) within a linearized file in some manner.
4. Altering values of data instances in different manners. Sometimes the new values may be derived arithmetically from the old values and other times new values may be arbitrarily defined from the old ones. An example of the former is the changing of payroll from hourly pay to weekly pay. An example of the latter is that the field 'sex', previously containing 'male' or 'female', will now become '1' or '0'.
5. Formation of trees, or decomposition of trees.

These are but some of the minimum capabilities required from a translation definition language.

Several existing languages [7, 19, 22, 25, 26] were investigated to determine if they may be used for data translation. It was found that they are either too low level, lack the required capability, or are too mathematically oriented for the type of users we visualized. As a result, the translation definition language, CONVERT, based on a few simple concepts was developed. The language is algebraic, (i.e. operators & operands), specifically oriented to operate on Forms, but has powerful and flexible restructuring capabilities.

The data mapping and restructuring facilities in CONVERT are provided by a set of Form operators. Figure 10 represents a list of the form operations and their formats as currently defined. The list includes component extraction, SELECT, SLICE, GRAFT, CONCAT, MERGE, SORT, ELIM-DUP, CONSOLIDATE, a set of built-in-functions (SUM, MAX, MIN, AVG and COUNT), assignment and CASE-assignment. The meanings and uses of some of the more interesting Form operations are discussed later.

Each of these Form operators operates on one or more Forms (or components of them) and produces a Form as a result. The resultant Form can then be used as an operand for another Form operation. Except for assignment and CASE-assignment all Form operations can be nested, and all have the same general format: Operator (Operands [options] [:Specified conditions]). In this statement the square brackets, [ ], are meta symbols denoting that the enclosed is optional.

In describing the operands, we use the following notations: "F" denotes a Form which could be either a Form name or the result of a Form operation. "f" denotes a field (i.e., a column in a Form correspondihg to an "item" in the DEFINE language). "c" denotes a component of a Form, which could be either a field or a sub-Form. "EXPR" denotes an arithmetic expression derivable from the fields of a Form. To be more specific, EXPR could be any of the following: 1) a constant; 2) a field name; 3) a built-in-function (e.g., SUM, MAX, MIN, AVG, COUNT); 4) an expression derivable from 1, 2, or 3 above, or recursively a derived expression enclosed in parenthesis, using +, −, *, / as arithmetic operators; 5) a sub-Form or group name. (Note that a sub-Form is not allowed to be an operand in arithmetic operation.) As a rule, the order of appearance of the components or EXPR in the statement determines the component order in the resulting Form.

In general, the *specified conditions*, i.e., "SC", can be expressed as a SC-EXPR, which is defined as logical factors connected by AND(s) and/or OR(s) and is not restricted to fields in one Form. A logical factor can be either 1) an EXPR compared with another EXPR, 2) an EXPR compared with ANY OF a one-column Form, or 3) an EXPR compared with ANY OF a list of single values. The permissible comparison operators include, =, ¬=, >, <, ¬>, ¬<, > = and < =. A logical factor is assigned a value of "true" or "false" according to the result obtained from evaluating the comparison. The evaluated logical factors are ANDed or ORed together as specified to determine the final "true" or "false" value. An operation is executed only if the SC-EXPR yields a "true" result.

In this paper only the operations SELECT, GRAFT, built-in-functions, and CASE-assignment are discussed. (For other operations, refer to [15].) Following are some simple examples to illustrate these operations.

In the examples, PTS is a parts-supplier file described in Fig. 8 but repeated here as a Form. INV is an inventory file containing P# (i.e., part number) and QH (i.e., quantity-on-hand). SUP is a supplier file having CN (i.e., company name), S# (i.e., supplier number) and CA (company address).

PTS

| P# | DES | S | | |
|---|---|---|---|---|
| | | S# | CN | UC |
| 2 | X | 4<br>2 | AB<br>BB | 5<br>4 |
| 3 | XX | 4<br>1 | AB<br>XB | 2<br>3 |
| 7 | Y | 7 | C | 7 |

INV

| P# | QH |
|---|---|
| 2 | 10 |
| 3 | 17 |
| 4 | 5 |
| 7 | 20 |

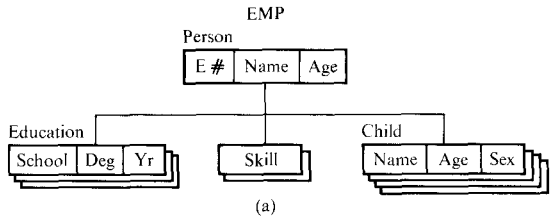
SUP

| CN | S# | CA |
|---|---|---|
| AB | 4 | SJ |
| BB | 2 | MV |
| C | 7 | SF |
| D | 3 | LA |
| XB | 1 | SJ |

1. SELECT ([$EXPR_1$, ··· $EXPR_n$] FROM $F$ [, ···] [:$SC$]

This operation selects part(s) of a Form if specified conditions are satisfied. For example, to create a new file consisting of part numbers of the parts supplied by suppliers located in San Jose, together with the corresponding part descriptions and their suppliers code numbers and names, one can use the SELECT operation as follows:

SELECT (P#, DES, S#, CN FROM PTS : PTS.S# = SUP.S# AND SUP.CA = 'SJ');



(a)

EMP

| Person | | | Education | | | Skill | Child | | |
|---|---|---|---|---|---|---|---|---|---|
| E # | Name | Age | School | Deg | Yr | | Name | Age | Sex |

(b)

**Figure 9** Translation analyst's views of Data in terms of 'Forms.' (a) Schema of hierarchical data (b) Form representation of hierarchical data.

**Figure 10** List of Form operations.
(Note) [ ] denotes enclosed are optional
{ } denotes one of the enclosed must exist
Underlined are reserved words
":" may be substituted by "*WHERE*"

1. Component extraction F($C_1$, $C_2$, ··· $C_n$)
2. Assignment
3. SELECT ([EXPR, ··· EXPR] FROM F [, ···] [:SC])
4. SLICE ($f_i$ ··· $f_j$ FROM F)
5. GRAFT ($F_1$, $F_2$, ··· ONTO $F_n$[AT F][:SC])
6. CONCAT ($F_1$, $F_2$, ··· ONTO $F_n$ AT f)
7. MERGE ($F_1$, $F_2$, ··· $F_n$)
8. SORT (F [BY [Ascending / Descending] $f_1$, $f_2$, ···][WITHIN PARENT])
9. ELIM_DUP (F)
10. CONSOLIDATE (F FOR UNIQUE {$f_1$, $f_2$, ··· / ($f_1$, $f_2$, ···), ($f_a$, $f_b$, ···)})
11. Built-in-Functions
    {SUM / MAX / MIN / AVG / COUNT} (f IN F [FOR UNIQUE $f_1$, $f_2$, ···] [:SC])
12. CASE Assignment

In this case, PTS is the source file from which a target Form consisting of P#, DES, S# and CN are to be constructed. Not all instances, however, in the source file produce an image in the target because we are interested in only those instances where the suppliers are in San Jose. Since the information about the location of a supplier appears only in the SUP file, one must find the connection between the PTS Form and the SUP Form through the use of some common information which in this case is S#. The resulting form is as follows:

| P# | DES | S# | CN |
|---|---|---|---|
| 2 | X | 4 | AB |
| 3 | XX | 4 | AB |
| | | 1 | XB |

In addition, the SELECT operation also provides a facility to derive new data. As mentioned earlier, computations can be performed on selected fields. To increase the unit costs of the parts in PTS by 25%, one may use the following statement:

NEWPTS(P#, DES, S#, CN, UC) ← SELECT (P#, DES, S#, CN, UC*1.25 FROM PTS);

2. GRAFT ($F_1, F_2, \cdots$ ONTO $F_n$ [$AT f$][:$SC$])

GRAFT combines two or more Forms into one Form when specified conditions are met. It produces Cartesian Products when ":SC" is omitted.

In general, the conditions to be satisfied can be stated as an SC-EXPR as described before. Since GRAFT operates on two or more Forms, it should be apparent that the SC-EXPR should include at least the logical factors which serve to tie the Forms together. For example, to form one file from the PTS and INV files such that the resulting file will have the information of the PTS file plus the quantity-on-hand (i.e., QH) obtained from INV, one can use the statement GRAFT (INV ONTO PTS: PTS.P# = INV.P#); here, the SC-EXPR serves as a tie between the PTS and the INV files. There are two tying fields: P# of PTS and P# of INV. Only the one in the Form after "ONTO" will appear in the resulting Form. The result is:

| | | S | | | |
|---|---|---|---|---|---|
| P# | DES | S# | CN | UC | QH |
| 2 | X | 4 | AB | 5 | 10 |
| | | 2 | BB | 4 | |
| 3 | XX | 4 | AB | 2 | 17 |
| | | 1 | XB | 3 | |
| 7 | Y | 7 | C | 7 | 20 |

This way of stating conditions is useful in most of the cases. There are situations, however, where some of the data exists only in some of the files. For example, P# = 4 exists in INV but not in PTS. By stating "PTS.P# = INV.P#" as the satisfying condition, the inclusion of P# = 4 in the new file is excluded. To include all P#s in the new file, leaving the missing information blank the "*PREVAIL*" clause is used to specify the conditions.

The PREVAIL clause, in general, takes the following format:

$f_1, f_2, \cdots$ PREVAIL $[f_j, f_k, \cdots f_n]$

where $f_1, \cdots f_n$ are the names of the fields whose values are to be "matched". The names on the left hand side of the key word "PREVAIL" are considered to be the *prevailing* fields. The union of the instances of the prevailing fields determine the instances to be included in the resulting Form. Take the PTS and INV files for example. The statement GRAFT (INV, ONTO PTS: INV.P# PREVAIL PTS.P#); produces the following:

| | | S | | | |
|---|---|---|---|---|---|
| P# | DES | S# | CN | UC | QH |
| 2 | X | 4 | AB | 5 | 10 |
| | | 2 | BB | 4 | |
| 3 | XX | 4 | AB | 2 | 17 |
| | | 1 | XB | 3 | |
| 4 | – | – | – | – | 5 |
| 7 | Y | 7 | C | 7 | 20 |

*3. Built-in-functions*

{SUM, MAX, MIN, AVG, COUNT} ($f$ IN $F$ [FOR UNIQUE $f_1, \cdots f_n$][:$SC$])

The built-in-function computes, respectively, the sum, maximum, minimum, average or count of the instances of a certain field of f in a Form F where the specified conditions are satisfied. All built-in functions have the same format and operate in exactly the same manner. If the "FOR UNIQUE $f_1, \cdots f_n$" option is taken, computation performed over instances of f for unique values of $f_1, \cdots f_n$ where $f_1, \cdots f_n$ must be ancestors (an ancestor is any generation of a parent in a hierarchical path) of f. If there is no "FOR UNIQUE" clause stated, computation is performed over all instances of f in the Form. For example,

SUM(UC IN PTS) results 5 + 4 + 2 + 3 + 7

T(A,B) ← SELECT(P#,COUNT(S# in PTS FOR UNIQUE P#) FROM PTS) results the following:

T

| A | B |
|---|---|
| 2 | 2 |
| 3 | 2 |
| 7 | 1 |

*4. CASE Assignment*

CASE Assignment allows varied operations to be performed over different instances. These varied operations must produce homogeneous results to be assigned to the resulting Form. The variation is dependent on some prescribed tests either 1) on the value of a single instance of a field, or 2) on a set of values of a specific field for unique parent or ancestors. Accordingly, there are two formats for the CASE assignment. Here only the first format is discussed. (Refer to [15] for second format.)

$$F \leftarrow \underline{\text{CASE}}\ (f\ COP\ v_1, v_2, \cdots v_n[,\underline{\text{OTHERS}}])\ (F_1, F_2, \cdots F_n[,F_{n+1}]);$$

With this format, assignment is allowed to be varied according to the value of an occurrence of the specified field, f. For each instance of f, its value is compared with $v_i$ (where $1 \leq i \leq n$) in the left to right order until a "true" result is obtained from the evaluation. At that time, the corresponding $F_i$ will be activated to provide the source for assignment. $F_i$ could be any of the Form operations (except the assignment and CASE assignment). The scope of these operations, however, is limited to the pertinent instance (not all instances of the Form). For this reason the italic $F$ (instead of F) is used to denote the Form operations effective for CASE assignment. Furthermore, $v_i$ and $F_i$ must be paired. If the optional pair of [,OTHERS] and $[,F_{n+1}]$ is not specified, no operation will be performed when all tests specified in f cop $v_i, \cdots v_n$ failed.

To change the entries of "FEMALE" and "MALE" in the SEX field of the source Form SF into SEX CODE where "0" represents female and "1" represents male, the following statement can be used:

```
T (E#, SEX_CODE) ← CASE (SF.SEX = 'MALE', 'FEMALE')
    (SELECT (E#, '1' FROM SF), SELECT (E#, '0' FROM SF));
```

Source: SF

| E# | SEX |
|---|---|
| 51 | MALE |
| 57 | FEMALE |
| 74 | MALE |
| 78 | FEMALE |

Target: T

| E# | SEX-CODE |
|---|---|
| 51 | 1 |
| 57 | 0 |
| 74 | 1 |
| 78 | 0 |

A description, in some detail, of the syntax and semantics of a few of the CONVERT operators was presented. This is believed to be sufficient to illustrate the flavor of the CONVERT language. A detailed description of each of the operators can be found in reference [15].

## Application example

As mentioned earlier other data structures can be expressed in linearized files, and hence the Forms. In this section we discuss how this can be done. In addition, an example that illustrates how the methodology can be used to specify data translation in a network environment is also given.

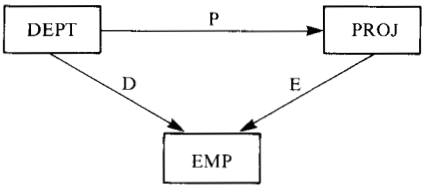



**Figure 11** A network example.

Consider the simple example of a network (Fig. 11), with nodes DEPT, PROJ, and EMP and edges D, P and E where D links all employees belonging to the same department, P links all the projects in a department, and E links all the employees working on the same project.

Our approach in dealing with networks is to decompose the network into a family of hierarchies each of which can be represented as a Form. This decomposition, however, is not unique for a given network. It depends on what is natural or convenient to the user. For instance, one way to decompose the above network is to have each node corresponding to a Form as in Fig. 12(a) where the connecting information, designated by each named edge, is embedded in the Forms. Alternatively, each named edge can be in itself represented as a Form as in Fig. 12(b). A combination of both techniques is equally applicable. The important point to note is that the connecting information is represented by symbolic pointers in terms of the keys. In the case where the connecting information is at a level away from the root of a tree, a concatenation of keys may be necessary. For example, suppose that an edge exists in the above network from PROJ directly to SKILL in EMP, then this symbolic pointer will be expressed by a concatenated key of E#.SKILL.

To illustrate the application of the CONVERT language, suppose that the above network is decomposed into a representation as shown in Fig. 12(a), and is reorganized such that it generates one file with the structure shown in Fig. 13 and Fig. 14.

One translation specification for this example may be stated as follows:

```
DEMP ← SELECT (D# FROM DEPT,
                E#, NAME, EDUCATION, SKILL FROM EMP:
                EMP.DEG¬ = 0 AND DEPT.E# = EMP.E#);

NEMP ← SELECT (D# FROM DEPT,
                E#, NAME, SKILL FROM EMP:
                EMP.DEG = 0 AND DEPT.E# = EMP.E#);

TDEPT ← GRAFT   (DEMP, NEMP ONTO DEPT:
                 DEPT.D# PREVAIL DEMP.D#, NEMP.D#);

DEPTDB ← GRAFT (PROJ ONTO TDEPT AT P#:
                DEPT.P# = PROJ.P#);
```

DEPT

| D# | MGR | P# | E# |
|---|---|---|---|
| 55 | SMITH | P1<br>P2<br>P3 | 541<br>551<br>552<br>553<br>554<br>555<br>556 |
| 54 | JONES | P4 | 542<br>543<br>544 |
| ⋮ | ⋮ | ⋮ | ⋮ |

EMP

| E# | NAME | EDUCATION | | SKILL |
|---|---|---|---|---|
| | | DEG | YR | |
| – | – | –<br>– | –<br>– | –<br>– |
| – | – | – | – | – |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

PROJ

| P# | LEADER | BUDGET | E# |
|---|---|---|---|
| P1 | 541 | 100K | 551<br>552<br>541 |
| P2 | 554 | 200K | 553<br>554<br>555 |
| P3 | – | 50K | 555<br>556 |
| P4 | 542 | 300K | 542<br>543<br>544 |
| ⋮ | ⋮ | ⋮ | ⋮ |

(a)

DEPT

| D# | MGR |
|---|---|
| 55 | SMITH |
| 54 | JONES |
| ⋮ | ⋮ |

P

| D# | P# |
|---|---|
| 55 | P1<br>P2<br>P3 |
| 54 | P4 |
| ⋮ | ⋮ |

D

| D# | E# |
|---|---|
| 55 | 541<br>551<br>552<br>553<br>554<br>555<br>556 |
| 54 | 542<br>543<br>544 |
| ⋮ | ⋮ |

PROJ

| P# | LEADER | BUDGET |
|---|---|---|
| P1 | 541 | 100K |
| P2 | 554 | 200K |
| P3 | – | 50K |
| P4 | 542 | 300K |
| ⋮ | ⋮ | ⋮ |

E

| P# | E# |
|---|---|
| P1 | 551<br>552<br>541 |
| P2 | 553<br>554<br>555 |
| P3 | 555<br>556 |
| P4 | 542<br>543<br>544 |
| ⋮ | ⋮ |

EMP

| E# | NAME | EDUCATION | | SKILL |
|---|---|---|---|---|
| | | DEG | YR | |
| – | – | –<br>– | –<br>– | – |
| – | – | – | – | – |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

(b)

**Figure 12** A node and its edges (a) represented as one Form (b) represented as separate Forms.

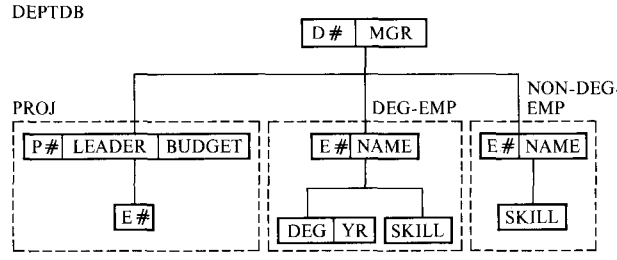



**Figure 13** Diagram of target structure.

There are obviously other ways of specifying the mappings in this translation. The CONVERT language is powerful enough that each translation analyst can specify the mappings in a way that is most natural to him.

### Data validation in conversion

One of the important tasks in the process of data conversion is data validation. Data validation in a broad sense includes the checking of the translation process per se or checking the information to see to what degree the conversion process is an information preserving mapping. For instance, the conversion on the above example is information preserving. However, if the target had been changed slightly such that E# no longer appears under PROJ, then the conversion is not information preserving. This area in itself requires much research and is not the subject of this discussion. In this paper only the simple problem of checking data to see if unexpected data exist is considered. Further, the work presented is only preliminary; more study is needed to make it complete.

There are two approaches to handle this class of errors: One is to attempt to correct the errors when they are found, and the other is simply to detect the errors without attempting to correct. Complete error correction, in our opinion, is generally impossible and attempts to correct errors without user interaction are usually futile. While providing interactive facilities including error correction is our long term goal, it is beyond the scope of this paper. Thus, we have limited ourselves to the problem of error detection at this time.

On investigating the common errors in data, one finds that many aspects of data checking can be described with the Data Definition Language, DEFINE, making it possible to detect the errors at a very early stage (i.e., when the source data is read). The following is a summary of the error checking capabilities existing in DEFINE.

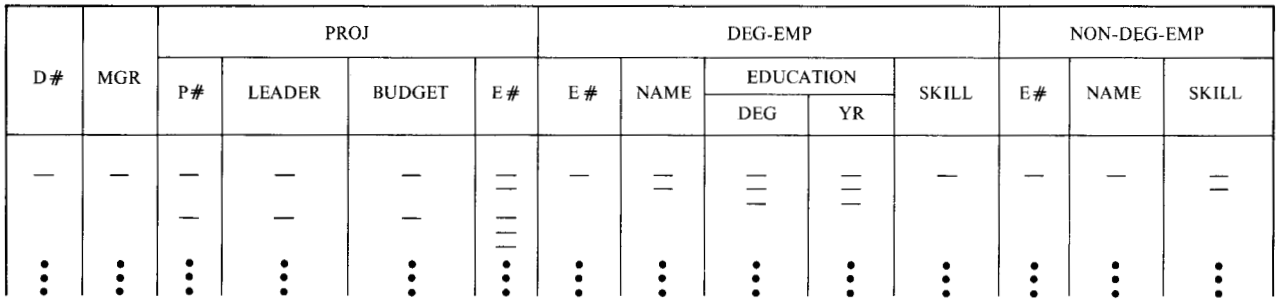



**Figure 14** Form representation of target structure.

1. *Record ordering* e.g., PART# in current record is expected to be greater than the PART# in the previous record. This can be specified in DEFINE with the ORDER clause, e.g., ORDERED ON PART# ASCENDING.
2. *Valid representation of data* This can be specified in DEFINE with the PICTURE clause. For example, to specify an 11 character item, N, in which: a) the first character must be a letter, b) the 4th and 8th characters consist of dashes (-), and c) the remaining characters are digits or letters (e.g., X9C-4BZ-M32). The specification of N could be precisely described by:

```
ITEM N:
  PICTURE CODE D IS 'A' OR '9';
  CHAR(PICTURE IS 'ADD[-]DDD[-]DDD');
END N;
```

3. *Acceptable values (or ranges of values) for specific fields.* Employee age can only be in the range between 18 and 65. This can be specified in DEFINE with the VALUE clause in the following manner: VALUE IS > 18 AND < 65.
4. *Conventions.* E.g., different spellings of "Expressway" should be coded as "EXPY". This can be specified in DEFINE with the TRANSLATE clause, TRANSLATE ('EXPWY', 'EXPRESS', 'EXPRESSWAY' TO 'EXPY').
5. *Record precedence relationship.* e.g., One type of record must precede another type. This can be specified in DEFINE with the correct structure of description.
6. *Mandatory occurrence.* e.g., A certain field must always have an occurrence. This can be specified in DEFINE with the unconditional descriptors or by giving a lower bound of greater than 0 for repeating or variable length data.

These are the kinds of errors that can be detected in the early stage of a conversion process and they can be specified within the powers of DEFINE [14]. There are other kinds of errors which cannot be conveniently specified in DEFINE without modification of the model. These can be roughly categorized as follows:

1. *Contextual validity* It is frequently the case that the validity of a data instance can only be determined in the context of other data instance. An example of this is as follows: If a department belongs to Research Division and is located in San Jose, the department number can only be in the range of K01 and K99.
2. *Validity which requires cross checking* For example, a PART# that exists in FILE 1 must also exist in FILE 2.
3. *Validity which requires computation.*

These categories can be checked by the following three validation statements in CONVERT.

```
VALIDATE (SC-EXPR1[,SC-EXPR2, ···])
VALIDATE IN CONTEXT (SC-EXPR1, SC-EXPR2 [, ···])
VALIDATE MEMBERSHIP (Member-tests)
```

(Note: SC-EXPR is discussed in a previous section and Member-tests is discussed in [15].)

Thus, examples of the validation statements for the fore-mentioned three categories are, respectively:

```
1. VALIDATE IN CONTEXT (EMP.DIV = 'RESEARCH', EMP.LOC = 'SAN JOSE',
   EMP.DEPT# >= K01 AND EMP.DEPT# <= K99);
2. VALIDATE MEMBERSHIP (FILE1.PART# CONTAINED IN FILE2 (PART#) OR
   SAME AS FILE2 (PART#));
3. VALIDATE (COUNT (NAME IN CITIZEN
        : CITIZEN.INCOME < 1000 OR CITIZEN.TAX < 100
      OR CITIZEN.INCOME/CITIZEN.TAX < 10) = 0);
```

Furthermore, validation can be either stated as a stand alone statement (as in the examples above), or be attached to a mapping specification. An example of the latter is

```
SELECT (A,B, C FROM F) VALIDATE (F.A + F.B
= F.C);
```

When standing alone, validity checking is done whenever the relevant Form is included in any mapping. When

attached to a mapping specification, validity checking is done only when that particular mapping specification is being carried out.

## Conclusion

In this paper we have described a methodology and model for data conversion. Unlike some previous works, the approach assumes that the source and target systems play an important role in the conversion process by transforming the source data into sequential files to be used as inputs to the conversion system and transforming the conversion system output into desirable target formats. Data conversion is a complex process, even when the input and output formats are limited to sequential files. It is believed that the process cannot be successful without user participation. Thus, the application of this method requires the users to describe the data structures of the linearized input and output files and to specify the mappings between the source and target data. Two languages were defined for this purpose: DEFINE for data description and CONVERT for mapping specification; both languages are in the process of implementation.

In defining any language the designers are always faced with the problem of tradeoffs between complexity and limited capability. The goal is to make these languages high level, nonprocedural, and simple to learn and use, but with sufficient capability to handle the common situations [27]. It is believed that these languages are indeed high level and nonprocedural by current standards and can handle most of the situations commonly encountered in data conversion. This belief is supported by the tests performed on several real life examples. Further, although the languages are designed to work in a specific environment, (DEFINE for sequential files and CONVERT for hierarchically structured data in tabular form), the model is general and the example in the paper demonstrated how a network structure can be decomposed into simpler forms acceptable in this model.

While the languages are relatively complete, several problems remain to be solved. First, given the capability of a data translation definition language like CONVERT, one can write mapping specification in many ways for the same task. It would be highly desirable if techniques can be developed so that some optimization can be done on an arbitrary mapping specification so that the translation process can be made more efficient. Second, one must define implementation algorithms to effect data conversion taking into account the time and resource constraints. Third, techniques to perform program translation, coupled to data conversion, must be developed to produce application conversion. Little work has been done on these problems; each of them requires much research. Our future effort is expected to be directed towards these problems.


## Acknowledgment

The authors are grateful to W. F. King III for his guidance and encouragement, to D. P. Smith for many helpful discussions and suggestion, to W. G. Tuel, M. C. Smyly, and G. C. Giannotti for providing real-life examples and to C. K. DeLong for typing this manuscript.


## References


1. J. P. Fry, "Introduction to Storage Structure Definition," ACM SIGFIDET Workshop on Data Description and Access, Houston, TX, 1970.
2. W. C. McGee, "Informal Definitions for the Development of a Storage Structure Definition Language," ACM SIGFIDET Workshop on Data Description and Access, Houston, TX, 1970.
3. J. W. Young, Jr., "A Procedural Approach to File Translation," ACM SIGFIDET Workshop on Data Description and Access, Houston, TX, 1970.
4. E. H. Sibley and R. W. Taylor, "Preliminary Discussion of a General Data-to-Storage Structure Mapping Language," ACM SIGFIDET Workshop on Data Description and Access, Houston, TX, 1970.
5. R. W. Taylor, "Generalized Data Base Management System Data Structures and Their Mapping to Physical Storages," Ph.D Dissertation, University of Michigan, 1971.
6. D. P. Smith, "An Approach to Data Description and Conversion," Ph. D Dissertation, University of Pennsylvania, 1971.
7. J. A. Ramirez, "Automatic Generation of Data Conversion Programs Using a Data Description Language (DDL)," Vols. I and II, University of Pennsylvania, May 1973.
8. J. P. Fry, D. P. Smith and R. W. Taylor, "An Approach to Stored Data Definition and Translation," ACM SIGFIDET Workshop on Data Description and Access, Denver Colo., 1972.
9. D. P. Smith, "A Method for Data Translation Using the Stored Data Definition and Translation Task Group Languages," ACM SIGFIDET Workshop on Data Description and Access, Denver, Colo., 1972.
11. E. H. Sibley and R. W. Taylor, "A Data Definition and Mapping Language," *Commun. ACM* **16,** 750 (1973).
12. B. C. Housel, V. Y. Lum and N. C. Shu, "Architecture to An Interactive Migration Systems (AIMS)," Proceedings of 1974 SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, MI, May 1-3, 1974.
13. S. Lin and J. Heller, "A Record Oriented, Grammar Driven Data Translation Model," Proceedings of 1974 SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, MI, May 1-3, 1974.
14. B. C. Housel, D. P. Smith, N. C. Shu and V. Y. Lum, "Data Translation Part II: DEFINE–A Nonprocedural Data Description Language for Defining Information Easily," Proceedings of ACM Pacific 75 Symposium, San Francisco, CA, April 17-18, 1975.
15. N. C. Shu, B. C. Housel and V. Y. Lum, "CONVERT–A High Level Translation Definition Language for Data Conversion," *Commun. ACM* **18,** 557 (1975).
16. J. A. Ramirez, N. A. Rin and N. S. Prywes, "Automatic Generation of Data Conversion Programs Using a Data Description Language," Proceedings of 1974 SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, MI, May 1-3, 1974.

17. E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun. ACM* **13,** 377 (1970).
18. E. F. Codd, "Further Normalization of the Data Base Relational Model," Courant Computer Science Symposia, Vol. 6, *Data Base Systems*, Prentice Hall, NY, May 1971.
19. E. F. Codd, "A Data Base Sublanguage Founded on the Relational Calculus," Proceedings of the 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control, San Diego, CA November 1971.
20. E. F. Codd, "Relational Completeness of Data Base Sublanguages," Courant Computer Science Symposia, Vol. 6, *Data Base Systems*, Prentice Hall, NY, May 1971.
21. D. D. Chamberlain and R. F. Boyce, "SEQUEL: A Structured English Query Language," ACM SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, MI, May 1-3, 1974.
22. G. Bracchi, A. Fedeli and P. Paolini, "A Language for a Relational Data Base Management System," Proceedings of the Sixth Annual Princeton Conference on Information Science and Systems, Princeton, NJ, March 1972.
23. G. Bracchi, A. Fedeli and P. Paolini, "Relational Data Base Management System," Proceedings of the ACM 72 Annual Conference, Boston, MA, August 1972.
24. G. Bracchi and P. Paolini, "Architecture of an On-Line Information Management System," Online 72 Conference Proceedings, 1972.
25. E. B. Altman, "A Hierarchic Presentation Independent Language (HRIL)1: Hierarchy Qualifications Functions," *Research Report RJ 1215*, IBM Research Laboratory, San Jose, CA, 1973.
26. P. L. Fehder, "HQL: A Set-Oriented Transaction Language for Hierarchically-Structured Data Bases," *Proceedings of ACM Annual Conference*, San Diego, CA, November 1974.
27. B. C. Housel and N. C. Shu, "A High-level Data Manipulation Language for Hierarchical Data Structure," *Proceedings of Conference on Data: Abstraction, Definition, and Structure* Salt Lake City, UT, March 1976.

*Received October 12, 1975; revised March 15, 1976*

*The authors are located at IBM Research Laboratory, 5600 Cottle Road, San Jose, CA 95193.*