Conversion of flat files and hierarchical data bases
Timo Niemi

Conversion of flat files and hierarchical data bases

ACADEMIC DISSERTATION

To be presented, with the permission of the Faculty of Economics and Administration of the University of Tampere, for public discussion in Auditorium AI of the University on October 18th, 1985, at 12 noon.

University of Tampere

Tampere 1985
Conversion of flat files and hierarchical data bases
Timo Niemi

Conversion of flat files and hierarchical data bases

University of Tampere

Tampere 1985
ABSTRACT

A precise specification for conversion of flat files and hierarchical data bases is presented. The specification allows the construction of the target data structure from many source data structures and it covers both data restructuring and data reformatting. These features have many important practical consequences. The comprehensive treatment of the data restructuring problem requires the manipulation of data at the schema, instance and value levels. We develop a formalism based on the data structure element (DSE) concept which is able to deal with the data restructuring problem generally and precisely at all these essential abstraction levels. Our specification contains a general restructuring process for hierarchical data bases which has many desirable new properties. The idea that abstract (formal) software specification is an essential phase in developing large and complex software has been widely accepted. In this study we apply the attribute method (grammars) to abstract software specification in the data base area and propose its use for translation oriented specification issues.

CR Categories and Subject Descriptors: D.2.1[Software Engineering]: Requirements/Specifications; E.1[Data]: Data Structures; E.2[Data]: Data Storage Representations; H.2.5[Database Management]: Heterogeneous Databases - data translation

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: data conversion, data restructuring, data reformatting, flat files, hierarchical data bases, abstract (formal) software specification, attribute method (grammars)
PREFACE

I wish to thank Professor Reino Kurki-Suonio for his guidance of my studies. In particular, his valuable criticism in the beginning of this work is gratefully acknowledged.

To Professor Pertti Järvinen and acting Professor Hannu Kangassalo I wish to express my gratitude for their encouragement at various stages of the work. The main parts of this work were written while Pertti Järvinen was the head of the Department of Mathematical Sciences/Computer Science. Hannu Kangassalo's excellent relations to the international data base community have been very useful for me, too.

I wish to thank Ph.L. Kalervo Järvelin for his careful reading of the various versions of this work. Many discussions with him have helped me to express my thoughts clearer and they have contributed new points of view.

I am grateful to Tampere City Science Foundation for the financial support. I wish also to express my thanks to the University of Tampere Editorial Board which has accepted my thesis for its scientific series.

Finally, my warmest thanks are due to my wife Ulla-Maija who has shared the side effects of this work with me.

Tampere, September 1985

Timo Niemi
CONTENTS

INTRODUCTION 1

PART I: A seven-tuple representation for hierarchical data structures

1. Introduction 151
   1.1. Data structure diagrams 151
   1.2. Tabular representation 151
   1.3. Requirements of the formal representation 152

2. The formal description of hierarchical data structures 153
   2.1. The functions for index manipulation 153
   2.2. The definition of a data structure element 153

3. Analysis of hierarchical data structures 155

4. Conclusions 156

PART II: Formal restructuring functions for hierarchical data bases

1. Introduction 385

2. The representation for hierarchical data bases 388
   2.1. The manipulation functions for indices 389
   2.2. The definition of the data structure element and its projection 389

3. The description of the target schema and the definition of the analysis functions 393

4. Exact definition of the restructuring functions 398
   4.1. The transformation of a hierarchical data base into its internal representation 399
   4.2. The functions for constructing target data bases 402

5. Conclusions 408

PART III: Specification of data restructuring software based on the attribute method

1. Introduction 425

2. The formal description of data structures 429

3. The constructing principle of abstract syntax and the abstract syntax for data restructuring 433
   3.1. The constructing principle of abstract syntax 434
   3.2. The abstract syntax for data restructuring 435
4. The definition of the attributes
   4.1. Informal characterization of the attributes

5. Informal consideration of the specified restructuring software

6. Conclusions

Appendix

PART IV: The specification of data reformatting in data conversion

1. Introduction

2. Primitive concepts, operations and notational conventions for data reformatting specification
   2.1. Auxiliary functions needed in data encoding

3. The encoding of logical data units
   3.1. The encoding of data items
      3.1.1. The encoding of non-numeric data item values
      3.1.2. The encoding of numeric data item values
   3.2. The encoding of records
   3.3. The encoding of flat files and hierarchical data bases

4. Specification of data reformatting aspects in data conversion
   4.1. The extension of the abstract syntax for data reformatting
   4.2. The definition of the attributes for data reformatting

5. Conclusions

CONCLUSIONS OF THE THESIS

THE COMMON REFERENCE LIST OF THE INTRODUCTION,
PART IV AND CONCLUSIONS OF THE THESIS
INTRODUCTION

Informally we can say that data conversion is a process which transforms data in a given hardware/software environment into such a representation of data which can be manipulated in the same or in a different hardware/software (briefly hw/sw) environment. The existing data or the data to be converted are called source data and the converted data are called target data. Often the conversion of application programs occurs at the same time. Su et al. have introduced a detailed model on how data conversion is associated with the conversion of application programs [36]. In this work we concentrate on data conversion.

It is possible that conversion costs of data and programs are a major factor in acquiring new hardware or software [15]. Likewise large conversions can often take one or more years [7]. Therefore it is very important to develop advanced tools to facilitate the conversion of data and programs. In this study we develop and specify precisely tools in terms of which we can conveniently to convert flat files and hierarchical data bases - also in complex data conversion cases.

Two essential parts are usually distinguished in data conversion: data restructuring and data reformatting (see e.g. [16]). Depending on the data conversion case it contains either data restructuring or data reformatting or both of them. In this study we shall also characterize those conversion cases occurring in practice in which our specifications can be utilized. In data restructuring we concentrate on transforming structural relationships among logical (user) data whereas data reformatting takes into account changes in the storing principles of data (for example, changing the EBCDIC character code to the ASCII character code).

It is not necessary that the hw/sw environment changes in data restructuring. For example, we may have to restructure structural relationships of stored data in a certain hw/sw environment due to changes in the requirements of users or new applications. On the contrary, data reformatting reflects changes in the hw/sw environment. It has been widely recognized that data restructuring is the most complex part in the data conversion process which requires both data restructuring and data reformatting (see e.g. [7]). In this work we emphasize restructuring of flat files and hierarchical data bases.
Navathe & Fry have recognized three essential abstraction levels (schema level, instance level and value level) in data restructuring [28]. The highest level at which the restructuring of logical (user) data can be described is the schema modification level. In turn, the schema modification have a direct effect on the instances of data. (instance level) and the instance operations required reduce finally to some basic operations of data item values such as eliminating or duplicating data item values etc. (value level) [28].

If we want to deal with the data restructuring problem exhaustively we must deal with it at these three abstraction levels. So far, there has been a lack of such a formalism which is able to deal with the data restructuring problem generally and precisely at all three abstraction levels. In this work we develop such a formalism. Our formalism is based on the concept of the data structure element (briefly dse) which allows flexible and precise interfaces between these different abstraction levels. Both source and target data structures are described as dse's. A dse is defined recursively with three construction rules. These construction rules have an essential role throughout this thesis.

When considering existing data restructuring languages or approaches to data restructuring one notes that many of them have not been implemented. Furthermore, it is typical that most works in the data conversion area are descriptive - not rigorous in nature. Usually data restructuring is illustrated in terms of different visualization means. We shall relate our dse description to these visualization means and show the benefits of the dse description.

In this study we shall develop a general restructuring process for hierarchical data bases which has many desirable new properties. In the structural sense the minimal information, which is needed for data restructuring, is the source data base and the target schema. We have this as a starting point for data restructuring. Namely, our restructuring process culminates to the definition of such a restructuring function whose parameters are a hierarchical source data base and a hierarchical target schema. This function constructs the hierarchical data base according to the target schema provided that it is constructable from the given source data base.
The developed restructuring function is based on the DSE description and in its definition several other functions are needed. Our restructuring process has two main phases. In the first main phase the hierarchical data base being restructured is transformed into its internal nonhierarchical representation. This phase is information preserving, i.e. we do not lose any information which is essential in the structural sense. The second main phase constructs the target data base from the internal representation by analyzing the given target schema.

In practice there are many restructuring cases which require the construction of a target data structure from many source data structures. In this study we give also an exact software specification which is able to construct the target data structure from many source data structures. In this specification the developed restructuring function and its components play an important role. This specification crosses also the boundaries of two data models because the source data structures being restructured and the target data structure can be based either on the flat file data model or on the hierarchical data model. This property has many important practical consequences.

In the data base literature the equivalence of schemas based on different data models has been studied mainly from the view point whether or not they represent the same universe of discourse (see e.g. [?]). However, only little attention have been paid to restructuring actual structural relationships among data when transforming data based on one data model into data based on another data model. Sometimes this is even impossible because certain structural relationships must exist among source data in order to be able to construct the target data structure. Our specification checks that there are the necessary structural relationships among source data in order that the target data structure can be constructed.

In this thesis we consider also those data conversion cases which require, in addition to data restructuring, data reformatting, too. For example, we are facing this kind of a conversion task when, in the context of the exchange or acquisition of a DBMS, the computer is also changed. In this work we specify both essential aspects associated with data conversion. Therefore we extend our specification to take into account also the data reformatting aspects.
Our starting point to data reformatting is that we utilize software available in the source and target systems as effectively as possible. This means that the load and unload utilities of file and data base systems have a central role in the sense that they provide a meaningful interface between the conversion process and the software available in a certain environment.

The specification of data reformatting aspects is based on the encoding of data structures. The result of the encoding process of a data structure is always some bit string. In general, we have numerous variations to realize data structures based on the flat file or hierarchical data model. Likewise, numeric data can be represented in various and complex forms. Therefore we must limit our consideration to some typical encoding situations only. For example, we shall represent all numeric data in the zoned (decimal) form [5], [21].

We shall demonstrate that the encoding features defined in this thesis are sufficient to create the bit string representation required by IMS load utilities from the dse description of a hierarchical data base which itself is independent of any dbms. We choose IMS [14], [38] to our sample system because it represents a typical hierarchical dbms which is large, complex and widely used.

The importance of precise specifications in the data base area has been emphasized in many contexts, for example at Very Large Data Base Conference in Mexico City a panel on "Type Specifications and Data Bases" (see the summary report [12] of this panel) was organized to discuss this issue. Several authors have observed that the abstract (formal) specification methods developed in programming language research can be applied to the precise specification in the data base area, too. The idea that abstract software specification is an essential phase in developing large and complex software has been also widely accepted.

Kurki-Suonio has developed, for the definition of programming languages, a specification method which is based on applying the attribute method (grammars) in the context of abstract syntax [25]. In this study we apply this method to abstract software specification and adapt it to the data base area. It has been widely accepted that one needs complementary specification
methods to formalize the various aspects of data base applications (see e.g. [17], [39]). In this study we propose the use of the attribute method for specifying translation oriented specification issues in the data base area. So far, the attribute method has been very little used in the data base area.

This thesis consists of four parts which are closely tied to each other.

The PART I contains the article
In this part we develop the dse description of hierarchical data structures and give both the explicit construction operations and implicit rules for it. Here our emphasis is to relate our formal representation to the conventional visualization means used in data restructuring. Likewise, we define some general analyzing functions for hierarchical data bases on the basis of our formalism.

The PART II consists of the article
In this part we define precisely the general restructuring function of hierarchical data bases. This function is based on the dse description and we can apply it some analyzing functions defined in PART I. The dse definition in PART I is extended considerably by defining a dse and its projection at the same time. In this extension we need also the concept of the zero element in order to make the representation well defined. Contrary to PART I our dse description of a source data base does not contain the key component because we do not necessary need this information in data restructuring.

The PART III consists of the article
Here, we give an exact specification which is able to construct a target data structure from many source data structures. Data structures can be based either on the flat file data model or on the hierarchical data model. In this specification the restructuring process defined in PART II has a central role. Therefore we present it in a compact form in the appendix of this article. In other words, if the reader is familiar with PART II it is not necessary to read this appendix at all.
In the PART IV we extend the specification of PART III by specifying data reformatting aspects, too. PART I, PART II and PART III have their own page numbers and references whereas introduction and conclusions of this thesis and PART IV have their common reference list at the end of the thesis. We shall refer to the articles in PART I, PART II and PART III outside of these parts with NIE-1, NIE-2 and NIE-3 respectively.

Throughout this thesis, if there is no danger of confusion, we shall denote a set consisting of a single element by the element itself. For example, if \( f \) is the function from \( A \) to \( B \) and it yields a single element, then we use, for brevity, the expression
\[
\bigcup_{i \in A} \{f(i)\}
\] instead of
\[
\bigcup_{i \in A} f(i).
\]
PART I
A SEVEN-TUPLE REPRESENTATION FOR HIERARCHICAL DATA STRUCTURES

Timo Niemi
University of Tampere, Department of Mathematical Sciences, Computer Science, P.O. Box 607, SF-33101 Tampere, Finland

(Received 14 June 1982; in revised form 5 April 1983)

Abstract—Different kinds of data structure diagrams, especially Bachman's, and tabular representations are very often used to visualize hierarchical data structures. The popularity of these visual modes of representation is due to the possibility they afford of describing data structures clearly and regardless of any implementation. However, it has been observed in diverse contexts that instead of the visualization method we need a general and exact representation containing the same structural components as those visual representations. In this paper we introduce a seven-tuple representation which can be used for exact description of any hierarchical data structure. We give both the explicit construction operations and the implicit rules which can be utilized in forming any hierarchical data structure. We show that the formal representation introduced contains the descriptive power of the visualization methods mentioned above. Furthermore we examine how, in the context of this representation, it is possible to define flexibly general analyzing functions for hierarchical data structures.

Keywords: hierarchical data structures, data structure diagrams, tabular representation, recursive data structures

1. INTRODUCTION

Several database management systems are based on the hierarchical data model. These DBMS's presuppose that all data structures used in applications are organized hierarchically. In the context of the hierarchical data structures there appears a variety of situations such as to present the result of the logical data base design, to illustrate the effects of DML (Data Manipulation Language) commands, to describe the transformation of data structures and so on, where different visualization methods are utilized. The basic requirement for any method of visualizing data is that it describes structural properties in a simplified manner. Bachman's diagrams[1] and tabular representations are the most typical modes of visualization associated with hierarchical data structures. We shall compare these visual representations with the formal representation introduced in this paper. In Fig. 1 the same hierarchical data structure is described in three different ways.

1.1 Data structure diagrams

Bachman's data structure diagrams have all the schema level three kinds of descriptive components: boxes, arcs and names. The names are either record type names or data item names. In the context of hierarchical data structures we have no need to name arcs as would be necessary in the context of network data structures. The boxes represent record types and the arcs between record types express the connections among them. Data item names are connected by arcs to those records types to which they belong. In data structure diagrams the key of a record type is expressed, for example, by underlining. Considering the data structure diagram of Fig. 1 it is easy to observe that the record type C depends immediately and the record type G indirectly on the root record type X. Further, the data items A and B belong to the record type X and the data item D is the key of the record type C.

Associated with a data structure diagram we need a separate figure for the instance level. Also, in the description of the instance level arcs can be used for expressing connections between records (instances of record types). In Fig. 1 there are two hierarchical data base trees (instances of a hierarchical data structure). Data item values are the basic elements in the description of the instance level.

G. A. Dale and N. B. Dale have used diagrams consisting of record types or their instances for illustrating different restructuring cases of hierarchical data structures[2, 3]. Navathe has also created a graphical representation of data, the so-called schema diagram, which takes into account more explicitly than Bachman's diagram how a data structure is realized at the instance level[4, 5]. Although the schema diagram is a description of the schema level it also includes information about the classification of relationship types (e.g. many to many, one to many, etc.) between instances of the record types. Also, associated with schema diagrams we need a separate figure for visualizing the instance level.

1.2 Tabular representation

It is typical of a tabular visualization that it contains both the schema and instance levels. In the tabular visualization of a hierarchical data structure record type names, data item names and parentheses are used as description components. A record type is described by listing after the record type name those data item names which belong to this record type. The hierarchy is expressed by using parentheses one within the other. The keys of the record types can be expressed, e.g. by underlining.

The instance level is described by grouping to-
1.3 Requirements of the formal representation

Both the data structure diagram and tabular representation afford a possibility of visualizing the actual structural properties of hierarchical data structures in a simplified way independent of implementation. These same advantages must also be included in the formal representation. Further, the formal description of hierarchical data structures must contain both the schema and instance levels because the manipulation of data occurs at both levels.

In addition, the representation has to be general, which means that it must permit any finite number of instances and hierarchy levels. Associated with our representation it must be possible to define analyzing and manipulation functions flexibly, generally and exactly. We shall show, moreover, that associated with our formal representation we can restrict the
number of structural components. Thus, we get the data structures allowed in a given data base management system based on the hierarchical data model.

2. THE FORMAL DESCRIPTION OF HIERARCHICAL DATA STRUCTURES

Substructures of hierarchical data structures are likewise typically hierarchical. Therefore, recursion is a natural way to define and to manipulate hierarchical data structures when the aim is generality and completeness. Ehlers[13], for example, has recognized this feature as typical for hierarchical data structures. Also in this paper hierarchical data structures are defined recursively.

In a variety of contexts a popular means of formal description is the so-called n-Tuple representation, where all components included in the description are expressed explicitly. For example, in the case of the description of data structures Ausillego et al.[14, 15] define data base schemas based on the relational model as n-tuples. In this paper any hierarchical data structure is described as a seven-tuple which, in the mathematical sense, represents a relation, the so-called unnormalized relation according to Codd's classification[6].

In the formal description of data structures the primitive types are integers (int), real numbers (real) and characters (char). All primitive types are associated with data items whereas record types of a hierarchical data structure are power sets in the mathematical sense. The construction rules must be treated in such a way that any hierarchical data structure can be constructed with them, beginning with data items which are the smallest logical data units in a hierarchical data structure.

A hierarchical data structure and the components needed in its construction are in general called data structure elements. In this paper. A data structure element is a relation which is represented as a seven-tuple \((r, X, KEY, RN, DN, I, f)\) where

\(r\) is an instance of the relation,
\(X\) is a relation type,
\(KEY\) is a set of keys,
\(RN\) is a set of record type names,
\(DN\) is a set of data item names,
\(I\) is an index set and
\(f\) is a naming function.

Next, we consider the intuitive meaning of the components of the seven-tuple, when it describes a hierarchical data structure. The component \(r\) is constructed from those data item values which are included in the hierarchical data structure. The component \(X\) always describes a set, in other words, it describes what kind of a mathematical concept the data structure represents. The component \(KEY\) expresses the keys of the record types of a hierarchical data structure. The components \(RN\) and \(DN\) contain all record type names and data item names in a hierarchical data structure. The structural properties of a hierarchical data structure are expressed by indices. The indices needed are included in the component \(I\). The component \(f\) associates one and only one index with each record type name and data item name.

In the recursive definition of the data structure elements we use the concept of data structure elements of the same type and we apply certain index manipulation functions. Therefore, before definition the concept in question and the functions being applied must be introduced.

Definition

If \(I_1, I_2, \ldots, I_n\) are data structure elements such that \(i = (r, X, KEY, RN, DN, I, f), i = 1, \ldots, n\) and if for any two data structure elements \(i, i' (i \neq i')\) the following condition holds:

\[X = X_1 \land KEY = KEY_1 \land RN = RN_1 \land DN = DN_1 \land I = I_1 \land f = f_1,\]

then the data structure elements are of the same type. This means that data structure elements of the same type can differ from each other with respect to the relation instance component.

In our formalism finite tuples are used. The tuples are denoted by angle-brackets, e.g. \(\langle 1, 2, 3 \rangle\)

The denotation \(\langle i, j \rangle\) means a tuple where the first element is \(i\) and others form the tuple \(j\). Analogically, \(\langle i, f \rangle\) is a tuple whose last element is \(i\). Also, in the concatenation of two tuples \(\xi, \zeta\) the denotation \(\langle \xi, \zeta \rangle\) is used; if for example \(\xi = \langle a, b \rangle\) and \(\zeta = \langle c, d \rangle\) then \(\langle \xi, \zeta \rangle = \langle a, b, c, d \rangle\). The concatenation of tuples is an associative operation. In other words \(\langle \xi_1, \xi_2, \xi_3 \rangle = \langle \xi_1, \langle \xi_2, \xi_3 \rangle \rangle\).

2.1. The functions for index manipulation

In the seven-tuple representation of a data structure element the index set \(I\) appears as a component. The elements of this set are called indices. Indices are finite tuples consisting of natural numbers. A single element of a tuple is called a component of an index.

The following functions are used in index manipulation:

- The primitive function \(elem(\xi)\) gives the number of components of the index \(\xi\), for example \(elem(\langle 1, 4, 5 \rangle) = 3\).
- The function \(max(\xi)\) gives the number of those indices of \(I\) which consist of only one component.

\[\max(\xi) = |\xi| \lor \xi \notin elem(\xi) = 1|\]

- The operation \(\circ\) is defined between the index set \(I\) and the non-negative integer \(n\) in the following way:

\[I \circ n = \{i + n, \xi \mid (i, \xi) \in I\}\]

For example, if \(I = \{\{1\}, \{1, 1\}, \{1, 1, 1\}, \{2\}\}\) then \(I \circ 2 = \{\{3\}, \{3, 1\}, \{3, 1, 1\}, \{4\}\}\) and \(\max(I) = 2\).

2.2. The definition of a data structure element

A data structure element is a seven-tuple \((r, X, KEY, RN, DN, I, f)\) which is constructed recursively by applying the following rules finitely.
A data item is a data structure element $(r, D, KEY, \emptyset, \langle A \rangle, \{1\}, f_A(A) = \langle 1 \rangle)$ where $r$ is a data item value with the property $r \subseteq D$; $D$ is some primitive type (int, real, char); $KEY$ is either $\emptyset$ (which indicates that this data item is not used in forming a key) or a set $\{A\}$ (this data item is used in forming a key); $\emptyset$ means that the data structure element does not contain any record type name; and the set $\{A\}$ consists of a data item name; the index set contains the index $\langle 1 \rangle$: $f_A(A) = \langle 1 \rangle$ expresses the index associated with the data item name $A$.

(2) If $t_1 = (r_1, X_1, KEY, RN_1, DN_1, I_{r_1}, f_{r_1})$ and $t_2 = (r_2, X_2, KEY_2, RN_2, DN_2, I_{r_2}, f_{r_2})$ are two data structure elements such that $(RN_1 \cup DN_1) \cap (RN_2 \cup DN_2) = \emptyset$ then the notation $\langle t_1, t_2 \rangle$ constructs the data structure element $\langle r_1, X_1, KEY, RN_1 \cup RN_2, DN_1 \cup DN_2, I_{r_1 \times r_2}, f_{r_1 \times r_2} \rangle$ where

$$I_{r_1 \times r_2} = I_{r_1} \cup I_{r_2} \cup \{max(I_{r_1})\}$$

and

$$f_{r_1 \times r_2} = f_{r_1}(a) \cup f_{r_2}(a) \mid a \in (RN_1 \cup DN_1 \cup RN_2 \cup DN_2)$$

such that

$$f_{r_1 \times r_2}(a) = f_{r_1}(a) \mid a \in (RN_1 \cup DN_1 \cup RN_2 \cup DN_2)$$

(3) If $T = \{(r_i, X_i, KEY, RN_i, DN_i, I_{r_i}, f_{r_i})\mid i = 1, \ldots, n\}$ is a set consisting of data structure elements of the same type and $A \in RN \cup DN$ then the notation $\{A\} \in T$ constructs the data structure element $\langle r_i, X_i, KEY, RN \cup \{A\}, DN_i, I_{r_i}, f_{r_i} \rangle$ where $P(X)$ is the power set of the relation type $X$,

$$I_{r_i} = \{1\} \cup \{1, \langle 1 \rangle \mid a \in \{A\}\}$$

$$f_{r_i}(a) = \{1\} \cup \{1, f_{r_i}(a)\} \mid a \in (RN \cup DN)$$

As a consequence of the construction operations the components of the seven-tuple representation of any data structure element have the following properties:

1. $r \subseteq X$
2. $RN \cap DN = \emptyset$
3. $\|RN\| + \|DN\| = |I|$.
4. $KEY \subseteq DN$.
5. The naming function $f_{r} : RN \cup DN \rightarrow I$ is bijective.
6. The relation type $X$ is a cartesian product $T_1 \times \cdots \times T_n$ where each $T_i (1 \leq i \leq n)$ can be either a primary type (int, real, char) or the power set $P(Y)$ of some other relation type $Y$.

The elements of the index set $I_y$, which are associated with the relation type $X = T_1 \times \cdots \times T_n$, are $n$-tuples such that

$$\forall \langle i_1, \ldots, i_n \rangle \in I_y, \langle i_1, \ldots, i_n \rangle \in I_y \iff T_i = P(Y) \text{ and } \forall y_i$$

Because the hierarchical data structures are also data structure elements they satisfy the mathematical properties described above. In fact, the properties 1-7 build up the set of implicit rules which can also be used to define any hierarchical data structure.

A data structure element constructed by the second rule has no equivalent term in the data base literature. We can generalize this operation so that it is possible to construct records by it. We shall use the notation $\langle t_1, t_2, \ldots, t_n \rangle$ to mean the following sequence of operations $\langle \ldots, \langle t_1, t_2 \rangle, \ldots, t_n \rangle$. The third rule is applicable for grouping records. This operation is needed because in a hierarchical data structure there can be several records of the same type depending on one record.

The component $KEY$ contains those data item names which are essential for identifying records from the viewpoint of the user. It depends entirely on the real world which are such data items. Therefore, the $KEY$ component of a data item can be an empty set, which indicates that this data item is not essential for identifying records.

Although the formal description of a hierarchical data structure can, in practice, be made easier by direct use of the set of properties 1-7, we illustrate below the sufficiency of the operations given to construct a hierarchical data structure from the data items (the smallest logical data units). We inquire what kind of an operation sequence produces the sample data structure of Fig. 1. We shall denote the data items $(r_i, D_i, 0, 0, [K], \{1\}, f(K) = \{1\})$ and $(X_i, D_i, [K], [K], \{1\}, f(K) = \{1\})$ by $D^I_i$ and $X^I_i$, where $K$ is a data item name; $D$ is some primitive type (int, char, real) and $\{a\}$ indicates that this data item is used as a key component. The construction of the sample data structure $t$ can be described by the following sequence of operations.

$$t = X^I_1 \cup \{X^I_2 \cup \{X^I_3 \cup \{X^I_4 \cup \{X^I_5 \cup \{X^I_6 \cup \{X^I_7\}\}\}\}\}\}$$

For example, the result of the operations $\{X^I_6 \cup \{X^I_7\}\}$
A seven-tuple representation for hierarchical data structures

Consider a seven-tuple \((r, e, K E Y, R N, D N, I, f)\)

We can note that the components \(KEY, RN, DN, I, f\)

describe things belonging to the schema level in a data base environment, whereas the component \(r\) describes the instance level. The relation type \(X\) determines what kind of a mathematical concept a data structure represents. The type of a hierarchical data base is always \(P(\zeta)\) where \(\zeta\) is any cartesian product and a data base tree is of \(\zeta\). In the sample data structure (Fig. 1) there are two data base trees \((a_1,\ldots,\gamma)\) and \((\beta_1,\beta_2,\ldots,\gamma)\) which are of type \(\zeta\) and 

\(\zeta\) is always associated with the root record type name.

In our seven-tuple representation, the structural features are hidden in indices, i.e., in the component \(I\). One and only one index is associated with each data item name and the entire index \(I\) represents the relationship between the indices. The index \(\{\zeta\}\) is always associated with the root record type name.

The relation type \(X\) may also be used to express the type of a certain substructure. Namely, \(X_i(\zeta, i)\) means that type with which an index \(\zeta\) is associated in the relation type \(X_i\). If in the same data structure we want to know the types of the data item \(A\) and the record type \(G\) we achieve them by the expressions \(X_i(A) = X_{i(A)}\) and \(X_i(G) = X_{i(G)}\). Of course, the inverse function \(f\) is bijective. In the inverse function \(f\) we apply the same rules to get the record type name associated with each data type name. The inverse function \(f\) is bijective on the record type name and also vice versa.

Moreover, the seven-tuple representation \(\{(d_0, e_1, \ldots, d_n)\}\) represents any hierarchical data structure we can give exact and formal definitions concerning the structural components of a hierarchical data structure. As an example, consider the definitions of the branching and non-branching record types. Informally, a record type is called branching if two or more record types depend on it and in other cases a record type is non-branching. To define these formally we use the function \(\text{successor}\) whose argument is an index \(\zeta\) associated with a record type, i.e.

\[\text{successor}(\zeta) = \{(\zeta, i) | (\zeta, i) \in \text{RN} \wedge (\zeta, i) \neq (\zeta, i) \}

The function \(\text{successor}\) gives the indices of those record types which depend directly on the record type represented by the index \(\zeta\). Using the function \(\text{successor}\) we get the following definitions for the branching and non-branching record types. If \(\text{successor}(f_i(R)) \geq 2\) then the record type \(R \in \text{RN}\) is branching, otherwise non-branching. In the sample data structure the root record type \(X\) is branching because \(\text{successor}(\text{successor}(X)) = \{(\zeta, 3)\} \neq \{(\zeta, 1), (\zeta, 4)\} \). Next consider some general analyzing functions for hierarchical data structures. The function \(\text{i-dependency}\) returns the index of that record type on which the record type given as an argument immediately depends. The argument \((\zeta, i)\) is the index associated with a record type, i.e., \(f\).

\[\text{i-dependency}(\zeta, i) = \zeta

The sample data structure the function \(\text{i-dependency}\) returns the index \(\{(\zeta, 1), (\zeta, 4)\}\). However, if \(\zeta\) is of type \(\zeta\) and \(\text{successor}(\zeta) \neq \emptyset\) then \(\text{successor}(\zeta) = \{(\zeta, i) | (\zeta, i) \in \text{RN} \wedge (\zeta, i) \neq (\zeta, i) \}

The term \(\text{leaf record type}\) is applied to those record types on which no record type depends either immediately or indirectly. The function \(\text{leaf}\) finds all the leaf record types of a hierarchical data structure and we define it as follows:

\[\text{leaf}(f_i) = \{(\zeta, i) | (\zeta, i) \in \text{RN} \wedge \text{successor}(\zeta, i) = \emptyset\}

In the sample data structure the set \(\{(\zeta, 1), (\zeta, 4)\}\) is the set of leaf record types. The corresponding record type names are obtained with

\[\bigcup_{(\zeta, i) \in \text{leaf}(f_i)} f_i = \{G, J\}\]
The key of a record type can be obtained by the function \( r\text{-key} \). The index \( i \) associated with a record type is the argument of this function, i.e. \( f_{r\text{-key}}^{-1}(\xi) \in RN \). The result is a set of the data items forming the key:

\[
\{ (\xi, i) \mid (\xi, i) \in I, r\text{-key}(\xi, i) = KEY \}
\]

In the sample data structure \( r\text{-key}(\xi, i) = (\xi, 3, 1) \), this means that the key set of the record type \( C \) (namely \( f_{r\text{-key}}^{-1}(\xi, 3, 1) = C \)) consists solely of the data item \( D(f_{r\text{-key}}^{-1}(\xi, 3, 1) = D) \).

The function \( r\text{-content} \) returns the data items of a record type in a hierarchical data structure. The argument is the index \( i \) associated with the record type, i.e. \( f_{r\text{-content}}^{-1}(\xi) \in RN \). The function is defined as follows:

\[
r\text{-content}(\xi) = \{ (\xi, i) \mid (\xi, i) \in I, r\text{-content}(\xi, i) \in DN \}
\]

In the sample data structure \( r\text{-content}(\xi, 3, 1) = \{ (\xi, 3, 1), (\xi, 3, 2), (\xi, 3, 3) \} \), the corresponding data item names are

\[
\{ D, E, F \}
\]

At the logical level, the access path type (or access path schema) \( \{d\} \) can be defined informally as a route to a record type. In hierarchical data structures, the access path type to a certain record type can be defined by specifying those record types on which the record type depends immediately or indirectly. The function \( a\text{-path} \) obtains the set of indices of those record types which form the access path type to the record type given as an argument. The argument \( \text{ind} \) has the property \( f_{a\text{-path}}^{-1}(\text{ind}) \in RN \).

\[
a\text{-path}((\text{ind})) = \begin{cases} \emptyset, & \text{if } \text{ind} = \emptyset; \\ \text{ind}, & \text{if } \text{ind} = \emptyset; \text{the root record type with which the index } (\xi) \text{ is associated}. \end{cases}
\]

In the sample data structure \( a\text{-path}(\{d\}) = \{ (\xi, 3, 4) \} \) is the index set \( \{ (\xi, 3, 4), (\xi, 3, 1), (\xi, 1) \} \).

We can combine these functions in a simple way. For example, the expression \( \bigcup_{\text{ind} \in a\text{-path}(\text{ind})} r\text{-content}(\text{ind}) \) gives the indices of those data items which belong to the access path type leading to \( \text{ind} \). In the sample data structure

\[
\bigcup_{\text{ind} \in a\text{-path}(\text{ind})} r\text{-content}(\text{ind}) = \{ (\xi, 3, 4), (\xi, 3, 1), (\xi, 1) \}
\]

The seven-tuple \( (\xi, \chi, KEY, RN, DN, I, f_{r\text{-key}}) \) is a general representation for any hierarchical data structure. However, dbms's based on the hierarchical data model have restrictions for the number of the permitted structure components. For example, IMS permits in its hierarchical data structures at most 255 record types (segment types in IMS terminology) which must be organized at most into 15 hierarchy levels. It is also possible to express such restrictions in the context of the seven-tuple representation. If we have a hierarchical data structure according to IMS, then the following conditions must hold in a seven-tuple:

1. \( |RN| \leq 255 \)
2. \( \forall i \in I, r\text{-key}(\xi, i) > 15 \Rightarrow f_{r\text{-key}}^{-1}(\xi) \in RN \)

4. CONCLUSIONS

In this paper a general and formal seven-tuple representation for hierarchical data structures has been presented. Both the explicit construction operations and the implicit rules for constructing any hierarchical data structure have been given. The representation is not bound to any hierarchical dbms. Although the representation is independent of implementation, we have shown that it enables us to restrict the number of the structure components permitted in a certain hierarchical dbms.

Our representation contains general descriptions of the schema and instance levels and it has the descriptive power of well-known visualizations like Bachmann's diagrams and tabular representation. In addition, the representation includes a component which describes what kind of a mathematical concept a data structure represents.

The representation allows flexible definition of general and exact analyzing and manipulation functions for hierarchical data structures. To illustrate this, some analyzing functions were defined. The seven-tuple representation is a convenient tool for exact definition of software products which manipulate hierarchical data structures. This representation, except for the KEY-component, has been used to define, in an exact way, the restructuring of hierarchical data structures at schema and instance levels [17, 18].

REFERENCES

PART II
Formal Restructuring Functions for Hierarchical Data Bases

Timo Niemi

Received July 1983; revised October 1983

Traditionally data restructuring of hierarchical data structures has been illustrated using examples in the context of the expressions of some transformation language. In this paper we define general and exact analysis and restructuring functions for hierarchical data bases. These functions describe restructuring of structural relationships among data structures, i.e. the construction of a target data base from a source data base is defined in detail. In fact, the analysis and restructuring functions define the tasks to be performed by the restructuring software at run time.

KEY WORDS: Hierarchical data structures; data restructuring; automatic data base conversion.

1. INTRODUCTION

There are many situations such as the changing requirements of a user, taking into use a data base management system, transferring data in a computer network etc. where data in files and data bases must be restructured. Data restructuring is one important part of data conversion. Depending on the case data conversion can in addition contain data reformatting, too, which takes into account changes in the storing principle of the data (for example the EBCDIC character code changes to the ASCII character code).

For the insufficient logical and physical data independency we must often convert application programs, too, in order that they are able to manipulate the converted data. Conversion costs of data and programs can be a major factor in acquiring new hardware or software. Also, the time required by conversion is often long. For these reasons there have been

1 University of Tampere, Department of Mathematical Sciences, Computer Science, P.O. Box 607, SF-33101 Tampere 10, Finland.
many efforts to develop and implement software to transform application programs and data flexibly into such a form that allows manipulation in the new environment. Su et al. have described a detailed model on how data conversion is associated with the conversion of application programs. In this paper we concentrate on data conversion, especially on data restructuring.

Considering different implementations of data converters and approaches to data conversion we note that they are based on analogical main phases. Very often, they can be seen to consist of the sequence of three main modules: READER, RESTRUCTURER and WRITER. The READER-module is responsible for reading the data to be converted (source data) and for transforming the data into an internal form processable by the RESTRUCTURER. The RESTRUCTURER-module performs all operations which are required in restructuring structural relationships among data. The WRITER-module has responsibility for writing the restructured data (target data) onto some storage medium. Target data can be in the different software/hardware environment than source data. The main modules in question can be split into many submodules. For example, we can distinguish in the READER- and WRITER-modules the PHYSICAL READER- and PHYSICAL WRITER-modules that deal with low level hardware details. In this paper we concentrate on defining the functions of the RESTRUCTURER when the RESTRUCTURER manipulates data structures organized hierarchically.

Many nonprocedural languages have been developed for restructuring data structures based on different data models. For example, Honeywell has developed a prototype that is able to restructure sequential files. Nonprocedural languages intended for restructuring network data structures are presented in. Shneiderman and Thomas have described an architecture for restructuring relational data bases when the relational algebra is used as data manipulation language. CONVERT of IBM, UMTDL (University of Michigan Translation Definition Language), and CDTL (Common Data Translation Language) of SDC are nonprocedural languages for restructuring hierarchical data structures. Also, ADAPT developed by Bell Laboratories can be used for restructuring hierarchical data structures.

Although there are nonprocedural transformation languages for hierarchical data structures, many of them have not been implemented. Usually, the expressive power of these languages is described by virtue of examples. Typically, a sample schema and some sample instances are used to illustrate the effects of the expressions of the transformation language. This kind of an approach can be called a user oriented one because it describes what are effects of expressions—it does not define how the effects
are achieved. One weakness of such an approach is that the restructuring process of hierarchical data bases is not presented in such a way which would describe, in general, how the target data base is constructed from the source data base.

The approach of this paper may be characterized mainly as an implementation oriented one. Namely, the formal functions introduced below define in detail how each element in the target data base is constructed from the source data base. The goal of this paper is not to introduce a certain software product. We rather aim at offering to implementers such analysis and restructuring functions of hierarchical data bases which define the restructuring operations to be needed in the implementation. The formal functions define the tasks of the restructurer at run time.

The term restructuring user\(^{(21)}\) is used to refer to a person who specifies a data restructuring case from the source environment to the target environment. In a data base environment data restructuring is one of the tasks of a data administrator.\(^{(31)}\) So far, the transformation languages have provided the restructuring user the possibility to describe the restructuring cases as a sequence of explicit restructuring operations. Likewise, the theoretical research has classified different restructuring operations. For example, Navathe\(^{(20)}\) and Fry & Navathe\(^{(21)}\) have named different restructuring operations of hierarchical data bases at the schema and instance levels.

However, it is not necessary to develop such software for restructuring hierarchical data bases where the restructuring user must specify all operations to be needed. In the structural sense the minimal information, which is needed for restructuring, is the source data base and the target schema. We have this as a starting point for restructuring. This means that the RESTRUCTURER analyzes the restructuring case instead of a user and performs the needed operations based on this analysis. In this paper we define functions that analyze the structural properties of hierarchical data bases. Based on the information given by the analysis functions, we define the general restructuring functions which perform all transformations in hierarchical data bases without any explicit information on the required operations. In this paper we define a sequence of functions to construct the target data base from the source data base by analyzing the target schema. From the viewpoint of the restructuring user this approach has the benefit that he or she does not need to express the restructuring operations in his or her restructuring description. In other words, his or her description becomes more straightforward.

Dale and Dale\(^{(6,7)}\) use diagrams to describe different restructuring cases. Here, instead of diagrams and other modes of visualization we need a general, formal representation for a hierarchical data base: Such a represen-
2. THE REPRESENTATION FOR HIERARCHICAL DATA BASES

From the viewpoint of this paper we state the following requirements for describing hierarchical data bases.

- The structural properties in data structures must be described as clearly as possible.
- The representation of a hierarchical data base has to contain both the schema and instance level.
- It must be possible to define exact analysis and restructuring functions flexibly in the context of the representation.

Considering hierarchical data structures we can note that their substructures are hierarchical, too. This feature is typical of the hierarchical data structures and it is apparent in the concept "form" of CONVERTo and in the tabular visualization of unnormalized relations. In the structural sense, recursion is a natural way to define and manipulate hierarchical data structures. Therefore, we define and manipulate hierarchical data structures recursively, too.

The components needed in the recursive definition of a hierarchical data structure are called here generally data structure elements. A hierarchical data base itself is thus a data structure element. A data structure element can be either a relation presented as a six-tuple \((r, X, RN, DN, I_x, f_x)\) or a so-called zero element (denoted by \(A\)). Structurally the relation used in this paper differs from the relation permitted by the relational model in the respect that another relation can be as a domain of the relation. This kind of relations are called power set type relations, too. The zero element is necessary to make the representation mathematically well defined. In the six-tuple

\[
\begin{align*}
  r & \text{ is called an instance of the relation} \\
  X & \text{ is a relation type} \\
  RN & \text{ is a set of the record type names} \\
  DN & \text{ is a set of the data item names} \\
  I_x & \text{ is an index set} \\
  f_x : RN \cup DN \to I_x & \text{ is a naming function}
\end{align*}
\]

In the recursive definition of a data structure element (briefly \(dse\)) we need the concept of the same type \(dse\)’s. We also apply certain manipulation
functions in the index set \( I_x \). Therefore, the definition of the concept in question and the manipulation functions are introduced before the definition of the dse.

**Definition.** Let \( t_1, t_2, \ldots, t_n \) be some dse's such that \( t_i = (r_i, X_i, RN_i, DN_i, I_x(t_i), f_{x_i}) \), \( i = 1, \ldots, n \). If for any two dse's \( t_i, t_j \) (\( i \neq j \)) the following condition holds \( (X_i = X_j) \land (RN_i = RN_j) \land (DN_i = DN_j) \land (I_x(t_i) = I_x(t_j)) \land (f_{x_i} = f_{x_j}) \) then the dse's are of the same type.

The indices of our formalism are finite tuples consisting of natural numbers. The tuples are denoted between angle brackets, for example \( \langle 1, 2, 3 \rangle \). The denotation \( \langle i, \xi \rangle \) means a tuple where the first component is \( i \) and others form the tuple \( \xi \). Analogically, \( \langle \xi, i \rangle \) is a tuple whose last element is \( i \). Also, the concatenation of two tuples \( \xi_1 \) and \( \xi_2 \) is denoted by \( \langle \xi_1, \xi_2 \rangle \), for example, if \( \xi_1 = \langle a, b \rangle \) and \( \xi_2 = \langle c, d, e \rangle \) then, \( \langle \xi_1, \xi_2 \rangle = \langle a, b, c, d, e \rangle \).

2.1. The Manipulation Functions for Indices

Indices are elements of the set \( I_x \). When manipulating indices we need the following functions.

- The primitive function \( \text{elem}(\xi) \) gives the number of components of the index \( \xi \), for example \( \text{elem}(\langle 1, 5 \rangle) = 2 \).
- The primitive function \( p(\xi; n) \) gives the subindex with length \( n \mid 0 < n \leq \text{elem}(\xi) \) from the beginning of the index \( \xi \), for example \( p(\langle 1, 4, 1 \rangle, 2) = \langle 1, 4 \rangle \).
- The primitive function \( \text{sum}(\xi) \) expresses the sum of the components of the index \( \xi \), for example \( \text{sum}(\langle 1, 2, 3 \rangle) = 6 \).
- \( \max(I_x) = \max \{ |\xi| \mid \xi \in I_x \land \text{elem}(\xi) = 1 \} \) (we use \( |Z| \) to denote the cardinality of the set \( Z \)).
- The operations \( \oplus \) and \( \ominus \) are defined between the index set \( I_x \) and the nonnegative integer \( n \) as follows:
  \[
  I_x \oplus n = \{ (i + n, \xi) \mid (i, \xi) \in I_x \} \\
  I_x \ominus n = \{ (i - n, \xi) \mid (i, \xi) \in I_x \land i > n \}
  \]
  For example, if \( I_x = \{ \langle 1 \rangle, \langle 1, 1 \rangle, \langle 1, 1, 1 \rangle, \langle 2 \rangle \} \), then \( I_x \oplus \max(I_x) = \{ \langle 3 \rangle, \langle 3, 1 \rangle, \langle 3, 1, 1 \rangle, \langle 4 \rangle \} \).

2.2. The Definition of the Data Structure Element and Its Projection

We extend the definition of a dse represented in\(^{153}\) by giving definitions for a dse and its projection at the same time. When \( t = (r, X, RN, DN, \ldots) \)
$I_x, f_x$ is a dse and $I$ a set of indices then we use $t[I]$ to denote the projection of $t$ on indices $I$. This projection is analogical to the well known projection operation of the relational data model.

**Definition.** A dse is either a six-tuple $(r, X, RN, DN, I_x, f_x)$ or the zero element $(\lambda)$. It satisfies the following rules formed recursively and it is constructed by applying these rules finitely.

1. A data item is a dse $(r, D, \emptyset, \{A\}, \langle 1 \rangle, f_D(A) = \langle 1 \rangle)$ where
   - $r$ is a data item value with the property $r \in D$,
   - $D$ is some primitive type (domain). Here primitive types are integers (int), reals (real) and character strings (char),
   - $\emptyset$ means that this dse does not contain any record type name,
   - $\{A\}$ a set that consists of a data item name $A$,
   - the naming function $f_D : \{A\} \cup \emptyset \rightarrow I_x$ has form: $f_D(A) = \langle 1 \rangle$ i.e., it associates the index $\langle 1 \rangle$ with the data item name $A$.

   If $t$ is a data item and $I$ is a set of indices then the projection $t[I]$ is defined in the following way:

   $$t[I] = \begin{cases} t, & \text{if } \langle 1 \rangle \in I \\ A, & \text{otherwise} \end{cases}$$

2. Let $t_1 = (r_1, X_1, RN_1, DN_1, I_x, f_x)$ and $t_2 = (r_2, X_2, RN_2, DN_2, I_x, f_x)$ be two dse’s such that $(RN_1 \cup DN_1) \cap (RN_2 \cup DN_2) = \emptyset$, then the notation $\langle t_1, t_2 \rangle$ constructs the dse $\langle (r_1, r_2), X_1 \times X_2, RN_1 \cup RN_2, DN_1 \cup DN_2, I_x \times I_x, f_x \times f_x \rangle$ where

   $$I_x \times I_x = I_x \cup [I_x \otimes \max(I_x)]$$

   $$f_x : RN_1 \cup RN_2 \cup DN_1 \cup DN_2 \rightarrow I_x \times I_x$$

   $$f_x(a) = \begin{cases} f_x(a), & \text{if } a \in RN_1 \cup DN_1 \\ (f_x(a) \otimes \max(I_x)), & \text{if } a \in RN_2 \cup DN_2 \end{cases}$$

   The zero element behaves in this operation as follows:

   $$\langle \lambda, t \rangle = \langle t, \lambda \rangle = t$$ when $t$ is any dse.

   If $t$ is a dse such that $t = \langle t_1, t_2 \rangle$ then the projection of $t$ on indices $I$ is defined as follows:

   $$t[I] = \langle t_1[I], t_2[I] \otimes \max(I_x) \rangle.$$
(3) Let \( T = \{ t_i, X, RN, DN, I_x, f_x, s_x \} \) be a finite set consisting of dse's of the same type and \( A \in RN \cup DN \), then the notation \( A : T \) constructs the dse

\[
\left( \bigcup_{i} t_i, P(X), RN \cup \{ A \}, DN, I_{p(x)}, f_{p(x)} \right)
\]

where

\( P(X) \) is the power set of the relation type \( X \),

\[
I_p(x) = \{ \langle 1 \rangle \cup \{ 1, \xi \} \mid \xi \in I_x \},
\]

\[
f_p(x) = RN \cup \{ A \} \cup DN \rightarrow I_{p(x)}
\]

\[
f_{p(x)}(a) = \begin{cases} 
1 & \text{if } a \in A \\
1, f_x(a) & \text{if } a \in RN \cup DN
\end{cases}
\]

If \( t_i = A_i, i = 1, \ldots, n \) then \( A : \bigcup_{i} t_i = A \).

If \( t \) is a dse such that \( t = A : T \) then the projection on indices \( I \) is defined as follows

\[
t[I] = A : \bigcup_{i} t_i[I]
\]

where \( t_i \in T \) and \( I = \{ \xi \mid \langle 1, \xi \rangle \in I \} \).

A dse constructed by the second rule above has no equivalent term in the data base literature but it is possible to construct records by repeating it finitely. The third rule is applicable for constructing record groups. This operation is needed because there can be several records of the same type depending on one record in a hierarchical data base.

It has been shown\(^{(22)}\) that we can construct the formal representation for any hierarchical data base using the rules defined above. Now we have also defined recursively the projection of a hierarchical data base. In restructuring of hierarchical data bases we can separate parts of hierarchical data bases by applying projection. As a consequence of the construction rules described above the components of the six-tuple representation of any dse have the following properties:

1\(^o\) \( r \in X \)

2\(^o\) \( RN \cap DN = \emptyset \)

3\(^o\) \( |RN| + |DN| = |I_x| \)

4\(^o\) The naming function \( f_x : RN \cup DN \rightarrow I_x \) is bijective.

5\(^o\) The relation type \( X \) is a Cartesian product \( T_1 \times \cdots \times T_n \) where each \( T_i \) (\( 1 \leq i \leq n \)) can be either a primitive type (int, real, char) or the power set \( P(Y) \) of some other relation type \( Y \).
The elements of the index set $I_x$, which are associated with the relation type $X = T_1 \times \cdots \times T_n$, are $n$-tuples such that

$$\forall i \in \{1, \ldots, n\} : (i) \in I_x \iff (i, \xi) \in I_x,$$

$$\text{iff } T_i = P(Y) \text{ and } \xi \in I_x.$$

In fact, the properties 1$^0$–6$^0$ define a dse implicitly. Because a hierarchical data base itself is a dse it satisfies the properties described above. The relation type of a hierarchical data base has the form $P(Y)$, where $Y$ can be any relation i.e., data base trees in a hierarchical data base are type of $Y$. The components $RN$ and $DN$ in the formal description of any hierarchical data base contain all record type names and data item names.

Next we represent a hierarchical sample data base $i$ both as a typical tabular representation (Fig. 1) and as a dse formalized in this paper (Fig. 2). $X$ (the root record type), $C$, $G$, $J$ are record types and $A$, $B$, $D$, $E$, $F$, $H$, $I$, $K$, $P$ are data items in the sample data base. The dotted lines separate the data base trees.

In the following we present the operation sequence constructing the formal representation for the sample data base $i$ when a data item (the smallest logical data unit) $(r, D, \emptyset, \{A\}, \{(1)\}, f(A) = \{(1)\})$ is briefly denoted by $A_0^i$. Also, we shall use the notation $\langle t_1, t_2, \ldots, t_n \rangle$ to mean the following sequence of operations $\langle \cdots \langle t_1, t_2 \rangle, t_3 \cdots, t_n \rangle$.

$$i = X: \langle A_{t_1}^{\text{int}}, B_{t_1}^{\text{rel}}, \rangle,$$

$$C: \langle D_{t_1}^{\text{char}}, E_{t_1}^{\text{int}}, F_{t_1}^{\text{int}} \rangle,$$

$$G: \langle H_{d_2}^{\text{int}}, I_{d_2}^{\text{int}} \rangle \cup \langle H_{d_1}^{\text{int}}, I_{d_1}^{\text{int}} \rangle \cup \langle D_{d_3}^{\text{char}}, E_{d_3}^{\text{int}}, F_{d_3}^{\text{int}} \rangle,$$

$$G: \langle H_{d_2}^{\text{int}}, I_{d_2}^{\text{int}} \rangle,$$

$$J: \langle H_{d_2}^{\text{int}}, I_{d_2}^{\text{int}} \rangle \cup \langle H_{d_1}^{\text{int}}, I_{d_1}^{\text{int}} \rangle \cup \langle D_{d_3}^{\text{char}}, E_{d_3}^{\text{int}}, F_{d_3}^{\text{int}} \rangle,$$

$$J: \langle A_{d_1}^{\text{int}}, B_{d_1}^{\text{rel}} \rangle.$$
\[
\{(a_1, b_1, \{d_1, e_1, f_1\}, \{h_1, i_1, j_1, k_1, l_1, m_1, n_1, o_1, p_1, q_1, r_1, s_1, t_1, u_1, v_1, w_1, x_1, y_1, z_1\}, S_1, T_1, U_1, V_1, W_1, X_1, Y_1, Z_1\},
\{(a_2, b_2, \{d_2, e_2, f_2\}, \{h_2, i_2, j_2, k_2, l_2, m_2, n_2, o_2, p_2, q_2, r_2, s_2, t_2, u_2, v_2, w_2, x_2, y_2, z_2\}, S_2, T_2, U_2, V_2, W_2, X_2, Y_2, Z_2\} \}
\]

Fig. 2. The sample data base \(I\) as a dse.

We shall use the given sample data base to illustrate restructuring operations. We shall also apply the construction rules used in the definition of the dse in the definitions of the restructuring operations.

Together with a dse we defined its projection, too. The result of projection is always some dse. For example, projection \(I[\{I\}]\) of the sample data base \(I\) (a dse), when

\[
I = \{\{1, 1\}, \{1, 2\}, \{1, 3, 1\}, \{1, 3, 4, 2\}\}
\]

is the following

\[
dse = \{(a_1, b_1, \{d_1, \{i_1, j_1\}\}, d_2, \{i_2, j_2\}),
\quad(a_2, b_2, \{d_3, \{i_3, j_3\}\})\},
\]

\[
P(\text{int} \times \text{real} \times P(\text{char} \times P(\text{int}))), \{X, C, G\}, \{A, B, D, I\},
\]

\[
\{\{1, 1\}, \{1, 2\}, \{1, 3\}, \{1, 3, 1\}, \{1, 3, 2\}, \{1, 3, 4, 2\}\},
\]

\[
f(X) = \{1, 3\}, f(A) = \{1, 3\}, f(B) = \{1, 3\}, f(C) = \{1, 3\}, f(D) = \{1, 3\}, f(E) = \{1, 3\},
\]

\[
f(G) = \{1, 3\}, f(H) = \{1, 3\}, f(I) = \{1, 3\},
\]

We leave it up to the reader as an exercise to apply in detail the projection rules above which define the result.

The projection can also be expressed by data item names instead of indices because the naming function is bijective. Thus, the sample projection can be expressed in the form \(\bigcup_{n \in N} f_\xi(n)\) when \(N = \{A, B, D, I\}\).

3. THE DESCRIPTION OF THE TARGET SCHEMA AND THE DEFINITION OF THE ANALYSIS FUNCTIONS

Consider the components of the formal representation of any hierarchical data base. Intuitively, it is easy to note that the component \(r\)
describes the instance level whereas $X$, $RN$, $DN$, $I_x$, $f_x$ describe the schema level. All structural properties in the representation have been hidden behind indices. In addition, the component $X$ describes what kind of a mathematical concept a hierarchical data base represents.

The representation has been extended in Ref. 25 by the key component which expresses the keys of all record types in a hierarchical data base. Information on the keys of the source data base is not used in the definition of the restructuring functions. Therefore, the component in question is not included in the formal representation of the source data base.

In the data restructuring process a data base according to the target schema is constructed from the source data base. Here, we shall use in this process the keys of the record types included in the target schema. Therefore, in the formal description of the target schema there is a component which contains the keys of all record types.

The formal representation of the target schema is also a six-tuple. However, it differs from the description of a dse in that respect that instead of the relation instance there is a component that expresses the keys of the record types. All the other components are same as in a dse. The target schema is described as a six-tuple $(key, Y, RN, DN, I_y, f_y)$ where

- $Y$ (the relation type)
- $RN$ (a set of record type names)
- $DN$ (a set of data item names)
- $I_y$ (an index set)
- $f_y$ (a naming function)

are described in the same way as in the context of a dse. The key component has the property: $key \subseteq DN$. It is worth noting that the formal representation of the target schema is not a dse although it has five similar components.

The functions, which manipulate the schema level components in the formal representations of the target schema or of the dse, are called analysis functions. Those functions, which transform dse's into other dse's, are called restructuring functions. The analysis functions deliver information on the structural properties of a hierarchical data base. The restructuring functions need this information.

Next we define the analysis functions needed in the restructuring of the hierarchical data base. These functions can be applied to a dse and a target schema. In the definitions we use the inverse function of the naming function. The inverse function is denoted $f_y^{-1}: I_y \rightarrow RN \cup DN$.

In the hierarchical data base the leaf record types are those record types on which any record type depends neither immediately nor indirectly. The
function leaf\(I_x\) gives the indices associated with the leaf record types of the hierarchical data base.

**Definition.**

\[
\text{leaf}(I_x) = \{ \xi \mid \xi \in I_x \land (f_x^{-1}(\xi) \in RN) \land (\forall \langle \xi, i \rangle \in I_x : f_x^{-1}(\langle \xi, i \rangle) \in DN) \}
\]

In the sample data base \(\text{leaf}(I_x) = \{ \langle 1, 3, 4 \rangle, \langle 1, 4 \rangle \}\). The corresponding record type names are expressed by

\[
\bigcup_{i \in \text{leaf}(I_x)} f_x^{-1}(i) = \{ G, J \}
\]

The function \(r\text{-content}\) yields the indices of those data items that belong to the record type given as the argument. The argument \(\xi\) must satisfy the condition \(f_x^{-1}(\xi) \in RN\).

**Definition.**

\[
r\text{-content}(\xi) = \{ \langle \xi, i \rangle \mid (\langle \xi, i \rangle \in I_x) \land (f_x^{-1}(\langle \xi, i \rangle) \in DN) \}
\]

In the sample data base \(r\text{-content} \langle 1, 3 \rangle = \{ \langle 1, 3, 1 \rangle, \langle 1, 3, 2 \rangle, \langle 1, 3, 3 \rangle \}\), i.e. the record type \(C\) consists of data items \(D, E\) and \(F\).

The function predecessor obtains the index of that record type on which the record type as the argument depends immediately. The argument \(\langle \xi, i \rangle\) is required that \(f_x^{-1}(\langle \xi, i \rangle) \in RN\).

**Definition.**

\[
\text{predecessor}(\langle \xi, i \rangle) = \xi
\]

From the indexing mechanism it follows that always \(f_x^{-1}(\xi) \in RN\). In the sample data base \(\text{predecessor}(\langle 1, 3, 4 \rangle) = \langle 1, 3 \rangle\), i.e. the record type \(G\) depends immediately on the record type \(C\).

At the logical level the access path type (or access path schema) can informally be defined as a route to a record type. In the hierarchical data base the access path type to a certain record type can be defined by expressing those record types on which the record type depends immediately on indirectly. The function \(\alpha\text{-pathtype}\) expresses the access path type to the record type given as an argument. The argument \(\text{ind}\) of the function has the property \(f_x^{-1}(\text{ind}) \in RN\).
Definition.

\[
a\text{-pathtype}(\text{ind}) = \begin{cases} 
\text{ind} \cup a\text{-pathtype}(&\text{predecessor(\text{ind})}) \\
\text{ind} &\text{if } \text{elem(\text{ind})} > 1 \\
\text{ind} &\text{if } \text{elem(\text{ind})} = 1/\text{the record type with which the index } \langle 1 \rangle \text{ is associated}^*/
\end{cases}
\]

In the sample data base \(a\text{-pathtype}((1, 3, 4)) = \{\langle 1, 3, 4 \rangle, \langle 1, 3 \rangle, \langle 1 \rangle\}\), i.e. the access path type to the record type with index \(\langle 1, 3, 4 \rangle\) consists of the record types \(G, C\) and \(X\).

We need the function \(d\text{-collection (data item collection)}\) to give all data item names belonging to a certain access path type. It is required that \(f_x^{-1}(\text{ind}) \in RN\) (\(\text{ind}\) is the argument of the function).

Definition.

\[
d\text{-collection}(\text{ind}) = \bigcup_{i \in a\text{-pathtype}(\text{ind})} r\text{-content}(i)
\]

In the sample data base

\[
d\text{-collection}((1, 3, 4)) = \{\langle 1, 3, 4, 1 \rangle, \langle 1, 3, 4, 2 \rangle, \langle 1, 3, 1 \rangle, \\
\langle 1, 3, 2 \rangle, \langle 1, 3, 3 \rangle, \langle 1, 1 \rangle, \langle 1, 2 \rangle\}
\]

This set expresses the indices of those data items which belong to the access path type leading to the record type \(G\).

The function \(\text{parpath}(\text{ind}_1, \text{ind}_2)\) gives the set of indices of those data items which belong to that part of the access path type delimited by the indices \(\text{ind}_1\) and \(\text{ind}_2\). The arguments of the function must satisfy the condition \(\text{ind}_1 \in a\text{-pathtype}(\text{ind}_2)\).

Definition.

\[
\text{parpath}(\text{ind}_1, \text{ind}_2) = \begin{cases} 
\bigcup_{i \in r\text{-content}(\text{ind}_1)} f_x^{-1}(i) \cup \text{parpath}(\text{ind}_1, \text{predecessor}(\text{ind}_2)) \\
\bigcup_{i \in r\text{-content}(\text{ind}_2)} &\text{if } \text{ind}_1 \neq \text{ind}_2 \\
\bigcup_{i \in r\text{-content}(\text{ind}_2)} &\text{if } \text{ind}_1 = \text{ind}_2
\end{cases}
\]

In the sample data base \(\text{parpath}((1, 3), \langle 1, 3, 4 \rangle) = \{\langle 1, 3, 4, 1 \rangle, \langle 1, 3, 4, 2 \rangle, \\
\langle 1, 3, 1 \rangle, \langle 1, 3, 2 \rangle, \langle 1, 3, 3 \rangle\}\).
The function following\((\text{ind}_1, \text{ind}_2)\) gives the index of that record type which has the following properties:

1. The record type in question depends immediately on the record type with the index \(\text{ind}_1\).
2. The record type in question belongs to the access path type leading to the record type identified by the index \(\text{ind}_2\).

The arguments of the function are required that \(\text{ind}_1 \neq \text{ind}_2\) and \(\text{ind}_1 \in \alpha\text{-pathtype}(\text{ind}_2)\).

**Definition.**

\[
\text{following}(\text{ind}_1, \text{ind}_2) = \langle \xi, i \rangle \text{ when } \langle \xi, i \rangle \in \alpha\text{-pathtype}(\text{ind}_2) \land \xi = \text{ind}_1
\]

In the sample data base following \((\langle 1 \rangle, \langle 1, 3, 4 \rangle) = \langle 1, 3 \rangle\).

In the definitions of the restructuring functions we shall need the function \(\text{next}\) which returns the indices associated with the leaf record types in a certain order. In the definition of the function \(\text{next}\) we use the function \(i\text{-subset}(A, i)(A \subseteq \text{leaf}(I_x))\). This function gives such indices from the set \(A\) that the sum of the \(i\) first components of these indices is minimal. The argument \(i\) is a positive integer.

**Definition.**

\[i\text{-subset}(A, i) = \{\xi \mid \xi \in A \land \forall z \in A: \text{sum}(p(z, i)) < \text{sum}(p(\xi, i))\}\]

The primitive functions \(\text{sum}\) and \(p\) have been defined above. Now, the function \(\text{next}\) can be defined as follows.

**Definition.**

\[\text{next}(A, i) = \begin{cases} \text{next}(i\text{-subset}(A, i), i + 1), & \text{if } |i\text{-subset}(A, i)| > 1 \\ x & \text{when } x \in i\text{-subset}(A, i), \text{ if } |i\text{-subset}(A, i)| = 1 \end{cases}\]

It follows from the indexing mechanism of the hierarchical data base that the function \(\text{next}\) always makes the set \(A\) smaller so that only one index belongs to it finally. In the sample data base the function \(\text{next}(\text{leaf}(I_x), 1)\) gives the index \(\langle 1, 3, 4 \rangle\) and the function \(\text{next}(\text{leaf}(I_x)\text{-next}(\text{leaf}(I_x), 1), 1)\) gives the index \(\langle 1, 4 \rangle\).

It is typical of the analysis functions defined above that they can be applied to the formal representations of both a dse and a target schema.
we define the function \textit{recordkey} which can be applied only to the target schema. It obtains the key of a record type in the target schema. A key may consist of one or more data items. Let \((\text{key}, \text{Y}, \text{RN}, \text{DN}, I_r, f_r)\) be any target schema. The function recordkey is defined in the following way.

\textbf{Definition.}

\[
\text{recordkey}(\xi) = \{ f_r^{-1}(\langle \xi, i \rangle) \mid \langle \xi, i \rangle \in I_r \land f_r^{-1}(\langle \xi, i \rangle) \in \text{key} \}
\]

The argument \(\xi\) is supposed to satisfy \(f_r^{-1}(\xi) \in \text{RN}\). We shall use the following sample target schema to illustrate the definitions of restructuring functions. (See Fig. 3.) In this target schema \(\text{recordkey}(\langle 1, 3 \rangle) = \{D\} \).

\section{EXACT DEFINITION OF THE RESTRUCTURING FUNCTIONS}

So far, we have defined analysis functions to analyze the structural properties of hierarchical data structures. Restructuring functions are needed to define the transformation process of hierarchical data bases at the instance level. In this paper we aim at defining the transformation process generally. Therefore, we need such an internal representation of a hierarchical data base from which it is possible to construct flexibly any data base derivable from this internal representation.

We can state the following requirements for the internal representation of a hierarchical data base:

\begin{enumerate}
  \item Any information essential in the structural sense must not disappear when transforming a hierarchical data base into its internal representation. This means that:
    \begin{itemize}
      \item A data base and its internal representation must have the same data item content.
      \item The internal representation must preserve the logical relationships among data.
    \end{itemize}
    These requirements guarantee that the original source data base can always when required be constructed from the internal representation.
  \item The internal representation must not contain any hierarchy. This is an essential requirement because it must be possible to construct
different hierarchical data bases with a different number of hierarchy levels and different hierarchical relationships from the same internal representation.

In restructuring a hierarchical data base we can separate two main phases. The first main phase transforms a hierarchical data base into its internal representation. The second main phase constructs the target data base according to the target schema from this internal representation. In this chapter we consider the functions used in these phases.

4.1. The Transformation of a Hierarchical Data Base into Its Internal Representation

In fact, we have already defined one restructuring function, the projection. Using this operation we can separate any substructure from a dse. Intuitively, in the definition of the transformation of a hierarchical data base into its internal representation we can separate the following steps.

1. Simple (unbranching) hierarchical structures are separated from the source data base. Each simple hierarchical data structure consists of the access path type leading to one leaf record type.

2. The simple hierarchical structures are transformed into non-hierarchical structures.

3. The nonhierarchical structures are joined with each other.

In the definitions of restructuring functions we often need the inverse operation for the construction operation $A : T$ (see the rule three in the definition of a dse). This inverse operation is denoted by $\sim$.

Definition. Let $d$ be a dse such that $d = A : T$ then $\sim d = T$.

In the sample data base $i$ the operation $\sim i = \{(a_1, b_1, \langle d_1, e_1, f_1, \langle h_1, i_1 \rangle \rangle), \langle d_2, e_2, f_2, \langle h_2, i_2 \rangle \rangle\}, \langle k_1, p_1, \langle k_2, p_2 \rangle \rangle, \langle k_2, p_2 \rangle \rangle, \langle k_3, p_3 \rangle \rangle\}$ (only the first components of the dse's are given). In this case the set $\sim i$ contains two data base trees of the data base.

Next we define the concatenation operation $(\cdot)$ which is needed in deleting the hierarchy of a hierarchical data base.

Definition. Let $t_1 = (r_1, X_1, RN_1, DN_1, I_{x_1}, f_{x_1})$ and $t_2 = (r_2, X_2, RN_2, DN_2, I_{x_2}, f_{x_2})$ be two dse's such that $(RN_1 \cup DN_1) \cap (RN_2 \cup DN_2) = \emptyset$. Then the concatenation
\[
\begin{align*}
t_1 \sim t_2 = \begin{cases} 
\langle t_1, t_2 \rangle, & \text{if } X_2 \text{ is not type of } P(Y) \text{ where } Y \\
\bigcup_{i \in t_1 \sim t_2} \langle i_1, i_2 \rangle, & \text{if } X_2 \text{ is type of } P(Y)
\end{cases}
\end{align*}
\]

The function \(l\)-compression (level compression) constructs a single level from the root record type and the record type immediately depending on it. A simple hierarchical data structure \(t\) is the argument of this function, i.e. \(t = (r, \quad X, \quad RN, \quad DN, \quad I_x, \quad f_x)\) has the property \(|\text{leaf}(I_x)| = 1\).

**Definition.**

\(l\)-compression\((t)\)

\[= f^{-1}_X(1): \bigcup_{k \in I_k} [\xi | \xi \in I_k \land \text{elem}(\xi) = 1 \land f^{-1}_k(\xi) \in DN] \sim \]

\[\forall k | \xi \in I_k \land \text{elem}(\xi) > 1 \land f^{-1}_k(\xi) \in DN\]

where \(I_k\), \(f_k\) denote the index set and the naming function associated with the dse \(k\).

We need this function in the recursive definition of the function \textit{flat}.

The function \textit{flat} constructs a nonhierarchical structure from a simple hierarchical data structure \(t = (r, \quad X, \quad RN, \quad DN, \quad I_x, \quad f_x)\). It is obvious that a dse is without hierarchy when \(|RN| = 1\).

**Definition.**

\textit{flat}\((t)\) = \[
\begin{cases} 
\text{flat}(l\text{-compression}(t)), & \text{if } |RN| > 1 \\
t, & \text{if } |RN| = 1
\end{cases}
\]

We have discussed above that in the transformation process the nonhierarchical structures must be joined with each other. In this we use the function \textit{join}, which has two dse's \(t_1 = (r_1, \quad X_1, \quad RN_1, \quad DN_1, \quad I_{x_1}, \quad f_{x_1})\) and \(t_2 = (r_2, \quad X_2, \quad RN_2, \quad DN_2, \quad I_{x_2}, \quad f_{x_2})\) as the arguments. Both arguments must be nonhierarchical, i.e. \(|RN_1| = |RN_2| = 1\).

**Definition.**

\textit{join}\((t_1, \quad t_2)\)

\[= f^{-1}_{x_1}(1): \{ (k_1, k_2 \quad \bigcup_{a \in DN_1 \cap DN_2} f_{x_1}(a)) \mid k_1 \in \sim t_1 \land k_2 \in \sim t_2 \land \forall b \in DN_1 \cap DN_2 : k_1[f_{x_1}(b)] = k_2[f_{x_2}(b)] \}\]
where the naming functions associated with the dse's $k_1$ and $k_2$ are denoted by $f_{k_1}$ and $f_{k_2}$.

The function normalization transforms any hierarchical data base $t = (r, X, RN, DN, I_x, f_x)$ into its internal representation. The other argument of the function is a set $A$, which consists of indices of leaf record types of $t$, i.e. $A \subseteq \text{leaf}(I_x)$.

**Definition.**

$$
\text{normalization}(t, A) = \begin{cases} 
\text{join}(\text{flat}(t[\text{d-collection}(\text{next}(A))]), \\
\text{normalization}(t, A\text{-next}(A))) 
, & \text{if } |A| > 1 \\
\text{flat}(t[\text{d-collection}(\text{next}(A))]), \\
, & \text{if } |A| = 1 
\end{cases}
$$

The analysis functions $d$-collection and next have been defined above. Thus, the function $\text{normalization}(t, \text{leaf}(I_x))$ defines the internal nonhierarchical representation for a data base $t$.

In Fig. 4, we illustrate how the sample data base $t$ is transformed into its internal representation. When we evaluate the functions above in detail we get the following dse

\[
\begin{align*}
\langle a_1, b_1, c_1, d_1, e_1, f_1, g_1, h_1, i_1, k_1, p_1 \rangle, \\
\langle a_2, b_2, c_2, d_2, e_2, f_2, g_2, h_2, i_2, k_2, p_2 \rangle, \\
\langle a_3, b_3, c_3, d_3, e_3, f_3, g_3, h_3, i_3, k_3, p_3 \rangle,
\end{align*}
\]

$P(\text{int} \times \text{real} \times \text{char} \times \text{int} \times \text{int} \times \text{int} \times \text{int} \times \text{int} \times \text{int} \times \text{int} \times \text{char})$,

$\{X\}, \{A, B, D, E, F, H, I, K, P\}$. 
\{\langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 1, 5 \rangle, \langle 1, 6 \rangle, \langle 1, 7 \rangle, \langle 1, 8 \rangle, \langle 1, 9 \rangle\).

\{(X) = \langle 1 \rangle, f(Y) = \langle 1, 1 \rangle, f(B) = \langle 1, 2 \rangle, f(D) = \langle 1, 3 \rangle, f(E) = \langle 1, 4 \rangle,
\quad f(F) = \langle 1, 5 \rangle, f(H) = \langle 1, 6 \rangle, f(I) = \langle 1, 7 \rangle, f(K) = \langle 1, 8 \rangle, f(P) = \langle 1, 9 \rangle\}\)

We denote this internal representation of the sample data base by \(\tilde{\mathcal{R}}\).

It is typical of the internal representation of a hierarchical data base that \(|RN| = 1\) and the relation type has the form \(P(Y)\), where all factors in the Cartesian product \(Y\) are some primitive types.

4.2. The Functions for Constructing Target Data Bases

Data items are the smallest logical data units of a hierarchical data base. Using the same data item names both in the source data base and in the target schema, we express the correspondence of these logical elementary units. Indices (associated with data items) cannot be used because the structure has been hidden behind indices, and in the structural sense, data items can lie in quite different positions in source and target data bases. Renaming of data items is not essential in the structural sense but it would require more manipulations in the formalism. Therefore, renaming of data items is not applied in this paper. Of course, the record type names may be different in the source and target data bases because their data item contents may be entirely different.

The second main phase in restructuring is to construct a hierarchical data base according to the target schema from the internal representation. The intuitive definition of the second main phase happens as follows.

1. We create simple (unbranching) hierarchical structures by analysing the target schema. Each simple hierarchical structure consists of an access path type leading to a certain leaf record type in the target schema.

2. Simple hierarchical structures constructed in this way are joined to the general (branching) hierarchical structure in the order given by the function next.

In transforming a hierarchical data base into its internal representation we eliminated hierarchy levels one by one with the function \(l\)-compression. Using the function \(h\)-form (hierarchy form), we constructed one hierarchy level of the target data base. Intuitively, \(h\)-form is the inverse operation of \(l\)-
compression. The function $h$-form has as its arguments two dse's $t_1 = (r_1, X_1, RN_{x_1}, DN_{x_1}, I_{x_1}, f_{x_1})$ and $t_2 = (r_2, X_2, RN_{x_2}, DN_{x_2}, I_{x_2}, f_{x_2})$ and two names $A$ and $B$. It must hold for $t_1$ and $t_2$ that their relation types have form $P(Y)$ and that $DN_{x_1} \cap DN_{x_2} \neq \emptyset$. When we apply the function $h$-form then $A$ and $B$ are record type names derived from the target schema.

**Definition.**

$$h\text{-form}(t_1, t_2, A, B) = A : \langle k_1, B ; S \rangle | \{ k_1 \in \sim t_1 \wedge k_2 \in \sim t_2 \wedge$$

$$S = \left\{ k_2 \left[ \bigcup_{a \in DN_{x_2} \cap DN_{y_2}} f_{k_2}(a) \right] \right\} | \left\{ k_1 \left[ \bigcup_{a \in DN_{x_1} \cap DN_{y_1}} f_{k_1}(a) \right] \right\}$$

where $f_{k_1}$ and $f_{k_2}$ are the naming functions associated with dse's $k_1$ and $k_2$.

We removed the hierarchy from the simple hierarchical data structure by the function flat. The function $s$-unnormialization (unnormialization of simple hierarchy) is intutively the inverse operation for this function. Namely, the function $s$-unnormialization constructs a simple hierarchical structure from the internal representation. The function $s$-unnormialization analyzes, which record types in the target schema form the access path type to the leaf record type with index ind, (an argument of the function), and constructs instances according to this structure from the internal representation.

The function $s$-unnormialization has among others as its arguments a data base in the internal representation $t = (r, X, RN_x, DN_x, I_x, f_x)$ and a target schema $s = (key, Y, RN_y, DN_y, I_y, f_y)$. Because the analysis functions except the function recordkey can be applied in both six-tuples we shall use $t$ and $s$ indices to functions to express whether they are applied to $s$ or $t$. Thus, for example $r$-content$(s)$(ind) means that the function is applied in the target schema.

The three other arguments of the function are: $k$ (a set of keys), ind, (an index which is included in the access path type leading to the leaf record type with index ind), and ind, (the index of a leaf record type in the target schema). In other words, the arguments have the following properties: $k \subseteq key$, ind, $\subseteq leaf(I_x)$, ind, $\subseteq a$-path(type(ind)).
Definition.

\[ s\text{-unnormalization}(t, s, k, \text{ind}_1, \text{ind}_2) \]

\[
= \begin{cases} 
\text{h-form} \left( t \left( \bigcup_{b \in B} f_s(b) \right) \right), \\
\text{s-unnormalization} \left( t \left( \bigcup_{a \in A} f_s(a) \right) \right), \\
\text{recordkey}(\text{ind}_1), \text{following}_i(\text{ind}_1, \text{ind}_2), \text{ind}_2), f_s^{-1}(\{1\}), \\
\text{f}_s^{-1}(\text{following}_i(\text{ind}_1, \text{ind}_2))), \\
\end{cases}
\]

where \( B = \bigcup_{i \in \text{r-content}, (\text{ind}_1)} f_s^{-1}(i) \cup k \), \( A = k \cup \text{recordkey}(\text{ind}_1) \cup \text{partpath}_i(\text{following}_i(\text{ind}_1, \text{ind}_2), \text{ind}_2); \)

if \( \text{ind}_1 \neq \text{ind}_2 \),

if \( \text{ind}_1 = \text{ind}_2 \),

When \( \text{ind}_2 \) is the index associated with a leaf record type then the function \( s\text{-unnormalization}(t, s, \emptyset, (1), \text{ind}_2) \) constructs the simple hierarchical structure from the internal representation \( t \). Consider the sample evaluation of the function \( s\text{-unnormalization} \) when we have the internal representation \( t \) and the sample schema \( \bar{s} \). The sample schema describes a simple hierarchy.

In the sample evaluation below we describe only the relation instance component from dse's.

\[ s\text{-unnormalization}(t, \bar{s}, \emptyset, (1), (1, 3)) \]

\[
= \begin{cases} 
\text{h-form}((\langle h_1, i_1 \rangle, \langle h_2, i_2 \rangle), \\
\langle h_1, i_1 \rangle, \ldots), \\
\text{s-unnormalization}(((\langle h_1, d_1, e_1, f_1 \rangle, \\
\langle h_2, d_1, e_1, f_1 \rangle, \langle h_1, d_2, e_2, f_2 \rangle, \\
\langle h_2, d_1, e_2, f_2 \rangle, \langle h_3, d_1, e_1, f_1 \rangle, \ldots), \\
\bar{s}, \langle H, (1, 3), (1, 2) \rangle, G, C) \\
\end{cases}
\]

\[
= (\langle h_1, d_1, e_1, f_1 \rangle), \langle h_2, d_1, e_1, f_1 \rangle, \\
\langle h_1, d_2, e_2, f_2 \rangle, \langle h_3, d_1, e_1, f_1 \rangle, \\
\langle h_1, d_3, e_3, f_3 \rangle, \ldots)
\]

\[
(\langle h_1, i_1 \rangle, \langle d_1, e_1, f_1 \rangle, \langle d_2, e_2, f_2 \rangle), \langle h_1, i_2 \rangle, \langle d_1, e_1, f_1 \rangle, \langle d_3, e_3, f_3 \rangle, \\
\langle h_1, i_2 \rangle, \langle d_2, e_3, f_3 \rangle, \ldots), \quad P(\text{int} \times \text{int} \times P(\text{char} \times \text{int} \times \text{int})), \quad (G, C), \\
\langle H, I, D, E, F \rangle, \quad (\{1\}, \{1, 1\}, \{1, 2\}, \{1, 3\}, \{1, 3, 1\}, \{1, 3, 2\}, \{1, 3, 3\}, \{ f(G) = (1) \}, \ f(H) = (1, 1) \), \ f(I) = (1, 2) \), \ f(C) = (1, 3) \), \ f(D) = (1, 3, 1), \ f(E) = (1, 3, 2) \), \ f(F) = (1, 3, 3) \)
In this paper we define generally the restructuring process of hierarchical data bases. Therefore, we need a function which is able to construct a general (branching) hierarchical data base from the internal representation. The definition of this function can be illustrated in the context of the following graph shown in Fig. 5. Nodes in the graph refer to record types of the target schema. The sample target schema $s$ is not suitable for this purpose because it contains the simple hierarchy only.

We can see the definition of the construction of the target data base to consist of the steps in Fig. 6. We suppose, of course, that it is possible to construct the target data base according to Fig. 5 from the internal representation. The numbers refer to the order in which the hierarchical data structures are constructed from the internal representation.
The structures (1), (2), (4), and (6) contain simple hierarchies whereas the structures (3), (5) and (7) are general (branching) hierarchies. The principle described above presupposes the definition of such function which combines two dse's, one of which can contain a general hierarchy and the other contains always a simple one. For this purpose we define the function \( h \)-join which has two dse's \( t_1 = (r_1, X_1, RN_1, DN_1, I_{x_1}, f_{x_1}) \) and \( t_2 = (r_2, X_2, RN_2, DN_2, I_{x_2}, f_{x_2}) \) as its arguments. The argument \( t_1 \) describes a general hierarchy, i.e. \( |\text{leaf}(I_{x_1})| \geq 1 \) and \( t_2 \) a simple one, i.e. \( |\text{leaf}(I_{x_2})| = 1 \). The function \( h \)-join associates those and only those parts of a simple dse \( t_2 \) which are not included in the dse \( t_1 \).

**Definition.**

\[
h\text{-}join(t_1, t_2)
\]

\[
\begin{align*}
(f^{-1}_{x_1}((1))): & \quad h\text{-}join(k_1, k_2) | k_1 \in \neg t_1 \land k_2 \in \neg t_2 \land k_1 \bigcup_{a \in A} f_{k_1}(a) \\
= & \quad k_2 \bigcup_{a \in A} f_{k_2}(a)
\end{align*}
\]

where \( f_{k_1} \) and \( f_{k_2} \) are the naming functions with dse's \( k_1 \) and \( k_2 \) and

\[
A = \{f^{-1}_{x_1}(i) | i \in r\text{-}content((1))|; \quad \text{if } f^{-1}_{x_1}((1)) \in RN_1\}
\]

\[
\langle t_1[A], h\text{-}join(t_1[B], t_2[C]) \rangle
\]

where

\[
A = \{a | a \in I_{x_1} \land \text{elem}(a) = 1 \land f_{x_1}(a) \in DN_1 \lor \}
\]

\[
\cap \{(i, \xi) | (i, \xi) \in I_{x_1} \land f^{-1}_{x_1}(i, \xi) \in DN_1 \land i \neq \max(I_{x_1})\}
\]

\[
B = \{(i, \xi) | (i, \xi) \in I_{x_1} \land f^{-1}_{x_1}(i, \xi) \in DN_1 \land i = \max(I_{x_1})\}
\]

\[
C = \{a | a \in I_{x_2} \land \text{elem}(a) > 1 \land f^{-1}_{x_2}(a) \in DN_2\};
\]

\[
\text{if } f^{-1}_{x_2}((1)) \in DN_1 \land RN_1 \cap RN_2 \neq \emptyset
\]

\[
\langle t_1[A], t_2[C] \rangle
\]

where

\[
A = \bigcup_{i \in DN_2 \land RN_2 \cap RN_1} f^{-1}_{x_2}(i);
\]

\[
\text{if } f^{-1}_{x_2}((1)) \in DN_1 \land RN_1 \cap RN_2 = \emptyset
\]
Next we define the function $g$-unnormaization (general unnormalization) for constructing a general hierarchy according to the target schema. This function has four arguments:

- A hierarchical data base $t_1 = (r_1, X_1, RN_1, DN_1, I_{x_1}, f_{x_1})$ which has been constructed from $t_2$,
- A data base in its internal representation $t_2 = (r_2, X_2, RN_2, DN_2, I_{x_2}, f_{x_2})$,
- A target schema $s = (\text{key}, Y, RN_y, DN_y, I_y, f_y)$,
- A set $A$ which contains indices of leaf record types of the target schema.

The following conditions are met by the arguments: $RN_1 \subseteq RN_y$, $DN_1 \subseteq DN_y$, $DN_1 \subseteq DN_2$, $DN_1 \subseteq DN_y$, $DN_1 \subseteq DN_2$, and $A \subseteq \text{leaf}(I_y)$.

**Definition.**

$g$-unnormaization$(t_1, t_2, s, A) = \begin{cases} 
\text{g-unnormaization}(\text{h-join}(t_1, s-\text{unnormaization}(t_2, s, A)), 1), & \text{if } A \neq \emptyset \\
\emptyset, & \text{if } A = \emptyset
\end{cases}$

Let $t = (r, X, RN, DN, I_x, f_x)$ be some hierarchical data base in the internal representation and $s = (\text{key}, Y, RN_y, DN_y, I_y, f_y)$ any target schema such that $DN_y \subseteq DN$. The construction of the target data base can now be defined by the function $\text{databaseformer}$ as follows.

**Definition.**

$\text{databaseformer}(t, s) = g$-unnormaization$(s-\text{unnormaization}((t, s, \emptyset, (1), \text{next}(\text{leaf}(I_y), 1)), t, s, \text{leaf}(I_y), \text{next}(\text{leaf}(I_y), 1)))$

Now we have defined such a sequence of functions which can restructure hierarchical data bases. Let $t$ be a hierarchical data base and $s$ a hierarchical target schema such that a data base according to $s$ is derivable from $t$. Thus, the function $\text{databaseformer}(\text{normalization}(t), s)$ defines all necessary restructuring operations for constructing the target data base from the source data base $t$. The construction of the target data base according to the sample target schema $s$ from the sample data base $l$ has been presented above in
The source data base \( \tilde{\mathcal{C}} \)

\[
\begin{array}{cccccccc}
A & B & C & \{D, E, F, G \} & J & \{K, P \} \\
\hline
a_1 & b_1 & d_1 & e_1 & f_1 & h_1 & i_1 & k_1 & p_1 \\
& & d_2 & e_2 & f_2 & h_2 & i_2 & k_2 & p_2 \\
a_2 & b_2 & d_3 & e_3 & f_3 & h_3 & i_3 & \\
& & & & & h_2 & i_2 & k_3 & p_3 \\
\end{array}
\]

was transformed into the following target data base

\[
\tilde{\mathcal{C}} \quad \{H, I \} \quad \tilde{\mathcal{C}} \quad \{D, E, F \}
\]

\[
\begin{array}{cccc}
\hline
h_1 & i_1 & d_1 & e_1 & f_1 \\
& & d_2 & e_2 & f_2 \\
h_2 & i_2 & d_1 & e_1 & f_1 \\
& & d_3 & e_3 & f_3 \\
h_3 & i_3 & d_3 & e_3 & f_3 \\
\end{array}
\]

Fig. 7. Summing up the restructuring case used in sample evaluations of functions.

detail in the context of sample evaluations. Thus sample transformation performs the following restructuring case described in the tabular form. (See Fig. 7.)

5. CONCLUSIONS

The non-procedural languages developed for restructuring users contain a set of explicit restructuring operations. In the structural sense, the minimal information, which must be given for restructuring data structures, is the source data base and the target schema. This means that we must build up restructuring software which is able to deduce the necessary operations by analyzing the source and target data structures. Software built up with this principle makes the description task of the restructuring user easier than in that alternative where he or she must design and describe the required operations. In this paper we have created a sequence of analysis and restructuring functions which define the restructuring of a hierarchical data base generally.

The formal functions in this paper define in detail how the target data base is constructed from the source data base. Therefore, this kind of an approach may be called an implementation oriented one. The restructuring of
Formal Restructuring Functions for Hierarchical Data Bases

Hierarchical data bases may produce records, that are quite different from those in the source data base. Likewise, in one restructuring case there may be many expansions, compressions or inversions of hierarchy levels. At the instance level these changes mean regrouping, eliminating or duplicating some data item values. The general restructuring functions of this paper define the total effect of the required restructuring operations at run time. We have defined the restructuring software precisely for hierarchical data bases.\(^{15,23}\) Knuth's attribute method\(^{17}\) was used in the definition and the functions described above are used to define the attributes.

The six-tuple used in the description of hierarchical data bases is general, i.e. it permits any number of record types and of hierarchy levels. Likewise, this representation is not bound to any hierarchical dbms. The formal representation of a hierarchical data base contains both the schema and instance levels. The operations used in the definition of a dse can also be applied for constructing data structures based on the flat file data model.\(^{24}\) Also, relations of the relation model can be described as analogical six-tuples which, of course, do not contain any hierarchy. In Ref.\ 26 we define a relational algebra based on an analogical six-tuple representation. In that the relational operations manipulate relations and their schemas at the same time.

REFERENCES


16. K. Lewis Kendall and J. P. Fry, A Comparison of Tree Translation Definition Languages, Working paper D1.5.1, Data Translation project University of Michigan (1976).


24. T. Niemi, A Formal Description of Data Structures Based on Flat File and Hierarchical Data Models, in Proc. The First Scandinavian Research Seminar on Data Modelling and Data Management, Tampere (January 1982).


PART III
Specification of Data Restructuring Software Based on the Attribute Method

Timo Niemi

Received April 1984; revised September 1984

The idea that abstract software specification is an essential phase in developing large and complex software has been widely accepted. In this paper, we specify in an abstract, but precise way, software for restructuring data structures based on the flat file and hierarchical data models. Our specification contains also the case that a target data structure is constructed from many source data structures. In data restructuring, data structures are transformed. We propose the use of the attribute method for these kinds of translation oriented specification situations in the data base area. We apply the attribute method in the context of abstract syntax instead of a concrete one.

KEY WORDS: Data base restructuring software; abstract (formal) software specification; attribute grammars.

1 INTRODUCTION

It has been widely recognized that data restructuring is a large and important part in data conversion (see Refs. 1 and 2). In this paper we specify software that is able to restructure data structures based on flat file or hierarchical data models. This work is associated closely with our previous work, where we define a sequence of functions in terms of which it is possible to restructure any hierarchical data base. However, in this work

1 University of Tampere, Department of Mathematical Sciences Computer Science, P.O. Box 607, SF-33101 Tampere 10, Finland.
we make a considerable extension of the previous contribution. Here, we specify software that constructs a target file or database with the desired structural properties from a set of existing files or databases (source data structures). Both the target and source data structures must be based either on the flat file data model or on the hierarchical data model.

Consider briefly some practical situations where this kind of software is useful. Let us suppose that we take into use a data base management system (DBMS) based on the hierarchical data model in an environment where all source data has been stored in conventional files (organized according to the flat file data model). In this case we may have to construct each hierarchical target data base from many source files.

In a data base environment the data administrator is responsible for data restructuring.23 Often new applications require that existing relationships among data must be restructured. In this kind of a restructuring case, the data administrator must often combine many hierarchical data bases or their parts into one target data base where the data have been organized in a new way.

Sometimes we have to convert hierarchical data into flat files, for example in certain data transferring cases of a computer network. Let us suppose the following situation. We have two nodes A (which is able to manipulate only flat files) and B (data organized hierarchically) in the computer network. When we need in A data stored in B we must restructure hierarchical data into a form that is nonhierarchical. This process may require collecting data from many hierarchy levels.

It has been observed in diverse contexts that the specification methods developed in programming language research have many important consequences for the data base area, too. One of the goals of abstract (formal) specification methods is to provide a tool for managing complexity when developing software. This is a very important viewpoint because the data base applications are large and complex in nature. It is obvious that the use of imprecise tools makes it difficult to manage complexity whereas suitable precise tools facilitate it. Due to these reasons many authors believe that abstract (or formal) software specification is an essential phase in software development.

Abstract software specification is a model of the software architecture. We can think that abstract software specification is some kind of a prototype of a software. In other words, abstract software specification defines precisely, beforehand, what we expect the software to do. We can test, experiment and possibly change it several times before the final and expensive implementation. Abstract software specification affords also the possibility of defining the software independent of implementation details. Namely this kind of a specification leaves the implementer free to optimize
the aspects concerning the effectiveness of software. Therefore abstract
software specification is sometimes called "abstract implementation" or
"abstract modeling technique."

It is also typical of abstract software specification that it offers an
abstraction on the behavior of software at run time. This is much more
than a pure input-output mapping performed by a software because
abstract software specification defines also precisely, and generally, those
data processing tasks and their interactions that are required to achieve the
result of the software.

Abstract software specification is also a tool for documentation of
software because it precisely defines the expressive power of our software. If
the semantics of the data languages are unclear then they are difficult to
understand and use. At Very Large Data Base Conference in Mexico City a
panel on "Type Specifications and Data Bases" was organized. The summary
report of this panel emphasized the importance of precise
specifications in the data base area.\(^\text{4}^\)

Many specification methods borrowed from programming languages
have been applied in the data base area, too. For example, specifications
concerning the relational data model have been presented using both the
Vienna Development Method (VDM)\(^\text{5,6}^\) and abstract data types.\(^\text{7,8}^\)
Relational query languages and relational data base updating are typical
specification issues. However, it is typical of these specification subjects that
they manipulate a data base of a given type but they do not transform the
actual structural properties of the data base. It has been widely recognized
(see e.g., Refs. 9 and 10) that we need complementary specification
approaches to formalize the various aspects of data base applications. In
this paper we propose the use of the well known attribute method\(^\text{11}^\) for
translation oriented specification issues. Our specification issue is translation
oriented. Very often data conversion is even called data translation.

Attribute grammars have been used for many purposes (e.g. defining
the semantics of programming languages, defining translators, and as tool
in optimization and correctness verification of programs). So far very little
of the attribute method has been applied in the data base area. Rid-
janovic and Brodie\(^\text{12}^\) propose its use for the specification of integrity con-
straints of a data base. In the data base area Kintala\(^\text{13}^\) has used the
attribute method in a typical translation oriented application. He considers
systems that translate queries based on a certain data model into a command
sequence based on another data model. In this work he uses attribute
grammars as a framework for automatic constructions of a query
language translator, similar to that of semantics-directed compiler
generators. Based on this framework it is also possible to give a formal
proof of the correctness of a query language translator as shown in Ref. 14.
In this paper we shall apply the attribute method for abstract software specification. In the data base area it is important to specify precisely both the semantics of software and the constraints related to data base structures and to data base operations. We have specified earlier a relationally complete query language with the attribute method. That study showed that these kinds of aspects can be defined precisely with the attribute method. In Ref. 16, we briefly and verbally characterized the specification of restructuring software with the attribute method but now we will define it in detail.

Contrary to the usual way of applying the attribute method in the context of concrete syntax, we shall apply it in the context of abstract syntax. Concrete syntax determines which kind of character strings are permitted in the language whereas abstract syntax generates abstract objects and defines structural relationships among them. The constructing principle of our abstract syntax is similar to the principle described in Ref. 17, because we can associate attributes with an abstract syntax constructed with this principle in the same way as with context-free grammars. The role of the abstract syntax, defined in this paper, is to generate different restructuring cases.

In the specification we divide the attributes into two categories: semantic attributes and checking attributes. The checking attributes have the property that they restrict the set of objects generated by abstract syntax. The semantic attributes define the semantic interpretation of abstract objects, and the checking attributes are needed to specify different kinds of constraints, in this case constraints related to data structures and their restructuring. For example, with checking attributes we specify the meaningfulness of the restructuring case like the derivability of the target data structure from the source data structures.

Generally taken we have many well-defined formalisms for manipulating data structures in the data base area. In our opinion it should be possible to use these formalisms (sometimes their position in a certain context is even established) in abstract software specification. We demonstrate that the specification method used in this paper can be adapted to manipulate data structures based on our earlier formalisms. Very often the situation is the contrary or the existing notation must be redefined for the specification method, for example in the specification methods based on specific data types.

In Section 2, we introduce a common data structure description for the flat file and hierarchical data models. Based on this description we have defined a sequence of functions in terms of which we can restructure a hierarchical data base. Both the description and the restructuring functions based on it are defined in the Appendix. We shall apply these
definitions in the definition of the attributes. The constructing principle of abstract syntax and the abstract syntax for generating restructuring cases are considered in Section 3. In Section 4, we define the semantic and checking attributes. We assess informally and in detail our approach and the properties of our restructuring software in Section 5, where we also give an example on the use of our software.

2. THE FORMAL DESCRIPTION OF DATA STRUCTURES

In this section we consider the formal description of data structures based on the flat file and hierarchical data models. It is possible to define a formal description for any hierarchical data base or any file based on the flat file data model with the construction rules introduced in the Definition 9 of the Appendix. The formal description of data structures contains both the instance and schema levels.

In Ref. 19 we have modified the construction rules of the Definition 9 (see the Appendix) so that they take into account the keys of records, too. We have shown that with this modification our formal description has the descriptive power of well known visualizations like Bachmann's diagrams and tabular representation. In this paper the formal description of source data structures does not contain the key component because it is not necessary in the data restructuring process.

Any data structure based on the flat file or hierarchical data models is described as a six-tuple \((r, X, RN, DN, I_r, f_r)\) where

- \(r\) is an instance of the relation
- \(X\) is a relation type
- \(RN\) is a set of the record type names
- \(DN\) is a set of the data item names
- \(I_r\) is an index set
- \(f_r : RN \cup DN \rightarrow I_r\) is a naming function

As a consequence of the construction rules described in the Definition 9 of the Appendix, the components of the six-tuple description have the following properties:

1° \(r \in X\)
2° \(RN \cap DN = \emptyset\)
3° \(|RN| + |DN| = |I_r|\)
4° The naming function \(f_r : RN \cup DN \rightarrow I_r\) is bijective
5° The relation type \(X\) is a Cartesian product

\(T_1 \times \cdots \times T_n\) where each \(T_i\) \((1 \leq i \leq n)\) can be either a primitive type (int, real, char) or the power set \(P(Y)\) of some other relation type \(Y\).
6° The elements of the index set $I_x$, which are
associated with the relation type $X = T_1 \times \cdots \times T_n$,
are $n$-tuples such that
\[ \forall i \in \{1, \ldots, n\}: \langle i \rangle \in I_x, \]
\[ \langle i, \xi \rangle \in I_x \text{ iff } T_i = P(Y) \text{ and } \xi \in I_y. \]

Let us suppose that we have a sample data base visualized with
Bachmann's diagram in Fig. 1.

If the data items $A, D, H,$ and $I$ are of type integers (int), and the data
items $B, E$ are of type character strings (char), and the data item $F$ is of
type reals (real) then our sample data base has the six-tuple (or data struc-
ture element, briefly dse) description of Fig. 2.

Our formal description corresponds to the concept “form” of
CONVERT.\textsuperscript{(20, 22)} This kind of a relation is also called power set type
relation.\textsuperscript{(23, 24)}

The flat file data model consists at the schema level only of two logical
data units: a data item and a record type. A data item is the smallest
logical data unit and a record type is a collection of data items.\textsuperscript{(25)} Usually
the instance of a record type is called a record and a set of records is called
a file. In fact the flat file data model can be interpreted as a special case of
the hierarchical data model where a hierarchical data base can consist of
only one record type or the root record type.

It is worth noting that the flat file data model does not define how the
data structures based on it must be implemented physically. In the physical
implementation we use and need information to access and process the
data efficiently. The data structures based on the flat file data model can be
organized, for example as a sequential file, an index sequential file, a hash
file and so on.

---

Fig. 1. The sample data base.
(\{a_1, b_1, \{d_1, e_1, f_1\}, \{d_2, e_2, f_2\}, \{h_1, i_1\}\}, \{a_2, b_2, \{d_3, e_3, f_3\}, \{h_2, i_2\}, \{h_3, i_3\}\}, p(int\text{x}char\text{x}p(int\text{x}char\text{x}real)\text{x}p(int\text{x}int)), \{X, C, G\}, \{A, B, D, E, F, H, I\}, \{c_1, c_2, c_3\}, \{a_1, a_2\}, \{b_1, b_2\}, \{f_1, f_2\}, \{e_1, e_2\}, \{h_1, h_2\}, \{i_1, i_2\}\})

Fig. 2. The sample data base as a data structure element.

In the structural sense a data structure based on the flat file data model is analogous to a relation of the relational data model. In Refs. 26 and 27 we have defined in an analogous way all the components and operations of the relational data model so that the schema and instance levels remain together.

Later we shall define an abstract syntax for data restructuring in such a way that we can define attributes in its context to describe data structures as six-tuples (or as dse's) as previously introduced. Due to this fact we can apply the functions of the Appendix.

In the recursive definition of a dse (see Definition 9 of the Appendix) the rule 2 concatenates only two dse's into one dse. By generalizing this rule, it is possible to construct records. Likewise we can concatenate all hierarchical substructures depending on a certain record into one structure in terms of this generalized rule. Later in the definition of the attributes we shall apply this generalized rule.

**Definition 1.** Let $t_1, t_2, \ldots, t_n$ be any dse's such that

$$t_i = (r_i, X_i, RN_{r_i}, DN_{r_i}, I_{r_i}, f_{r_i}), \quad i \in \{1, \ldots, n\}$$

then

$$\times_{i=1}^n t_i = \langle t_1, t_2 \rangle, \ldots, t_n \rangle$$

Definition 9 in the Appendix shows how $\langle t_1, t_2 \rangle$ is defined.

The function data baselormer shown in the Appendix (see Definition 30) requires the description of the key(s) of the record type(s) of the target schema. In fact this function consists of a sequence of functions defined in the Appendix. Because we shall apply this function in the definition of the semantics of the restructuring software we will now consider how we must describe a target schema for this function. A target schema is also described as a six-tuple, however not as a dse. A target schema has the following formal description: $(key, Y, RN, DN, I, f_i)$ where the components $Y$ (a relation type), $RN$ (a set of record type
names), $DN$ (a set of data item names), $I_x$ (an index set), $f_x$ (a naming function) are described in the same way as in the context of a dse. The key component express the key(s) of the record type(s), i.e. it has the property: key $\subseteq DN$.

Later in the definition of the attributes we will need the constructing rules for a target schema. Due to the fact mentioned previously these rules differ from the corresponding rules of a dse (see Definition 9 in the Appendix) in respect with the key component.

**Definition 2.** Let

$$s_1 = (\text{key}_1, Y_1, RN_1, DN_1, I_{s_1}, f_{s_1})$$

and

$$s_2 = (\text{key}_2, Y_2, RN_2, DN_2, I_{s_2}, f_{s_2})$$

be two (target) schema components such that $(RN_1 \cup DN_1) \cap (RN_2 \cup DN_2) = \emptyset$. The expression $\langle s_1, s_2 \rangle$ constructs the schema component

$$(\text{key}_1 \cup \text{key}_2, Y_1 \times Y_2, RN_1 \cup RN_2, DN_1 \cup DN_2, I_{s_1 \times s_2}, f_{s_1 \times s_2})$$

where

$$I_{s_1 \times s_2} = I_{s_1} \cup (I_{s_2} \oplus \text{max}(I_{s_1}))$$

and

$$f_{s_1 \times s_2}(a) = \begin{cases} f_{s_1}(a), & \text{if } a \in RN_1 \cup DN_1 \\ f_{s_2}(a) + \text{max}(I_{s_1}), & \text{if } a \in RN_2 \cup DN_2 \end{cases}$$

The function max and the operation $\oplus$ have been defined in the definitions 5 and 6 in the appendix.

The definition above is analogous to the construction rule 2 in Definition 9. In the same way we define the analogous construction rule for the construction rule 3 of a dse.

**Definition 3.** Let $s = (\text{key}, Y, RN, DN, I_x, f_x)$ be any schema component and $A$ a record type name such that $A \notin RN \cup DN$ then $A[:]$s constructs the schema component

$$(\text{key}, P(Y), RN \cup \{A\}, DN, I_{p[1]}, f_{p[1]})$$

where

$$I_{p[1]} = \{1\} \cup \{1, \xi \mid \xi \in I_x\}$$

and

$$f_{p[1]}(a) = \begin{cases} f_{p[1]}(A) = 1, & \text{if } a \in RN \cup DN \\ f_{p[1]}(a) = 1, f_{s}(a) : a \in RN \cup DN \end{cases}$$
Analogously to Definition 1 we define the generalized concatenation operation for schema components.

**Definition 4.** Let \( s_1, s_2, s_3, \ldots, s_n \) be any schema components such that \( s_i = (\text{key}_i, Y_i, RN_i, DN_i, I_n, f_n), i \in \{1, \ldots, n\} \). Now

\[
\forall s_i = \{ \cdots \{ (s_1, s_2), s_3, \ldots, s_n \} \}
\]

See Definition 2 above.

Furthermore we shall use the following notational conventions in the definition of the attributes.

**Notational Convention A.** Let \( t \) be a dse or a schema (component) then \( \sigma_i(t); i \in \{1, \ldots, 6\} \) selects the \( i \)th component. For example, if \( t \) is a dse then \( \sigma_1(t) \) and \( \sigma_2(t) \) select the relation instance and relation type components. If \( t \) is a schema (component) then \( \sigma_i(t) \) gives the keys of the record types in the schema (component).

**Notational Convention B.** Let \( X \) be the relation type component of a dse or of a schema (component) and let \( \xi \) be an index; i.e. \( \xi \in I \), then with \( X_\xi \) we denote that type in \( X \) with which the index \( \xi \) is associated. For example, in our sample data base \( X = P(\text{int} \times \text{char} \times P(\text{int} \times \text{char} \times \text{real}) \times P(\text{int} \times \text{int})) \) and in this case \( X_{(1,2)} = \text{char} \) and \( X_{(1,4,1)} = \text{int} \).

3. THE CONSTRUCTING PRINCIPLE OF ABSTRACT SYNTAX AND THE ABSTRACT SYNTAX FOR DATA RESTRUCTURING

Abstract syntax generates abstract objects and it determines structural relationships among them. Abstract syntax affords the possibility of concentrating on essential concepts and their interactions in the specification because by its use we can avoid irrelevant details associated with the representation of a language. Due to this fact the size of abstract syntax is also much smaller than the size of corresponding concrete syntax.

In the data base area, for example Date has defined an abstract syntax for illustrating the abstract structure of various components of the relational data model. The semantics of these components he defines mainly informally.

Kurki-Suonio has introduced a method which associates semantics with abstract syntax. In this method we can associate attributes with abstract syntax in the same way as with context-free grammars. He applies this method for programming language specifications. We apply the same constructing principle of abstract syntax in this paper.
3.1. The Constructing Principle of Abstract Syntax

An abstract syntax consists of structural productions that express structural relationships among (abstract) objects. It is typical of structural relationships that they resemble the productions of context-free grammars. Each nonterminal of an abstract syntax represents a class of objects and we assume the classes of different nonterminals disjoint. An object may have other objects as its components, however so that an object and its component objects form always a tree structure. In a tree structure objects depending immediately on the root object are called immediate components of the root object. Consider the following graph, where each node represents one object.

\[
\begin{array}{c}
  a \quad x \quad f \\
  b \\
  c \quad d \quad e
\end{array}
\]

In the graph the immediate components of the object \( x \) are \( a, b, \) and \( f \), and all its components are \( a, b, c, d, e, f, g \). The immediate components of \( b \) are \( c, d, e \). In abstract syntax there can be for each nonterminal many structural productions that indicate the different ways in which objects of this class may have other objects as their immediate components.

Next we consider an example of how structural productions are described in abstract syntax. The structural production

\[
db: Data \ base \rightarrow \text{RESTRUCTURING} \langle f: File, ts: Target \ schema \rangle
\]

expresses that an object belonging to the class \( Data \ base \) has two immediate components, one of which belongs to the class \( File \) and the other to the class \( Target \ schema \). \( \text{RESTRUCTURING} \) is the name of the structural production and \( db, f, \) and \( ts \) are the names of objects in this structural production.

In abstract syntax the notations

\[
f: Obj^* \quad \text{and} \quad f: Obj^+
\]

denote a sequence that consists of an arbitrary finite number of objects belonging to the class \( Obj \). The notation \( f: Obj^* \) permits a sequence to be empty whereas the notation \( f: Obj^+ \) expresses that the sequence consists at least of one object. We identify the individual objects in a sequence by \( f[1], f[2], \ldots, f[n] \) when \( n \) is the number of objects in the sequence. The index set \( \{1, \ldots, n\} \) is denoted by \( i(f) \). Due to the indexing mechanism the
The cardinality of \(|i(f)|\) or \(|i(f)|\) expresses the greatest number in the sequence. 
\(|i(f)|\geq 0\) in the case \(f:Obj^*\) and \(|i(f)|\geq 1\) in the case \(f:Obj^+\).

We terminate a structure by indicating explicitly that an object of a class is atomic. For example, the structural production

\[
\text{rn: Record type name} \rightarrow \text{R-NAME} \\
\]

expresses that an object belonging to the class Record type name is atomic. Generally, we can say that the structure of an atomic object class has no interest from the viewpoint of the system being specified.

### 3.2. The Abstract Syntax for Data Restructuring

The role of the abstract syntax intended for data restructuring is to generate different restructuring cases. Therefore, the abstract syntax must be able to generate different kinds of (abstract) objects belonging to the schema and instance levels. We build up the abstract syntax in such a way that in the definition of the attributes it is possible to apply the functions previously defined and in the Appendix. The role of the attributes is to define in detail how the target data structure is constructed from a finite set of source data structures.

The abstract syntax consists of the following numbered structural productions.

1. \(db: \text{Data base} \rightarrow \text{RESTRACTURING} \langle f: \text{File}, \\
   ts: \text{Target schema} \rangle \)
2. \(f: \text{File} \rightarrow \text{FILECONNECTIONEXPR} \langle f_1: \text{File}, f_2: \text{File} \rangle \)
3. \(f: \text{File} \rightarrow \text{SOURCEFILE} \langle r: \text{Record}^+ \rangle \)
4. \(r: \text{Record} \rightarrow \text{RECORD} \langle \text{rn: Record type name}, \text{de: D-itemexpr}, \\
   g: \text{Group}^*, \text{key: D-name}^+ \rangle \)
5. \(g: \text{Group} \rightarrow \text{GROUP} \langle r: \text{Record}^+ \rangle \)
6. \(de: \text{D-itemexpr} \rightarrow \text{DATAITEM} \langle \text{dn: D-name}, \text{dt: D-type}, \\
   d: D-value \rangle \)
7. \(ts: \text{Target schema} \rightarrow \text{T-SCHHEMA} \langle rt: \text{Record type} \rangle \)
8. \(rt - 1: \text{Record type} \rightarrow \text{R-TYPE} \langle \text{rn: Record type name}, \text{ds:} \\
   D-item description^+, rt - 2: \text{Record type}^*, \text{key: D-name}^+ \rangle \)
9. \(ds: \text{D-item description} \rightarrow \text{D-DESCRIPTION} \langle \text{dn: D-name}, \text{dt: D-type} \rangle \)
10. \(ds: \text{D-item description} \rightarrow \text{KEY-DATAITEM} \langle \text{dn: D-name}, \text{dt: D-type} \rangle \)
11. \(dn: \text{D-name} \rightarrow \text{D-NAME} \rangle \)
12. \(rn: \text{Record type name} \rightarrow \text{R-NAME} \rangle \)
13. \(dt: \text{D-type} \rightarrow \text{D-TYPE} \rangle \)
14. \(d: \text{D-value} \rightarrow \text{D-VALUE} \rangle \)
Intuitively the structural productions (1-3) describe how a desired target data structure is constructed from source data structures. Finally a target data structure is restructured from one **File** object (the structural production (1)). In the structural production (2) we combine two File objects into one **File** object. The File objects to be combined can be either source data structures (generated by the structural production (3)) or intermediate files (generated by the structural production (2)). In Fig. 3 we illustrate how we construct a target data structure from four source data structures. We refer to the structural productions with their numbers.

With the definition of the attributes associated with these structural productions we define the construction of a target data structure in detail. Likewise we check with the attributes that the target data structure is derivable from the source data structures. An (abstract) object belonging to the class Data base is similar to an object belonging to the class File. The definition of the new class is necessary, however because a File object could be an immediate component of a File object generated by the structural production (2). Of course we want to avoid this situation.

Intuitively the structural productions (3-6) and (11-14) describe the elements of which data structures based on the flat file or hierarchical data model consist. The structural productions (7-13) generate the components of the target schema. The structural production (10) generates the descriptions of those data items from which the keys of the record types of the target schema are constructed.

Any object generated by an abstract syntax can be represented as a directed graph where each node corresponds to one abstract object. The structural productions (4-6) and (11-14) form that part of our syntax which generates records. A sample object generated by this part is represented in Fig. 4. In the graph we express the classes of objects, the names of the structural productions, and the values of the atomic objects. The sample object in Fig. 4 corresponds to one of the two data base trees in Fig. 1.

---

**Fig. 3.** The illustration for constructing a target data structure.
4. THE DEFINITION OF THE ATTRIBUTES

We suppose that the reader is familiar with the attribute method. In this paper the attributes are used for two purposes: first to define the semantic interpretations of abstract objects (semantic attributes) and second to restrict the set of abstract objects generated by abstract syntax.
(checking attributes). To facilitate understanding of the definition of the attributes we consider the background assumptions which must hold.

Data items are the smallest logical data units in data restructuring. The values of data items are basic elements that remain unchanged in the data restructuring process. In this process we can regroup, eliminate or duplicate values of data items but actual new values are not created. Thus each data item of the target schema has a corresponding data item in the source data structures. We suppose that same names are used for corresponding data items in the source data structures and in the target schema.

We can simplify our notation because we do not allow renaming of data items. However, this feature has no effect on restructuring the structural relationships among data. The record type names can, of course, be different in the source data structures and in the target schema because their structure and data item content can be quite different.

We have previously illustrated how a target data structure is constructed in the final phase from one File object. In order to achieve this File object we must connect data structures. In this connection we need the values of the corresponding data items (we use the same names) to indicate relationships among data. We check with the checking attribute that the data structures to be connected have the corresponding data items.

4.1. Informal Characterization of the Attributes

It is desirable that each formal specification is connected to its environment by informal explanations. Therefore we discuss the attributes associated with our abstract syntax informally before their exact definition. The semantic attributes are dse, n-form, s-element, s-content and r-name. The attribute s-content is inherited while the others are synthesized.

With the attribute dse we describe the source data structures and the components needed in their construction as dse's, which in turn are described as mathematical six-tuples. Definition 9 in the Appendix contains the definitions for a data structure element and its projection. The result of data restructuring process is a dse corresponding to the target schema, i.e. it is expressed with the attribute dse of the starting nonterminal Database. The definition of the semantics can be characterized as follows.

1. Each source data structure is transformed into its internal representation.

2. Source data structures in their internal representation are connected to one nonhierarchical intermediate file.
3. The function databaseformer (see Definition 30 in the Appendix) constructs the target data structure from this intermediate file.

The attribute n-form (normal form) is associated only with File objects. It describes a data structure in its internal representation or in its normal form. The internal representation of a hierarchical data structure does not contain any hierarchy. A data structure based on the flat file data model is already in its normal form. In Definition 25 in the Appendix we have defined the function normalization which transforms a hierarchical data structure into its normal form without losing any structural information in this process. When applying this function to a flat file it yields the flat file in question. Data structures in their normal form can be connected by the function join (Definition 24 in the Appendix).

The attribute s-element (schema element) describes the target schema as a six-tuple \((key, Y, RN, DN, I, f_r)\) where the components have the properties defined previously. In the definition of the attribute s-element we shall use the Definitions 2-4.

The attribute s-content (schema content) is set-valued and it is needed to express the data item content of the target schema. Namely we extract from source data structures only the values of those data items which are included in the target schema.

The record type names are expressed by the attribute r-name. This information is needed in grouping records of the same type (see the structural productions (3) and (5)). In the grouping of records we shall apply the rule 3 of Definition 9 in the Appendix. This rule requires a record type name.

In the definition of the checking attributes we shall use the semantic attributes. Our checking attributes are: n-unique, right-key, type-compatible, c-check, e-legal, c-legal, c-compatible, sametype and legal-value. All checking attributes are Boolean valued and synthesized. The attributes in question have the value true in each object generated by our abstract syntax because the abstract syntax generates only those objects which have the legal values of the checking attributes. In this way the checking attributes restrict the set of objects generated by the abstract syntax.

The attribute n-unique (name unique) checks that the names of data items and record types are unique. The operations 2 and 3 of Definition 9 in the Appendix require the uniqueness of the data item and record type names. The uniqueness of the names is also necessary to guarantee that the naming function in a dsr or a target schema is bijective. When we apply the Definitions 1 and 4 into the definition of the attributes, we must check the uniqueness of the names in dsr's and schema components.

The attribute right-key checks that the key of each record type has
been constructed from data items included in the record type. As a typical feature of a data structure we also define this attribute for source data structures, although we do not manipulate explicitly the keys of the record types of source data structures in the restructuring process.

The value true of the attribute type-compatible indicates that the types of the corresponding data items (we use the same names) are compatible in the source data structures and in the target schema.

The value true of the attribute c-check (content check) expresses that each data item in the target schema has a corresponding data item in the source data structures. In the other case the target data structure is not derivable from the source data structures.

The attribute e-legal (extraction legal) checks that each source data structure contains data items that are needed in the construction of the target data structure. Because we define this attribute for the nonterminal File we must also define it for File objects generated by the structural production (2). In this case we define its value as true.

We define the attribute c-legal (connection legal) to check that the data structures to be connected have one or more corresponding (common) data items in terms of whose values the associated data are identified. We use the common names for the corresponding data items as explained previously. We define this attribute for the nonterminal File.

The attribute c-compatible (connection compatible) checks that the corresponding data items used in the connection of data structures have the same types.

We can group records with the rule 3 of Definition 9 in the Appendix. The requirement for this grouping operation is that the records for a group are of the same type. This is checked by the attribute sametype. The attribute sametype is defined for the nonterminals File and Group.

The attribute legal-value is defined to check that each data item value in the source data structures belongs to its domain.

Next we give exact definitions for the semantic and checking attributes. We refer to the structural productions with their numbers.

\[ \text{semantic attributes} \]
\[
dse(d_b) = \text{databaseformer}(n-form(f), s-element(ts))
\]
\[/* \text{The function databaseformer has been defined in Definition 30 in the Appendix} */\]
\[
s-content(f) = \sigma_s(s-element(ts))
\]
\[/* \text{see notational convention A} */\]

\[ \text{checking attributes} \]
\[
c-check(d_b) = \sigma_s(s-element(ts)) \subseteq \sigma_s(n-form(f))
\]
type-compatible(db) = \bigwedge_{a \in s \in s\text{-element}(ts))} \sigma_4(n\text{-form}(f)) f_4(a)
= \sigma_4(s\text{-element}(ts)) f_4(a)
\text{where } f_4 \text{ and } f_4 \text{ are the naming functions associated with } n\text{-form}(f) \text{ and } s\text{-element}(ts)

/* see notational convention B */

(2) semantic attributes

n-form(f) = \text{join} \left( \bigcup_{i \in A} f_1(i), \bigcup_{i \in B} f_2(i) \right)

where \( f_1 \) and \( f_2 \) denote the naming functions associated with n-form\((f_1)\) and n-form\((f_2)\)

\[ A = s\text{-content}(f_1) \cap \sigma_4(n\text{-form}(f_1)) \] \[ B = s\text{-content}(f_2) \cap \sigma_4(n\text{-form}(f_2)) \]

/* In A and B we extract only the data items needed for constructing the target data structure. The function join has been defined in Definition 24 in the Appendix */

dse(f) = n\text{-form}(f)

s-content(f_1) = s\text{-content}(f)

s-content(f_2) = s\text{-content}(f)

checking attributes

e-legal(f) = s\text{-content}(f) \cap \sigma_4(n\text{-form}(f_1)) \neq \emptyset
\land s\text{-content}(f) \cap \sigma_4(n\text{-form}(f_2)) \neq \emptyset

c-legal(f) = \sigma_4(n\text{-form}(f_1)) \cap \sigma_4(n\text{-form}(f_2)) \neq \emptyset

c-compatible(f) = \bigwedge_{i \in A, \sigma_4(n\text{-form}(f_1)) f_1(i)} \sigma_4(n\text{-form}(f_1)) f_1(i)
\text{where } f_1 \text{ and } f_2 \text{ denote the naming functions associated with n-form\((f_1)\) and n-form\((f_2)\)}

sametype(f) = true

n-unique(f) = true

(3) semantic attributes

dse(f) = r\text{-name}(r[1]) : \bigcup_{i \in r} \{dse(r[i])\}

n-form(f) = \text{normalization} (dse(f), \text{leaf}(\sigma_4(dse(f))))

/* The function normalization and the operation: have been defined in Definitions 9 and 25 in the Appendix. The function leaf has been defined in Definition 10 of the Appendix */

checking attributes

e-legal(f) = true

dse(f) = true

n-form(f) = true
c-legal(f) = true

\text{c-compatible}(f) = true

\text{sameType}(f) = \bigwedge_{i, j \in f} \sigma_2(\text{dse}(r[i])) = \sigma_2(\text{dse}(r[j]))
\land \sigma_3(\text{dse}(r[i])) = \sigma_3(\text{dse}(r[j]))
\land \sigma_4(\text{dse}(r[i])) = \sigma_4(\text{dse}(r[j]))
\land \sigma_5(\text{dse}(r[i])) = \sigma_5(\text{dse}(r[j]))
\land \sigma_6(\text{dse}(r[i])) = \sigma_6(\text{dse}(r[j]))

\text{n-unique}(f) = \text{r-name}(r[1]) \notin \sigma_3(\text{dse}(r[1])) \cup \sigma_4(\text{dse}(r[1]))
/* Because we used \text{r-name}(r[1]) as the name in \text{the operation: see the previous definition of the attribute dse} */
this name must be disjoint with other record type and data item names of a dse (see Definition 9 in the Appendix)*/

(4) \text{semantic attributes}

\text{dse}(r) = \begin{cases} 
\prod_{i \in \text{dse}} \prod_{j \in \text{dse}} \text{dse}(r[i]), & \text{if } t(g) = \emptyset /* leaf record */ \\
\bigcup_{i \in \text{dse}} \prod_{j \in \text{dse}} \text{dse}(g[i]), & \text{if } t(g) \neq \emptyset 
\end{cases}
/* See definition 1 */

\text{r-name}(r) = r_n
/* We need this attribute for grouping records of the same type */

\text{checking attributes}

\text{right-key}(r) = \text{key} = \bigcup_{i \in \text{dse}} \sigma_2(\text{dse}(r[i]))

\text{n-unique}(r) = \begin{cases} 
\bigwedge_{i, j \in \text{dse}} \sigma_2(\text{dse}(r[i])) \neq \sigma_2(\text{dse}(r[j])), & \text{if } t(g) = \emptyset \\
\left( \bigwedge_{i, j \in \text{dse}} \sigma_2(\text{dse}(r[i])) = \sigma_2(\text{dse}(r[j])) \right) \land \\
\left( \bigcup_{i \in \text{dse}} \sigma_2(\text{dse}(r[i])) \right) \land \left( \bigcup_{i \in \text{dse}} \sigma_3(\text{dse}(r[i])) \right) \cup \\
\sigma_4(\text{dse}(g[i])) = \emptyset \right) \land \\
\left( \bigwedge_{i, j \in \text{dse}} \sigma_3(\text{dse}(r[i])) \cup \sigma_4(\text{dse}(r[i])) \\
\sigma_5(\text{dse}(g[i])) \right) \cup \\
\sigma_6(\text{dse}(g[i])) = \emptyset, & \text{if } t(g) \neq \emptyset 
\end{cases}
(5) semantic attributes
\[ \text{dsc}(g) = \text{r-name}(r[1]) \cup \bigcup_{i \in \text{dr}} \{ \text{dse}[r[i]] \} \]

checking attributes
\[ \text{sam-type}(g) = \bigwedge_{i \neq j, i,j \in \text{dr}} \sigma_3(\text{dse}[r[i]]) = \sigma_2(\text{dse}[r[j]]) \]
\[ \land \sigma_3(\text{dse}[r[i]]) = \sigma_1(\text{dse}[r[j]]) \]
\[ \land \sigma_4(\text{dse}[r[i]]) = \sigma_4(\text{dse}[r[j]]) \]
\[ \land \sigma_5(\text{dse}[r[i]]) = \sigma_5(\text{dse}[r[j]]) \]
\[ \land \sigma_6(\text{dse}[r[i]]) = \sigma_6(\text{dse}[r[j]]) \]

\[ \text{n-unique}(g) = \text{r-name}(r[1]) \notin \sigma_3(\text{dse}(r[1])) \cup \sigma_4(\text{dse}(r[1])) \]
/*Due to the attribute sam-type \( \sigma_3(\text{dse}[r[1]]) = \sigma_1(\text{dse}[r[i]]) \)
and \( \sigma_4(\text{dse}[r[1]]) = \sigma_4(\text{dse}[r[i]]), i \neq 1, i \in i(r)*/

(6) semantic attributes
\[ \text{dse}(dc) = (d, dt, \emptyset, \{ dn \}, \{ \langle 1 \rangle \}, f(dn) = \langle 1 \rangle) \]
/*See the rule 1 of Definition 9 in the Appendix*/

checking attributes
\[ \text{right-value}(dc) = \begin{cases} 
\text{true}, \text{if } d \in dt \\
\text{false}, \text{if } d \notin dt 
\end{cases} \]

(7) semantic attributes
\[ \text{s-element}(rs) = \text{s-element}(rt) \]

(8) semantic attributes
\[ \text{s-element}(rt - 1) = \begin{cases} 
\text{rn}[:] & \text{if } i(rt - 2) = \emptyset \\
n_{i(rt)} & \text{else} \\
\text{s-element}(ds[i]) \\
\text{s-element}(rt - 2[j]) & \text{if } i(rt - 2) \neq \emptyset \\
\text{s-element}(rt - 2[j]) & \text{else} \\
\end{cases} \]
/*See the Definitions 2-4*/
checking attributes

\[
\left( \bigwedge_{i \neq j, \{i,j\} \in \Delta(d)} \sigma_4(s\text{-element}(ds[i])) \right) \\
\neq \sigma_4(s\text{-element}(ds[j])) \land \\
\forall n \neq \bigcup_{i \in \Delta(d)} \sigma_4(s\text{-element}(ds[i])), \\
\text{if } \mathcal{N}(\mathcal{R} - 2) = \emptyset \\
\left( \bigwedge_{i \in \Delta(d)} \sigma_4(s\text{-element}(ds[i])) \right) \\
\neq \sigma_4(s\text{-element}(ds[j])) \land \\
\left( \bigwedge_{i \in \Delta(d)} (\sigma_4(s\text{-element}(\mathcal{R} - 2[i]))) \\
\cup \sigma_4(s\text{-element}(\mathcal{R} - 2[j]))) \right) \\
\cap \left( \bigwedge_{i \in \Delta(d)} (\sigma_4(s\text{-element}(\mathcal{R} - 2[i]))) \\
\cup \sigma_4(s\text{-element}(\mathcal{R} - 2[j]))) \right) = \emptyset \land \\
\left( \bigcup_{i \in \Delta(d)} \sigma_4(s\text{-element}(ds[i])) \right) \\
\cap \left( \bigcup_{j \in \mathcal{N}(\mathcal{R} - 2)} \sigma_4(s\text{-element}(\mathcal{R} - 2[j])) \right) = \emptyset \land \\
\left( \left( \bigcup_{i \in \Delta(d)} \sigma_4(s\text{-element}(ds[i])) \right) \\
\cup \left( \bigcup_{j \in \mathcal{N}(\mathcal{R} - 2)} \sigma_4(s\text{-element}(\mathcal{R} - 2[j])) \right) \right) = \emptyset \\
\text{if } \mathcal{N}(\mathcal{R} - 2) \neq \emptyset
\]

(9) semantic attributes
s\text{-element}(ds) = (\emptyset, dt, \emptyset, \{dn\}, \{<1>\}, f(dn) = <1>)

(10) semantic attributes
s\text{-element}(ds) = (\{dn\}, dt, \emptyset, \{dn\}, \{<1>\}, f(dn) = <1>)

The generated object in Fig. 4 corresponds to one of the two data base trees of our sample data base. By evaluating the attributes defined above we get the following value for the attribute dse of this object

\[
\langle a_1, h_1, \{<d_1, e_1, f_1>, <d_2, e_2, f_2>\}\{<h_1, i_1>\} >, \\
\text{int} \times \text{char} \times P(\text{int} \times \text{char} \times \text{real}) \times P(\text{int} \times \text{int}), \\
\{C, G\}, \{A, B, D, E, F, H, I\}, \\
\]
\{\langle 1 \rangle, \langle 2 \rangle, \langle 3 \rangle, \langle 3, 1 \rangle, \langle 3, 2 \rangle, \langle 3, 3 \rangle, \langle 4 \rangle, \langle 4, 1 \rangle, \langle 4, 2 \rangle \},
\{f(A) = \langle 1 \rangle, f(B) = \langle 2 \rangle, f(C) = \langle 3 \rangle, f(D) = \langle 3, 1 \rangle, f(E) = \langle 3, 2 \rangle,
  f(F) = \langle 3, 3 \rangle, f(G) = \langle 4 \rangle, f(H) = \langle 4, 1 \rangle, f(I) = \langle 4, 2 \rangle \}\}

We have specified our data restructuring software in an abstract but precise way. The semantics of our software was defined with the semantic attributes that define in detail all data processing tasks needed in the construction of a target data structure. Our specification contains the manipulations of the instance and schema levels. We utilized in the specification the restructuring functions defined in.\(^2\) This was possible because our abstract syntax generates objects for which we can define such attributes that are required by these functions.

In database software specification it is also important to specify all constraints related to database structures and their manipulations. In our specification these kinds of constraints are defined by the checking attributes. Generally we can say that the checking attributes restrict the set of restructuring cases generated by our abstract syntax so that only legal and meaningful restructuring cases are generated.

In Ref. 30 we have compared the specification method used in this paper with the VDM, which has been applied widely for specification in the data base area, too. The comparison concentrated on their suitability in specifying the essential features of data base applications such as the expressive power of data base software and constraints related to data base structures and operations. We did not set these formal specification methods into an absolute order of priority because it has been widely recognized (see Refs. 9 and 10) that we need more complementary specification approaches to formalize the various aspects of one data base application.

5. INFORMAL CONSIDERATION OF THE SPECIFIED
RESTRUCTURING SOFTWARE

We have formally specified a data restructuring software. In this section we consider informally, and in detail, both our approach and the restructuring capability of our software. Furthermore, we relate the properties of our restructuring software to research done by others in this area and we give a practical example on its use.

Usually the existing restructuring languages are based on a certain data model. When considering these languages we note that they provide, from the viewpoint of a user, two basic approaches to data restructuring. One can be characterized as operation oriented and in the other approach the user specifies in a simplified manner the correspondence between the source and the target data.
CONVERT of IBM\textsuperscript{20-22} represents a typical operation oriented approach to restructuring hierarchical data. CONVERT is a high-level nonprocedural restructuring language, where each data structure is represented as a tabular format called a form. CONVERT provides for its user a set of high-level restructuring operations (Form operations). By nesting explicitly Form operations the user describes a restructuring case. These operations form a process graph whose effective performance, based on the pipelining technique, has been examined in Ref. 31. Also, CDTL (Common Data Translation Language) of SDC\textsuperscript{32} is an operation oriented restructuring language that consists of eleven functions intended for restructuring hierarchical data structures.

UMTDL\textsuperscript{33,34} (University of Michigan Translation Definition Language) is a nonprocedural data restructuring language that is based on a different approach to data restructuring. Namely, UMTDL provides for its users no explicit restructuring operations in terms of which should define the restructuring cases. In UMTDL the correspondence between the source and the target data is defined with ITEM-statement. A UMTDL definition can be seen to consist of a series of Item-to-Item assignments. However, the user must define with a ACCESS PATH the source data that is needed in constructing the target data. In approaches of the UMTDL-type the restructuring algorithm has the burden of finding and performing the appropriate restructuring operations. Although Version II of the Michigan Data Translator was designed to restructure hierarchical data structures, Data Translation Project of Michigan University did not implement it.\textsuperscript{35}

The expressive powers of UMTDL, CONVERT, and CDTL, with respect to the hierarchical restructuring operations classified by Navathe and Fry\textsuperscript{136} were compared in Ref. 34. From the viewpoint of the user our approach to data restructuring resembles most UMTDL because the software specification does not presuppose that the user should define explicitly the required restructuring operations. The approach of this paper is based on our previous work (see Ref. 2) where we standardize the restructuring process of hierarchical data bases in terms of two main phases. In the first main phase the hierarchical data base is transformed into its internal nonhierarchical representation. This phase has been defined so that we do not lose any information that is essential in the structural sense, i.e. the phase is information preserving. This means that any hierarchical data base can be always reconstructed from its internal representation. In the Appendix, the function normalization (Definition 25), which itself has been defined on the basis of other functions, transforms a hierarchical data base into its internal representation.

The second main phase constructs the target data base from the inter-
nal representation. We have defined this main phase generally so that any target database that is derivable from the internal representation can be constructed. The second main phase is performed with the function database-former (Definition 30). In the Appendix, we have listed the definitions of this function and the other functions used in its definition. It is worth noting that the construction process of a target database from the source database is not always information preserving, because the data item content of the target database can be a subset from the data item content of the source database. Because these two main phases had also an essential role in our software specification, we have recalled the definitions of these phases in the Appendix. A detailed discussion about these functions and their interrelationships has been published in Ref. 2.

By standardizing the restructuring process we aim at two main goals: effective restructuring capability and a straightforward user interface. Consider first the restructuring capability of our restructuring process defined by the functions in the Appendix. It is desirable that a data restructuring software is able to restructure as many different restructuring cases as possible. The generality is a basic principle in the definition of the two main phases included in our restructuring process. In the structural sense we can characterize the restructuring capability of our restructuring process as follows. It is able to construct any hierarchical target database that is derivable from the given source database.

In the definition of the restructuring process we have concentrated on restructuring structural relationships among data because it is the most complex task in data restructuring. Due to this fact the definition of the process does not contain a data selection facility. With this facility one selects to the target database only those data item values that meet the given selection criteria. However, data selection is a quite straightforward task and it can also be performed before the actual restructuring process as in CDTL with the SUBSET-function.\textsuperscript{(32)}

Inversion is the restructuring operation that reverses the hierarchical relationships of records in the hierarchical database. Many authors (see Refs. 34 and 36) find inversion the most complex one of the hierarchical restructuring types since it often requires scanning every instance in the hierarchical database. In fact, inversion is the sequence of compression and expansion, which are also hierarchical restructuring operations.\textsuperscript{(36)} For example, CDTL contains the INVERSION-function for inversion whereas in CONVERT inversion must be defined as the sequence of SLICE (corresponds to compression) and CONSOLIDATE (corresponds to expansion) operations. Due to the restructuring capability of our restructuring process it is obvious that it is able to perform all those restructuring operations of the existing hierarchical restructuring languages, which
change structural relationships among data. Of course our restructuring process is able to perform inversion, too. In Ref. 2 we have evaluated, in detail, how the functions of our general restructuring process perform inversion.

It is important to remember that in this paper we have the view point of software specification to data restructuring. In other words, the pure user view point is not sufficient. Navathe and Fry[36] have recognized three essential abstraction levels, (schema level, instance level, and value level) which are involved in data restructuring. Usually the expressive power of data restructuring languages is considered from the view point of the user who is mainly concerned with the schema level. However, the schema modifications have a direct effect on the instances of data and the instance operations required reduce finally to some basic operations of data item values (so called value operations) such as eliminating or duplicating data item values (see Ref. 36). In our software specification we must manipulate data at all these three abstraction levels and therefore our restructuring process has been defined precisely to perform data manipulations at these abstraction levels. One of the ideas of our data structure element concept is to allow flexible and precise interactions between these different abstraction levels.

We mentioned previously that one essential goal for our generalized restructuring process is a straightforward user interface. In the structural sense the minimal information, which is needed for restructuring a hierarchical data base is the source data base and the desired target schema.[21] We have this as a starting point: the source data base and the target schema are the parameters for our restructuring process. From the view point of a restructuring user (Navathe's term[37]) this approach has the benefit that he or she does not need to express explicitly restructuring operations in his or her restructuring description. In other words, the user's description becomes more straightforward.

The usefulness of this feature becomes obvious in complex restructuring cases. Let us assume a restructuring case where the source and target data bases share no record type with the same data item content and in the construction of the target data base we need several expansions, compressions, and inversions of hierarchy levels. The description of this kind of a restructuring case most often requires that the restructuring user must define many intermediate files (e.g. in CONVERT intermediate Forms) when using an operation oriented restructuring language. In this case the description and design of a network of the restructuring operations resembles the creation of an algorithm although the restructuring languages themselves are typically nonprocedural languages. However, in those restructuring cases where only a few restructuring operations are needed,
the feature in question is less important. For example, in CDTL unchanged data can be moved flexibly to the target data base from the source data base with the AS-IS-function.

We have considered our restructuring process in detail, because it has a central role in our software specification and also the extensions of this paper are based on the same kind of an approach. The software specification of this paper has an essentially more extensive restructuring capability in comparison with the restructuring process. Namely the software specification of this paper allows the construction of the target data structure from many source data structures. Data structures can be based on flat file or hierarchical data models.

Data restructuring has been also investigated in the context of the relational model. In structural sense, a flat file and a relation of the relational model resembles each other. However, it is typical for the relational model that it is manipulated with the mathematically exact theory, where different consistency constraints such as functional dependencies have a central role. Flat files, instead, do not require the corresponding theory and they can be stored and manipulated in a non-DBMS environment, too.

Schneiderman and Thomas\(^{38}\) have developed a set of restructuring operations for the relational model. In this work they have not aimed at specifying any software product but the have concentrated on introducing operations in terms of which it is possible to develop the automatic system conversion facility for relational data base systems. Likewise the transformations between the relational model and the hierarchical model are outside of the scope of their paper. We have also proposed the formalization of the relational data model on the basis of data structure element concept to facilitate precise specifications and proofs concerning relational data base restructuring\(^{27}\).

Our software specification constructs one target file or target data base in one run. Below we have summarized those restructuring cases that our software specification is able to restructure. In terms of this specification we can restructure the following:

a. one flat file into another flat file
b. many flat files into one flat file
c. one hierarchical data base into another hierarchical data base
d. many hierarchical data bases into one hierarchical data base
e. one hierarchical data base into one flat file
f. many hierarchical data bases into one flat file
g. one flat file into one hierarchical data base
h. many flat files into one hierarchical data base
i. a set consisting of both flat files and hierarchical data bases into
one flat file or one hierarchical data base.

Our specification checks that there are the necessary structural
relationships among the source data in order that the target data structure
can be constructed. If we must construct many target data structures, then
each target data structure is constructed in its own run. Consider as an
example that interesting restructuring case where a hierarchical data base is
constructed from many flat files. In practice, we are facing this restructur-
ing case when acquiring a hierarchical DBMS to a non-DBMS environ-
ment (data stored as flat files). In this case we must often integrate many
original flat files into a hierarchical data base.

Let us suppose the following sample enterprise environment. The
enterprise has many salesmen, each of which can sell many products
produced by the enterprise. A specific product can have one or more
salesmen. In the process of production each product is assembled from one
or more parts. Each part can have many suppliers.

In this environment there are four source flat files: S-FILE, P-FILE,
PA-FILE, SU-FILE. The record type of S-FILE is SELLER and it consists
of the data items S-NAME, SALARY, P-CODE and SALES (we underline
the key of each record type). S-NAME and SALARY indicate the name
and salary of a seller and SALES expresses how much a seller has sold a
particular product (P-CODE). The record type of P-FILE is PRODUCT
and it contains the following data items: P-CODE, P-NAME, and PRICE,
which express the code, name, and price of a product.

The record type of a PA-FILE is PART and it has the data items:
PA-CODE (the code of a part), PA-NAME (the name of a part), QTY
and P-CODE. QTY expresses how many parts (with PA-CODE) are
needed to a product (with P-CODE). The record type of SU-FILE is SUP-
PLIER with the data items SU-NAME (the name of a supplier),
ADDRESS (the address of a supplier), SU-TIME and PA-NAME. The
data item SU-TIME indicates how soon a supplier is able to supply a par-
ticular part (PA-NAME) to the enterprise.

In our software specification data structures are presented as dse's. In
Ref. 19 we have shown that a dse has the descriptive power of well known
visualizations like Bachman's diagram and tabular representation.
Therefore we can illustrate in the context of our sample data restructuring
case data structures with these visualization methods. In Fig 5 we have
illustrated our sample data restructuring task. The source data structures
(previously mentioned) with their sample instances are described as tabular
representations and the desired target schema as a Bachman's diagram.
Fig. 5. The flat source files and the target schema for the sample restructuring case.

From this starting point our software constructs the target data base (conforming to the target schema) in Fig. 6. Of course it performs several checkings as mentioned above.

It is typical of our formal software specification that it defines the expressive power of our software precisely. This means, among other things, that we specify those structural properties that must be satisfied by the data structures that are manipulated by our software. In the specification we defined the structural properties of flat files and hierarchical data bases precisely. Thus our software specification is not able to restructure network databases. However, it is worth noting that networks can be expressed as a family of hierarchies.391

Fig. 6. The tabular representation of the target data base of the sample restructuring case.
Certain structural relationships must hold among the source data in order that the data structure conforming to the given target schema can be constructed from the given source data structures. With the checking attributes, we specified whether a target data structure is derivable from the source data structures.

Our software specification is complete in the sense that it defines in detail how each data item value in the target data structure is constructed from the source data structures. This required that we dealt with the data restructuring problem at all three abstraction levels classified by Navathe and Fry. In our software specification the construction of a target data structure was defined with the semantic attributes.

The focus of this paper is to specify data restructuring software on the basis of the attribute method. A specification of this type is also called abstract implementation because it specifies the software independently of implementation details. For example, it leaves the implementer free to optimize the aspects concerning the effectiveness of the software.

There are two main approaches to the implementation of data restructuring software (see Ref. 40): the interpreter approach and the generator approach. The distinction between these two approaches is similar to the distinction between an interpreter and a compiler of a programming language. Although we have applied in this paper the attribute method for software specification it is also possible to generate compilers (compiler-compilers) on the same basis. It is obvious that we can also use attribute grammars to generate data restructures in the same way as they are used to generate compilers.

6. CONCLUSION

In the data base area there are many situations that require restructuring of data in files and data bases. In this paper we have specified a data restructuring software that is able to restructure hierarchical data bases and files based on the flat file data model. Our software permits the construction of a target data structure from many source data structures. This feature has many important practical consequences.

Abstract software specification is a tool to manage complexity when developing large softwares. Our abstract specification is based on the well known attribute method. So far very little has been applied in the data base area. In this paper we propose using it for translation oriented specification tasks.

In our specification we applied the attribute method in the context of abstract syntax. In data base applications it is important to specify both the semantics of software and constraints related to data base structures.
and their manipulations. For the former purpose we define semantic attributes and for the latter checking attributes. In our specification task it is important to specify the derivability of the target data structure, i.e., whether a target data structure is derivable from source data structures. The derivability in our specification was expressed by the definitions of different checking attributes. The checking attributes restricted the set of restructuring cases generated by our abstract syntax so that only legal and meaningful restructuring cases were generated.

The applier of the attribute method can use his or her own structures in the definition of the attributes. This feature is important because it supports the intuition of the software developer. It is also valuable if we can utilize, without redefining, the formalisms and functions defined earlier in abstract software specification. In this paper we have demonstrated that our specification method can be adapted flexibly to manipulate notation defined earlier.

The specification method used in this paper requires to master only a few notational conventions. This is because the attribute method is based on a certain way of thinking and also the constructing principle of abstract syntax contains only a few notational conventions.

APPENDIX

The basic notational conventions (briefly nc):

nc-1: Tuples are denoted between angle brackets
nc-2: The notation $\langle i, \xi \rangle$ means a tuple where the first component is $i$ and others form the tuple $\xi$. Analogically, $\langle \xi, i \rangle$ is a tuple whose last element is $i$.
nc-3: The concatenation of two tuples $\xi_1$ and $\xi_2$ is denoted by $\langle \xi_1, \xi_2 \rangle$.

The definitions associated with indices:

1. Indices are tuples consisting of natural numbers.
2. The primitive function $\text{elem}(\xi)$ gives the number of components of the index $\xi$.
3. The primitive function $\rho(\xi, n)$ gives the subindex with length $n$ ($0 < n \leq \text{elem}(\xi)$) from the beginning of the index $\xi$.
4. The primitive function $\text{sum}(\xi)$ expresses the sum of the components of the index $\xi$.
5. $\max(I_1) = \{\xi \mid \xi \in I_1 \land \text{elem}(\xi) = 1\}$ (we use $|Z|$ to denote the cardinality of the set Z)
The operations $\oplus$ and $\ominus$ are defined between the index set $I_x$ and the nonnegative integer $n$ as follows:

6. $I_x \oplus n = \{ \langle i + n, \xi \rangle \mid \langle i, \xi \rangle \in I_x \}$
7. $I_x \ominus n = \{ \langle i - n, \xi \rangle \mid \langle i, \xi \rangle \in I_x \land i > n \}$.

The definitions associated with a data structure element (dse):

8. Let $t_1, t_2, \ldots, t_n$ be some dse's such that

$$t_i(r, X_i, RN_i, DN_i, I_x, f_{x_i}), \quad i = 1, \ldots, n$$

If for any two dse's $t_i, t_j$ ($i \neq j$) the following condition holds ($X_i = X_j$) $\land$
($RN_i = RN_j$) $\land$ ($DN_i = DN_j$) $\land$ ($I_x = I_x$) $\land$ ($f_{x_i} = f_{x_j}$) then the dse's are of the same type.

A dse and its projection are defined at the same time.

nc-4: Let $t = (r, X, RN, DN, I_x, f_x)$ be a dse and $I$ a set of indices.

Then $t[I]$ denotes the projection of $t$ on indices $I$.

9. A dse is either a six-tuple $(r, X, RN, DN, I_x, f_x)$ or the zero element (A). It satisfies the following rules formed recursively and it is constructed by applying these rules finitely.

**Rule 1.** A data item is a dse $(r, D, \emptyset, \{A\}, \{\langle 1 \rangle\}, f_0(A) = \langle 1 \rangle)$

where

$r$ is a data item value with the property $r \in D$,

$D$ is some primitive type (domain). Here primitive types are int (int), reals (real) and character strings (char),

$\emptyset$ means that this dse does not contain any record type name,

the set $\{A\}$ consists of a data item name $A$,

the naming function $f_0 : \{A\} \cup \emptyset \to I_x$ has form: $f_0(A) = \langle 1 \rangle$

i.e., it associates the index $\langle 1 \rangle$ with the data item name $A$.

If $r$ is a data item and $I$ is a set of indices then the projection $t[I]$ is defined in the following way:

$$t[I] = \begin{cases} t, & \text{if } \langle 1 \rangle \in I \\ A, & \text{otherwise} \end{cases}$$

**Rule 2.** Let $t_1 = (r_1, X_1, RN_1, DN_1, I_{x_1}, f_{x_1})$ and $t_2 = (r_2, X_2, RN_2, DN_2, I_{x_2}, f_{x_2})$ be two dse's such that $(RN_1 \cup DN_1) \cap (RN_2 \cup DN_2) = \emptyset$, then the notation $\langle t_1, t_2 \rangle$ constructs the dse $\langle r_1 \times r_2, X_1 \times X_2, RN_1 \cup RN_2, DN_1 \cup DN_2, I_{x_1 \times x_2}, f_{x_1 \times x_2} \rangle$ where
\[ I_{n_1 \times n_2} = I_{n_1} \cup (I_{n_2} \odot \max(I_{n_1})) \]
\[ f_{n_1 \times n_2}: RN_1 \cup RN_2 \cup DN_1 \cup DN_2 \to I_{n_1 \times n_2} \]
\[ f_{n_1 \times n_2}(a) = \begin{cases} f_{n_1}(a), & \text{if } a \in RN_1 \cup DN_1 \\ f_{n_2}(a) + \max(I_{n_1}), & \text{if } a \in RN_2 \cup DN_2 \end{cases} \]

The zero element behaves in this operation as follows:
\[ \langle A, t \rangle = \langle t, A \rangle = t \text{ when } t \text{ is any dse.} \]

If \( t \) is a dse such that \( t = \langle t_1, t_2 \rangle \) then the projection of \( t \) on indices \( I \) is defined as follows:
\[ t[I] = \langle t_1[I], t_2[I \odot \max(I_{n_1})] \rangle \]

**Rule 3.** Let \( T = \{ (r_i, X, RN, DN, I_x, f_x) \mid i = 1, \ldots, n \} \) be a finite set consisting of dse's of the same type and \( A \notin RN \cup DN \), then the notation \( A : T \) constructs the dse
\[ \left( \bigcup_i r_i, P(X), RN \cup \{ A \}, DN, I_{p(x)}, f_{p(x)} \right) \]

where \( P(X) \) is the power set of the relation type \( X \),
\[ I_{p(x)} = \{ \langle 1 \rangle \} \cup \{ \langle 1, \xi \rangle \mid \xi \in I_x \} \]
\[ f_{p(x)} = RN \cup \{ A \} \cup DN \to I_{p(x)} \]
\[ f_{p(x)}(1) = \langle 1, f_x(a) \rangle; \quad a \in RN \cup DN \]

If \( t_i = A, i = 1, \ldots, n \) then
\[ A : \bigcup_i t_i = A \]

If \( t \) is a dse such that \( t = A : T \) then the projection on indices \( I \) is defined as follows
\[ r[I] = A : \bigcup_i t_i[I] \text{ where } t_i \in T \text{ and } I = \{ \xi \mid \langle 1, \xi \rangle \in I \} \]

The definitions of the analyzing functions
10. leaf \((I_x) = \{ \xi \mid \xi \in I_x \land (f_x^{-1}(\xi) \in RN) \land (\forall \langle \xi, i \rangle \in I_x : f_x^{-1}(\langle \xi, i \rangle) \in DN) \} \}
11. r-content \((\xi) = \{ \langle \xi, i \rangle \mid (\langle \xi, i \rangle \in I_x) \land (f_x^{-1}(\langle \xi, i \rangle) \in DN) \} \}
12. predecessor(\langle \xi, i \rangle) = \xi \]
13. $a$-pathtype($ind$)

\[
\begin{cases}
\text{ind} \cup a$-pathtype(predecessor($ind$)), & \text{if } \text{elem}(ind) > 1 \\
\text{ind}, & \text{if } \text{elem}(ind) = 1/*\text{the root record type with which the index } <1> \text{ is associated}*/
\end{cases}
\]

14. $d$-collection($ind$) = $\bigcup_{i \in a$-pathtype($ind$)} r$-content($i$)

15. $\text{partpath}(ind_1, ind_2)$

\[
\begin{cases}
\bigcup_{i \in r$-content($ind_2$)} f^{-1}_v(i) \cup \text{partpath}(ind_1, \text{predecessor}(ind_2)), & \text{if } ind_1 \neq ind_2 \\
\bigcup_{i \in r$-content($ind_2$)} f^{-1}_v(i), & \text{if } ind_1 = ind_2
\end{cases}
\]

16. following($ind_1, ind_2$)

\[
\langle \xi, i \rangle \text{ when } \langle \xi, i \rangle \in a$-pathtype($ind_2$) \land \xi = ind_1
\]

17. $i$-subset($A, i$) = \{ $\xi \in A \land \exists z \in A: \text{sum}(p(z, i)) < \text{sum}(p(\xi, i))$ \}

18. next($A, i$) = \{ $\text{next}(i$-subset($A, i), i + 1), \text{ if } |i$-subset($A, i)| > 1$,

\[\text{if } |i$-subset($A, i)| = 1\}

19. recordkey($\xi$) = \{ $f^{-1}_v(\langle \xi, i \rangle) \mid \langle \xi, i \rangle \in I, \text{ and } f^{-1}_v(\langle \xi, i \rangle) \in \text{key}$ \}

The definitions of the restructuring functions for transforming a data structure into its internal representation:

20. Let $d$ be a dse such that $d:A:T$ then $\sim d = T$.

21. Let $t_1 = (r_1, X_1, RN_1, DN_1, I_{v_1}, f_{v_1})$ and $t_2 = (r_2, X_2, RN_2, DN_2, I_{v_2}, f_{v_2})$ be two dse's such that $(RN_1 \cup DN_1) \cap (RN_2 \cup DN_2) = \emptyset$.

Then the concatenation $t_1 \cap t_2$ = $\bigcup_{i \in \sim t_2} \langle t_1, t_i \rangle$, if $X_2$ is not a type of $P(Y)$ where $Y$

22. $l$-compression($t$)

\[
\text{for } i(k(1)):\bigcup_{k \in i} k[$ \{ $\xi \mid \xi \in I_k$

\[\land \text{elem}(\xi) = 1 \land f^{-1}_v(\xi) \in DN_k]$ \}\land k[$ \{ $\xi \mid \xi \in I_k$

\[\land \text{elem}(\xi) = 1 \land f^{-1}_v(\xi) \in DN_k]$ \}

23. $\text{flat}(t)$ = $\{ \text{flat}(l$-compression($t$)), if $|RN| > 1

\{ I, \text{ if } |RN| = 1 $}
24. join\(t_1, t_2\)
\[
\begin{align*}
&= f^{-1}_{x_1}(\langle 1 \rangle) : \left\{ \left( k_1, k_2 \left[ \bigcup_{a \in D_{N_1} \cap D_{N_2}} f_{k_1}(a) \right] \right) \mid k_1 \in \sim t_1 \wedge k_2 \in \sim t_2 \wedge \forall b \in D_{N_2} \cap D_{N_1} : k_1[f_{k_1}(b)] = k_2[f_{k_2}(b)] \right\}
\end{align*}
\]
where the naming functions associated with the dse's \(k_1\) and \(k_2\) are denoted by \(f_{k_1}\) and \(f_{k_2}\).

25. normalization\(t, A\)
\[
\begin{align*}
\text{normalization}(t, A) = \begin{cases} 
\text{join}(\text{flat}(t[\text{d-collection(next(A))}])), & |A| > 1 \\
\text{normalization}(t, A-\text{next}(A)), & |A| = 1 \\
\text{flat}(t[\text{d-collection(next(A))}]), & |A| = 1 
\end{cases}
\end{align*}
\]
The functions for constructing the target data structure from the internal data representation:

26. h-form\(t_1, t_2, A, B\)
\[
\begin{align*}
&= A : \left\{ \left( k_1, B : S \right) \mid k_1 \in \sim t_1 \wedge k_2 \in \sim t_2 \wedge S \right\} \\
&= \left\{ k_2 \left[ \bigcup_{a \in D_{N_2} \cap D_{N_1}} f_{k_2}(a) \right] \mid k_1 \left[ \bigcup_{a \in D_{N_1} \cap D_{N_2}} f_{k_1}(a) \right] \right\} \\
&= k_2 \left[ \bigcup_{a \in D_{N_2} \cap D_{N_1}} f_{k_2}(a) \right]
\end{align*}
\]
where \(f_{k_1}\) and \(f_{k_2}\) are the naming functions associated with dse's \(k_1\) and \(k_2\).

27. s-unnormalization\(t, s, k, \text{ind}_1, \text{ind}_2\)
\[
\begin{align*}
&= \begin{cases} 
\text{h-form} \left( t \left[ \bigcup_{b \in B} f_s(b) \right] \right), & \text{s-unnormalization} \left( t \left[ \bigcup_{a \in A} f_s(a) \right] \right), \\
& s, k \cup \text{recordkey} (\text{ind}_1), \text{following}(\text{ind}_1, \text{ind}_2), \text{ind}_2, \\
& f_s^{-1}(\langle 1 \rangle), f_s^{-1}((\text{following}(\text{ind}_1, \text{ind}_2))) \end{cases}
\end{align*}
\]
where
\[
B = \bigcup_{i \in \text{content}(\text{ind}_1)} f_s^{-1}(i) \cup k
\]
\[
A = k \cup \text{recordkey} (\text{ind}_1) \cup \text{partpath}(\text{following}(\text{ind}_1, \text{ind}_2), \text{ind}_2);
\]
if \(\text{ind}_1 \neq \text{ind}_2\)
\[
\begin{align*}
&= \begin{cases} 
\text{h-form} \left( t \left[ \bigcup_{b \in B} f_{s_1}(b) \right] \right), & \text{s-unnormalization} \left( t \left[ \bigcup_{a \in A} f_{s_1}(a) \right] \right), \\
& s, k \cup \text{recordkey} (\text{ind}_1), \text{following}(\text{ind}_1, \text{ind}_2), \text{ind}_2, \\
& f_{s_1}^{-1}(\langle 1 \rangle), f_{s_1}^{-1}((\text{following}(\text{ind}_1, \text{ind}_2))) \end{cases}
\end{align*}
\]
where
\[
B = \bigcup_{i \in \text{content}(\text{ind}_1)} f_{s_1}^{-1}(i) \cup k
\]
\[
A = k \cup \text{recordkey} (\text{ind}_1) \cup \text{partpath}(\text{following}(\text{ind}_1, \text{ind}_2), \text{ind}_2);
\]
if \(\text{ind}_1 = \text{ind}_2\)
28. \( h\)-join\((t_1, t_2) \)

\[
\begin{align*}
&f_{s_1}^{-1}(\langle 1 \rangle) = \{ h\text{-join}(k_1, k_2) \mid k_1 \in t_1 \land k_2 \in t_2 \\
&\land \ k_1 \left[ \bigcup_{a \in A} f_{s_1}(a) \right] = k_2 \left[ \bigcup_{a \in A} f_{s_2}(a) \right] \};
\end{align*}
\]

where \( f_{s_1} \) and \( f_{s_2} \) are the naming functions associated with dse's \( k_1 \) and \( k_2 \) and

\[
A = \{ f_{s_1}^{-1}(i) \mid i \in r\text{-content}(\langle 1 \rangle) \};
\]

if \( f_{s_1}^{-1}(\langle 1 \rangle) \in RN_1 \)

\[
\langle t_1[A], h\text{-join}(t_1[B], t_2[C]) \rangle
\]

where

\[
A = \{ a \mid a \in I_n \land \text{elem}(a) = 1 \land f_{s_1}(a) \in DN_1 \}
\]

\[
\cup \{ \langle i, \xi \rangle \mid \langle i, \xi \rangle \in I_n \land f_{s_1}^{-1}(\langle i, \xi \rangle) \in DN_1 \land i \neq \max(I_n) \}
\]

\[
B = \{ \langle i, \xi \rangle \mid \langle i, \xi \rangle \in I_n \land f_{s_1}^{-1}(\langle i, \xi \rangle) \in DN_1 \land i = \max(I_n) \}
\]

\[
C = \{ a \mid a \in I_n \land \text{elem}(a) > 1 \land f_{s_1}^{-1}(a) \in DN_2 \};
\]

if \( f_{s_1}^{-1}(\langle 1 \rangle) \in DN_1 \land RN_1 \cap RN_2 \neq \emptyset \)

\[
\langle t_1, t_2[A] \rangle
\]

where

\[
A = \bigcup_{i \in DN_2 - DN_1} f_{s_1}^{-1}(i);
\]

if \( f_{s_1}^{-1}(\langle 1 \rangle) \in DN_1 \land RN_1 \cap RN_2 = \emptyset \)

29. g-unnormalization\((t_1, t_2, s, A) \)

\[
g\text{-unnormalization}(h\text{-join}(t_1, s\text{-unnormalization}(t_2, s, A, 1)), t_2, s, A\text{-next}(A, 1)); \text{if } A \neq \emptyset
\]

\[
t_1; \text{if } A = \emptyset / \text{all leaf record types of the target schema have been dealt with}^*/
\]

30. databaseformer\((t, s) = g\text{-unnormalization}(s\text{-unnormalization}

\[
(t, s, \emptyset, \langle 1 \rangle, \text{next(leaf}(I_s), 1)), t, s,
\]

\[
\text{leaf}(I_s)\text{-next(leaf}(I_s), 1))
\]

REFERENCES


17. R. Kurki-Suonio, Towards better structured definitions of programming languages, Stanford University, STAN-CS-75-500 (1975).


PART IV
THE SPECIFICATION OF DATA REFORMATTING IN DATA CONVERSION

Timo Niemi
University of Tampere
Department of Mathematical Sciences
Computer Science
P.O. Box 607
SF-33101 Tampere 10, Finland

Abstract: General data conversion contains two essential parts: data restructuring and data reformatting. In this paper we concentrate on data reformatting which takes into account the change of the software/hardware environment when storing data. In our previous work [NIE-3] we have specified precisely data restructuring of flat files and hierarchical data bases. Now we extend this specification to cover data reformatting, too. In other words, after this extension we have specified all essential aspects associated with data conversion. In our extension the encoding of data structures has an essential role. The result of the encoding process of any data structure is a bit string. Our specification deals with the data reformatting problem at the bit string level. Our specification method is based on applying the attribute technique (attribute grammars) in the context of abstract syntax.

Keywords: Data reformatting, data conversion, data encoding, formal specification.
1. INTRODUCTION

The hardware/software (briefly hw/sw) environment determines how (logical) data structures are represented on some external storage medium. In other words, it determines the storing principle of data in detail. It defines how instances of different logical data units such as data items, record types etc. are encoded into bit strings. The process, which defines the bit string representation of a data structure, is called the encoding of a data structure. To enable correct processing of stored data in different hw/sw environments with different storing principles we must change representations of stored data. In other words, the encoding of data is different in these environments. The changing of representations of stored data is called data reformatting. For example, if we want in a hw/sw environment based on the EBCDIC character code process data, which have been stored in a hw/sw environment based on the ASCII character code, we must reformat stored data.

The data reformatting problem is due to the fact that in the existing file and data base management systems (dbms) there are too few levels of explicit data description with different abstraction levels and mappings between them. In terms of the DDL's (Data Description Languages) of the existing dbms's the programs are separated from the description of data. In spite of this the data are still dependent on the hw/sw system which accesses them. In turn, this is the reason to the usual impossibility of processing data in an environment where it has not been created.

It has been widely recognized that we also need such a description level which specifies explicitly the storing principles of data. This means that this description level must contain the aspects associated with the encoding of data structures. For example, the ANSI/X3/SPARC proposal [3] provides, in order to achieve logical and physical data independency, three separate description levels which are associated through explicit mappings. In the internal schema of this proposal one specifies the storing principles of data in detail. Likewise, the DIAM approach [31] makes a clear distinction between logical data structures and their encodings. From the view point of data reformatting any advanced dbms should contain a low level description associated with data encoding and mappings to this level because in terms of them we can insulate the changes in the encoding of data into one level. Also, our experimental dbms [24]
contains a description level for the aspects associated with the encoding of a data structure.

Because the existing DBMSs do not contain an explicit description for the storing principles of data, several non-procedural languages (e.g. [5], [6], [33], [34], [35]) have been developed for this purpose. Usually the expressive power and semantics of these languages is characterized verbally. In this paper we aim at a specification level which defines precisely also those encoding operations which are necessary to achieve bit string representations of flat files and hierarchical data bases.

Two essential parts are usually distinguished in data conversion: data restructuring and data reformatting (see e.g. [16]). Data restructuring transforms structural relationships among logical data that are of interest to the user. It has been recognized that the restructuring process is the most complex part in a generalized data conversion system (see e.g. [7]). We have considered the restructuring process of flat files and hierarchical data bases in detail in [NE-2], [NE-3]. In this paper we shall concentrate on data reformatting.

Depending on the data conversion case, we need either data restructuring or data reformatting or both of them. The data conversion case consisting of only data reformatting arises when the hw/sw environment is changed but the logical data structure (user data) is preserved unchanged. The really troublesome data conversion cases require both data restructuring and data reformatting. For example, we are facing this kind of a conversion case when, in the context of the exchange or acquisition of a DBMS, the computer is also changed.

Considering approaches to data conversion and the developed prototypes of data converters we note that they are based on an analogical architecture. In Fig.1 we describe the sequence of main modules of this architecture (see e.g. [4], [11], [27], [33]).
Fig. 1  A simplified model of data conversion

The READER-module accesses the source data and transforms the source data into an internal form processable by the RESTRUCTURER. The RESTRUCTURER-module performs the restructuring of data. The WRITER-module creates the restructured data onto some storage medium. Data reformatting is associated with the READER and WRITER modules whereas the RESTRUCTURER module is associated with data restructuring. The attempt is to divide the complex data conversion problem into more manageable subproblems by this module architecture. Furthermore, this module architecture affords the possibility of running the modules in different computer environments if required.

The main modules in question are often split into many submodules. It depends on the basic philosophy of data converters how detailed the stored data descriptions are and how many different intermediate forms of data are needed during the data conversion process. For example, according to Shoshani the tasks of the READER and the WRITER can be performed by the utilities available in the source and target systems [32]. In his approach the data conversion problem reduces mainly to data
restructuring problem. In some other approaches the roles of the READER and the WRITER have been seen so essential that they have been divided into submodules. For example, in [33], [35] the PHYSICAL READER and the PHYSICAL WRITER have been distinguished from the READER and the WRITER to manipulate low level hardware details. One of the purposes of this is that the READER and WRITER modules can be used in different hw/sw environments by reprogramming only partly the PHYSICAL READER and the PHYSICAL WRITER.

Considering a data base stored under some dbms, we can note that it does not contain only data that interests the user (user data), but also a large amount of control information (such as indices, overflow chains, delete flags, etc.) specific to a particular system. The existing dbms's provide their users load utilities in terms of which physical data bases organized according to some options can be created from their input (or source) files. The loaded data base contains both user data and control information specific to a dbms whereas the input file of a load utility consists often only of user data.

Likewise many existing dbms's contain unload utilities which are the inverse of load utilities. In other words, the unload utilities reduce the complex physical structures of data bases to very simple physical structures which consist mainly of user data. It is typical of the outputs of unload utilities that they can be reloaded. The purpose of unload utilities is to afford the possibility of changing physical organizations of data bases because their output can be reloaded to a physical data base which has been organized according to different options. It is worth noting that the existing dbms's provide no tools which are able to restructure structural relationships among user data. We have proposed this kind of tools to be included into dbms's [23].

We think that there are no good reasons to include the tasks of the load and unload utilities in the actual data conversion software. This is because these tasks would make the conversion software larger and more complex, and because these utilities are available in the environment. The data restructuring process is concerned only with user data. Therefore it is most flexible to perform the data restructuring process after the unload process which removes from a data base the complex internal structures specific to a particular system. In this mode structural relationships among the user data are easy to restructure. After the restructuring
process the data are loaded with the load utility into some physical data base. In this paper we see the data conversion process as consisting of the following sequential phases:

\[
\text{PHYSICAL SOURCE DATA} \rightarrow \text{UNLOAD} \rightarrow \text{SOURCE DATA}_{UL} \rightarrow \text{SOURCE DATA}_{UL} \rightarrow \text{Source system}
\]

\[
\text{DATA FORMATTING-1} \rightarrow \text{SOURCE DATA}_{R} \rightarrow \text{RESTRUCTURING} \\
\]

\[
\text{TARGET DATA}_{R} \rightarrow \text{DATA FORMATTING-2} \\
\]

\[
\text{TARGET DATA}_{L} \rightarrow \text{LOAD} \rightarrow \text{PHYSICAL TARGET DATA} \rightarrow \text{Target system}
\]

Let us consider briefly the phases of a data conversion process and their roles:

- PHYSICAL SOURCE DATA → UNLOAD → SOURCE DATA_{UL}; with the unload utility of the source dbms we remove from a source data base the complex internal structures specific to the source dbms.

- SOURCE_{UL} → DATA FORMATTING-1 → SOURCE DATA_{R}; this phase changes the format of source data into the form (SOURCE DATA_{R}) processable by the DATA RESTRUCTURER.

- SOURCE DATA_{R} → RESTRUCTURING → TARGET DATA_{R}; this phase restructures structural relationships among logical (user) data.

- TARGET DATA_{R} → DATA FORMATTING-2 → TARGET DATA_{L}; restructured data (TARGET DATA_{R}) are transformed into the prespecified format of the load utility.

- TARGET DATA_{L} → LOAD → PHYSICAL TARGET DATA; the physical target data is created with the load utility available in the target system.
Also, flat files can be organized physically in various ways such as sequential file, indexed sequential file and so on. The physical organization alternatives of flat files can contain, except for user data, information intended to access and process the stored data efficiently. We suppose that the flat files have their own "unload utilities" in terms of which this kind of an information can be removed. Likewise we can assume that in a certain hw/sw environment there are "load utilities" for different physical organization alternatives of flat files. If a hw/sw environment does not contain this kind of utilities it is a quite straightforward task to construct them.

We can specify data reformatting in our data conversion model by specifying the correspondences between SOURCE DATA, and SOURCE DATA, on one hand, and between TARGET DATA, and TARGET DATA, on the other. Although SOURCE DATA, and TARGET DATA, do not contain complex internal structures of dbms's they, however, contain many typical features of the source and target hw/sw environments. In this paper we specify the bit string representations of SOURCE DATA, and TARGET DATA,. Because the UNLOAD and LOAD processes can be performed with utilities available in the source and target environments we can illustrate the specification of the general conversion process (contains both data restructuring and data reformatting) with the attribute technique in the following simplified way.

![Diagram](image)

A

SOURCE DATA STRUCTURES
(as dse descriptions)

TARGET DATA STRUCTURE
(as a dse description)

B

THEIR REPRESENTATIONS
AS BIT STRINGS
(SOURCE DATA,)

ITS REPRESENTATION
AS A BIT STRING
(TARGET DATA,)

Fig 2. The illustration for the specification of data conversion

The attributes associated with A define data restructuring of flat files and hierarchica data bases. This specification has been presented in [NIE-3].
The attributes associated with B define data reformatting. In this paper we shall define these attributes.

Data restructuring contains the transformation of structural relationships among logical (user) data. The dse descriptions of data structures afford the possibility of specifying data restructuring precisely at the all essential abstraction levels indicated by Navathe & Fry [28]. Therefore we defined the restructuring of flat files and hierarchical data bases on the basis of the dse description. Of course in the different phases of the data restructuring process the dse descriptions of data structures have their corresponding bit string representations. However, we do not specify these bit string representations here because in data restructuring the emphasis is in the transformation of structural relationships.

Above the specification of data reformatting is based on the specification of the encoding of the logical data structures. It is worth noting that the result of the encoding process of data structures describes data as a bit string representation independent of its placement on storage media. In the encoding of a logical data structure we rather transform a dse representation into a bit string that can be mapped onto a storage medium.

Based on the approach described in Fig.2 we state the following requirements for the specification of data reformatting:

1. It must contain the encoding of each source data structure and the target data structure into the corresponding bit string representation.

2. The specification together with the data restructuring specification in [Nie-3] must form one specification for data conversion of flat files and hierarchical data bases. In other words, the abstract syntax and the definition of the attributes for data reformatting must be connected to the abstract syntax and the definition of the attributes for data restructuring in [Nie-3].

3. In addition to the hw/sw environment, the user specifications (e.g. the lengths of data items specified by a user) have an effect on the encoding of data structures. Our formal specification must also take into account these aspects specified by a user which have an effect to the encoding of data.
The representations of data at different abstraction levels and the interface between them must be defined precisely and clearly in the specification.

Generally taken we can encode a data structure with numerous different ways. We try to simplify the encoding of data by reducing possible representations of data. The encoding of data and the process of data reformatting become more manageable if we do not need to manipulate many different representations. For example, numeric data can be represented in various (e.g. zoned, packed, floating point etc.) and complex forms. In this paper we shall represent all numeric data in the zoned form. However, this convention does not limit data conversion because, we can assume that in all hw/sw environments there is a unique transformation into this form from the other representations of numeric data and vice versa.

We have numerous variations to realize data structures based on the flat file or hierarchical data model. Therefore we can consider only some usual situations. Typical commercial dbms's are large and complex. For example, IMS developed by IBM is a widely used hierarchical dbms. We shall show that the encoding features of our specification are sufficient for encoding the load format required by IMS. In this paper we emphasize, how the encoding of a data structure can be specified precisely on the basis of the attribute method. It is obvious that we can specify additional encoding features in the same way.

The exact specification of data reformatting requires the manipulation of strings. Therefore we define in chapter 2 briefly the necessary notational conventions and primitive operations for the manipulation of strings. Many aspects associated with data encoding shall be described in our specification as functions. We introduce these functions in chapter 2, too.

The encoding of a data structure reduces to the encoding of its logical data units. In chapter 3 we define the encoding functions for the logical data units included in the flat file or hierarchical data models. In the context of the definition of these functions we shall state explicitly all our background assumptions in the encoding.

In chapter 4 we extend the abstract syntax presented in [NIE-3] in such a way that we can specify the whole data conversion process (data restructuring+data reformatting). The attributes are associated with the extended abstract syntax in the same way as with the abstract syntax in [NIE-3] which we assume to be familiar to the reader. We also assume that the reader masters the notation and functions introduced in [NIE-2], [NIE-3].
2. PRIMITIVE CONCEPTS, OPERATIONS AND NOTATIONAL CONVENTIONS FOR DATA REFORMATTING SPECIFICATION

First we define those concepts and primitive operations which we shall apply in the processing of strings. They can be found in standard literature (see e.g. [1], [19], [20]) and therefore they are introduced briefly. Throughout the paper, we shall underline the primitive operations introduced in this chapter.

Def-1: From our viewpoint a string is a finite sequence of available symbols. The digits (0, ..., 9), alphabetic characters (A, ..., Z), special characters (such as comma, space, plus, minus etc.) are symbols in this paper. We also allow the empty string, denoted by $\varepsilon$, which contains no symbols.

Def-2: The length of a string $s$, written $\text{len}(s)$, is the number of symbols in it. The length of the empty string is 0 or $\text{len}(\varepsilon)=0$.

For example, $\text{len}(12ab)=4$.

Def-3: If $s_1$ and $s_2$ are strings then the string $s_1s_2$ is called the (concatenation of $s_1$ and $s_2$. Instead of $s_1s_2$ we shall denote the concatenation of $s_1$ and $s_2$ by $\text{concatenation}(s_1, s_2)$. If $s$ is any string then $\text{concatenation}(\varepsilon, s)=\text{concatenation}(s, \varepsilon)=s$. In this paper we also allow the concatenation of more than two strings and we denote this by $\text{concatenation}(s_1, s_2, \ldots, s_n)=s_1s_2\ldots s_n$.

For example, $\text{concatenation}(a, bc, def)=abcdef$. Trivially,

$$\text{len}(\text{concatenation}(s_1, s_2, \ldots, s_n)) = \sum_{i=1}^{n} \text{len}(s_i).$$

Def-4: Let $s$ be any string and $n$ a natural number then $\text{power}(s, n)=\text{concatenation}(s, s, \ldots, s)$. If $n=0$ then $\text{power}(s, n)=\varepsilon$ $n$ times.

For example, $\text{power}(ab, 3)=ababab$

Def-5: If $s_3=s_1s_2$ is a string then $s_1$ (non-empty string) is called a head (or prefix) and $s_2$ (non-empty string) a tail (or suffix) of $s_3$. 
Def-6: The operation delete deletes a given head from a string, i.e., if \( s_3 \) is a string such that \( s_3 = s_1 s_2 \) then \( \text{delete}(s_3, s_1) = s_2 \).

For example, if \( s = abc \) then \( \text{delete}(s, ab) = c \).

Def-7: The operation first gives the first symbol from a non-empty string.

For example, if \( s = abc \) then \( \text{first}(s) = a \).

2.1. Auxiliary functions needed in data encoding

The purpose of auxiliary functions is to obtain information for the actual encoding of logical data units. When we later on shall define attributes in our formal specification all the auxiliary functions are constructed in the same way. We represent the auxiliary functions in a standard way or as a set consisting of ordered pairs. In other words our auxiliary functions are based on binary relations. We define a function in terms of a binary relation as follows. If \( f \) is a binary relation on \( A \times B \) and \( x \) is in this relation for each \( a \in A \) there is exactly one \( b \in B \) such that \( \langle a, b \rangle \in f \) then the binary relation \( f \) is called a function from \( A \) to \( B \). With the notational convention \( \text{dom}(f) \) we mean the set \( \{ a \mid \langle a, b \rangle \in f \} \). Symbolically, we shall write \( f : A \rightarrow B \) and instead of \( \langle a, b \rangle \in f \) we shall write \( f(a) = b \).

Usually the formal specification methods based on specific data types provide expressions in terms of which it is possible to define and manipulate this kind of functions as abstract objects. For example, in VDM (Vienna Development Method, based on denotational semantics) we can define abstract objects of type map (one of the composite data types of VDM [10]) which is analogous to our way of representing functions.

In data encoding we need the following four auxiliary functions: \( \sigma \)-table (character code table), \( \lambda \)-table, \( \sigma \)-table and \( \tau \)-table (record type code table). Next we consider these functions and their role in data encoding in detail. The function \( \sigma \)-table is needed to describe the character code used in the hw/sw environment. From the view point of data reformatting it is important to observe that in any computer system the stored data can be represented on the basis of some character code. In
a character code there is for each symbol its unique bit string representation. Usually, in business data processing machines, eight bits (one byte) are used to represent symbols. ASCII and EBCDIC are the two most widely used character codes.

Also the numeric data can be represented on the basis of a character code. Numeric data are said to be in zoned decimal form ([5], [21]) or sometimes briefly decimal (e.g. [6]) if they are encoded with a certain character code. We have as a starting point in data encoding that all numeric data are represented in the zoned decimal form. From the viewpoint of data reformatting this decision has the following consequences:

- Some character code is available in any hw/sw environment and thus the zoned decimal form for numeric data. In other words, the zoned form representation is a common representation for numeric data in different hw/sw environments. This representation is capable of guaranteeing the unique interpretation of numeric data during data reformatting process which is necessary for us.

- We can assume that in any hw/sw environment numeric data in the zoned form can be transformed into any other representation of numeric data and vice versa. If the source data are not in zoned decimal form we can transform them into this form before the actual conversion process. In addition, the load utilities of many dbms's require or at least permit that the numeric data in their input files are represented in the zoned form.

- By representing numeric data for data conversion process in zoned decimal form we avoid the manipulation of many different representations, i.e. the data reformatting process is simplified more manageable.

- It is typical that computers cannot perform arithmetic operations on zoned decimal numbers. This feature has no matter because during the data reformatting process we do not perform arithmetic operations but only transfer stored data between hw/sw environments.

Let SYMB be a set of the symbols in a character code and B-STRING a set of the corresponding bit strings in this character code. In this case, a character code is the function which we denote by \( cc-table \) : SYMB \( \rightarrow \) B-STRING. If \( a \) is a symbol in SYMB, i.e. \( a \in \text{SYMB} \), then \( cc-table (a) \) gives the bit string representation of \( a \).
For example, if we describe the EBCDIC character code in the way above then SYMB={SPACE,'...,='A,...,Z,0,...,9}, B-STRING={01000000,01001011,...,01111110,...,11101001, 11110000,...,11111110} and co-table={<SPACE,01000000>, <',01001011>,...,<01111110>,<A,11000001>,..., 01111110},<00,11100000>,...,<9,11111110>}. Because it is an explicit symbol we wrote it explicitly above. For example, in this case co-table(=)=01111110.

The function length-table expresses how many characters (bytes) have been allocated for storing instances of different data items. Typically, users are responsible for specifying how much space is needed to represent the values of a data item. In this paper the data structures being encoded are represented as six-tuples (r,x,RN,DN,I_x,f_x) or as dse's. Because each data item has a length we can define the function length-table as follows length-table:DN+I where I is the set of positive integers. Let a be any data item name such that a ∈ DN. Now,length-table(a) gives the number of characters (bytes) allocated to the values of the data item a.

In data encoding it is important to know the order in which the data are stored. In this paper the function sort-table is needed to sort records. The function is defined as follows sort-table:DN+{ASC,DESC}; where DN is the subset of those data items in a dse description (r,x,RN,DN,I_x,f_x) or DNS=DN, which are used in sorting. ASC and DESC are indicators to express the sorting order either ascending (ASC) or descending (DESC). If a is any data item name in the set DN or a ∈ DN then sort-table(a) yields the sorting indicator associated with a.

Very often in the DBMS's based on such a data model, which permits the manipulation of many record types, each record type has its unique record type code. Record type codes in records are used to identify uniquely the record type whose instance the record is. In this paper we encode hierarchical data bases which contain instances of many different record types. We suppose that in each stored record there is also the record type code of the corresponding record type.

The function rto-table is needed to express the record type codes associated with the record types of data structures. The function rto-table is defined as follows rto-table:RN+R-CODES where RN is a set of record type names in a dse description of a hierarchical data structure; R-CODES is a set of record type codes such
that \( |R-CODES| = |RN| \). In the context of our specification below we shall check that each record type has its unique code. If \( r \) is any record type name in \( RN \) or \( RN \) then \( r = \text{table}(r) \) expresses the record type code associated with \( r \).

3. THE ENCODING OF LOGICAL DATA UNITS

In [NIE-1], [NIE-2], [NIE-3] we have considered those logical data units of which flat files and hierarchical data bases consist. We suppose that the reader knows these logical data units and their description as dse's. In this chapter we define functions in terms of which instances of different logical data units can be encoded. Conceptually our encoding functions are analogous to the collapsing operation of SDDTG (Stored-Data Definition and Translation Task Group) [35].

3.1 The encoding of data items

In our dse description data items can be type of int, real or char. In this paper data items of different types are encoded in a quite analogous way due to the fact that during the conversion process we represent all numeric data in the zoned decimal form.

3.1.1 The encoding of non-numeric data item values

Now, we consider the encoding of data items which are of type char. In this encoding we need the function \( ao-table \) above to encode the symbols forming the character string to be encoded. We must know how many characters (bytes) have been allocated to storing the data item values and what is the padding character used by the hw/sw environment. A hw/sw environment uses some padding character to pad unused space in the case that the length of a data item value is smaller than the allocated space. Furthermore, we need the justification factor which expresses how a data item value is justified with respect to the allocated space, i.e. a justification factor can be right or left. The length of a data item is specified by the user whereas the other aspects are features of the hw/sw environment.

For the definition of the function \( ao-encoding \), which maps a value of a data item of type char into a bit string representation, we need the function \( v-encoding \). In terms of this function we replace the symbols of a data item value (the argument \( v \)) with their bit string representations in a character code (the argument \( ao-table \)).
\textit{v-encoding}(v, \textit{cc-table})

\[
\text{catenation}(\textit{cc-table}(\text{first}(v)), \textit{v-encoding}
\begin{cases}
\text{delete(\text{first}(v),v,\textit{cc-table}), if \text{len}(v)>1} \\
\text{\textit{cc-table}(v), if \text{len}(v)=1}
\end{cases}
\]

The function \textit{ch-encoding} has the following arguments (or parameters).
- \textit{dse} or the six-tuple description \((r,X,\emptyset,\text{DN},I_X,I_X')\) of a data item. In this case \(X\text{-char}\).
- the function \textit{cc-table}.
- \textit{length} is a positive integer which expresses the allocated space.
- \textit{p-char} expresses the padding character which is used to pad the unused space \((=\text{length-}\text{len}(\sigma_1(\text{dse}))\)
- \textit{justif} expresses the justification factor \((=\text{left, right})\).

\textit{ch-encoding}(\textit{dse}, \textit{cc-table}, \textit{length}, \textit{p-char}, \textit{justif})

\[
\text{catenation}(\textit{v-encoding}(\sigma_1(\textit{dse}), \textit{cc-table}), \textit{power}(\textit{cc-table}
\begin{cases}
\text{\textit{p-char},\textit{length-}\text{len}(\sigma_1(\textit{dse})), if justif=\text{left}} \\
\text{\textit{catenation}(\textit{power}(\textit{cc-table}(\textit{p-char}),\textit{length-}\text{len}(\sigma_1(\textit{dse}))),} \\
\text{\textit{v-encoding}(\sigma_1(\textit{dse}),\textit{cc-table})), if justif=\text{right}}
\end{cases}
\]

In the \textit{ch-encoding} function above we assume that the length of a data item being encoded is smaller or equal than the allocated space, i.e. \(\text{len}(\sigma_1(\textit{dse}))<\text{length}\). This means that we do not need to truncate data item values.

Example 1. Let us assume the following hw/sw environment:
- It uses the EBCDIC character code.
- The padding character is space.
- The justification factor is left.

In this environment we store instances of the data item \textbf{NAME} for which 8 characters (bytes) have been allocated. Let TIMO be its one instance, i.e. it has the following dse description \{(TIMO, char, $\emptyset$, \{NAME\}, $\{<1>\}$, $f$(NAME)=$\langle1\rangle$)}
(we denote this dse description by dse.). In this case
the function ch-encoding(dse, co-table, 8, space, left),
where co-table describes the EBCDIC character code, yields
the following bit string

```
111000111100100111010011011001000000010000001000000
T I M O space space space
```

- Example 2. We have the same example as above but the
  following hw/sw environment:

  - It uses the eight bits ASCII character code (even
    parity)
  - The padding character is space.
  - The justification factor is right.

In this environment the function ch-encoding(dse, 
co-table, 8, space, right), where co-table describes the 
ASCII character code, gives the following bit string

```
10100000101000001010000011010100110010011001101
space space space space T I M
```

- 0

3.1.2. The encoding of numeric data item values

Numeric data items are either of type int or real. There are numerous ways to encode numeric data item 
volumes. They may be encoded as zoned decimal, packed, 
binary or floating point forms. Likewise many data 
compression techniques have been developed in terms of 
which data can be represented effectively.

From our view point it is not essential how effective 
the representations of data are during the data conversion 
process. Instead it is rather more important that the 
representation of data preserves the correct inter-
pretation of data during the whole conversion process.
The representation of data must be such that to and from this representation it is possible to transform other representations of numeric data in the source and target systems.

We have chosen above the zoned decimal form for representing numeric data during the conversion process. In addition, we associate with the encoding of numeric data the following features:

- The padding character is zero.
- The justification factor is right.
- Numeric data can be signed or unsigned. In the case that they are signed we use explicitly the sign indicator '+' or '-' in the encoding of numeric data item values (this corresponds to the EXPLICIT specification in the CODASYL STORAGE SCHEMA proposal [6]). To abbreviate our discussion we assume that the length specification of a numeric data item includes the space for the sign indicator, too.

- We deal only with the fixed point representation of reals. In the encoding of reals we encode the decimal point, too. Often in the data encoding the decimal point is not encoded at all but the user is responsible for expressing the assumed decimal point in his description. If the load utility of a data base or a file presupposes that the decimal point is not encoded explicitly in its input file it is a straightforward task to eliminate decimal points from the input file. This means that we encode integers and reals similarly in this paper.

The function `num-encoding` transforms a value of the numeric data item into the zoned form representation. The arguments of the function are

- `dse` or the six-tuple description \((r,X,\phi,DN,1, f_X)\) of a numeric data item, i.e. \(c_2(dse) = \text{int or real}\).
- \(c_2\) is the function above.
- \(\text{length}\) is a positive integer which expresses the allocated space in characters (bytes) including the space for the sign indicator and the decimal point.
num-encoding(dce, cc-table, length)

catenation(cc-table(first(σ₁(dce))), power(cc-table(0), length-len(σ₁(dce))), v-encoding(delete(first(σ₁(dce)), v₁(dce))))

otherwise / * the value is unsigned */

Let us suppose that we have a sw/hw environment using the EBCDIC character code. Let -125 be the numeric data item value for which 5 characters (bytes) have been allocated (containing the sign indicator). In this case the function num-encoding((-125, int,...), cc-table, 5) yields the following bit string: 0110000011110000111100011111001011110101.

The zoned decimal representation of numeric data is very space consuming and therefore in computer systems it is usually possible to represent numeric data in packed decimal forms. For example, in the IBM environment [21] each decimal number is represented with 4 bits in the packed decimal form (this corresponds to the PACKED DECIMAL option in the CODASYL DSDL [6]). Thus two decimal numbers can be packed into one byte (=8 bits).

3.2. The encoding of records

In principle, data item values and records can be stored in fixed or variable length. The variable length can be used if we do not wish to use a fixed prespecified space for storing values of a data item or we do not require a regular record structure. The variable length can be realized in various way. For example, we can use some characters (delimiter data) in certain positions (postfix, prefix) to delimit data item values from each other [35].

The fixed record length is realized by placing the encoded data item values in a sequential order. In this paper we require that some fixed length has been allocated to each
data item. We also suppose that the instances of the data items belonging to a record type are mandatory in each record. In other words, in this paper we encode fixed length records.

It is worth noting, that the data base trees (data base records in IMS terminology [38]) of a hierarchical data base have no fixed length. Instead, all instances of each record type have the same length. This is due to the fact that there can be a varying number of records of the same type in different data base trees.

From the viewpoint of the encoding of records the concatenation order of the encoded data item values is essential. In the dse description of a record we have a unique index for each data item belonging to a record. The encoded data item values are stored in the order indicated by indices. Let \( (\text{JONES}, 33, 2050.50), \text{char} \times \text{int} \times \text{real} \times \text{char}, \{\text{NAME, AGE, SALARY}\}, \{(1), (2), (3)\}, \{\text{f(NAME)=<1>, f(AGE)=<2>, f(SALARY)=<3>}\} \) be the dse description of our sample record.

The encoding of this sample record or its contiguous bit string representation consists of the bit string representation of the data item value JONES which is immediately followed by the bit string representations of the data item values 33 and 2050.50.

Next we define the function \( \text{record-encoding} \) which encodes a record into a contiguous bit string representation. The function has the following arguments:

- \( dse \) or the six-tuple description of a record.
- \( as-table \) is the function above.
- \( length-table \) is the function introduced above.
- \( p-char \) expresses the padding character above.
- \( justif \) expresses the justification factor above.

In the definition of the function we denote the naming function of \( dse \) by \( f_{dse} \).
record-encoding(dse, cc-table, length-table, p-char, justif)

\[
\text{catenation}(\text{ch-encoding}(dse[<1>], cc-table, length-table(f_{dse}(<1>)), p-char, justif), record-encoding(dse[\sigma_2(dse)-<1>], cc-table, length-table, p-char, justif))
\]

, if \(|\sigma_4(dse)|=1\Lambda_2(dse)<1>\#char

\[
\text{catenation}(\text{num-encoding}(dse[<1>], cc-table, length-table(f_{dse}^{-1}(1))), record-encoding(dse[\sigma_2(dse)<1>], cc-table, length-table, p-char, justif))
\]

, if \(|\sigma_4(dse)|=1\Lambda_2(dse)<1>\#char

\[
\text{ch-encoding}(dse, cc-table, length-table(f_{dse}(<1>)), p-char, justif)
\]

, if \(|\sigma_4(dse)|=1\Lambda_2(dse)=\text{char}

\[
\text{num-encoding}(dse, cc-table, length-table(f_{dse}(<1>)))
\]

, if \(|\sigma_4(dse)|=1\Lambda_2(dse)<1>\#char

The function record-encoding encodes data item values in the order indicated by the indices and concatenates them into a contiguous bit string.

3.3. The encoding of flat files and hierarchical data bases

The storing order of records is essential in the encoding of flat files and hierarchical data bases. Although the dse descriptions of data structures define precisely the structural relationships among logical data units, they do not define the storing order among instances of the logical data units (the first component of a dse description). From the viewpoint of encoding flat files and hierarchical data bases we can distinguish the following two essential orders:

1) The order of the records of the same type.
2) The order of the records of different types (this is only concerned with the hierarchical data model).
In general, we cannot assume that records can be sorted in the loading phase where a physical file organized according to the flat file data model or a physical hierarchical database is created by some load utility. For example, some load utilities of IMS require that their input files have been sorted before the actual load. To abbreviate our discussion we assume that the records belonging to a certain record type are sorted to the ascending (ASC) or descending (DESC) order on the basis of the values of one data item. Sometimes this is a restriction also in practice. For example in IMS. One of the ideas in the hierarchical organization of data is that at a certain hierarchy level we would not need to sort data on the basis of many data items.

We need the function \texttt{choose} to sort records of the same type. This function has two arguments:
- \texttt{dse} or the six-tuple description of the data structure being sorted.
- \texttt{sort-table} is the function above.

In general, the sorting of hierarchical data bases is much more complex than the sorting of flat files. This is due to the fact that sorting may be needed at several hierarchical levels. Furthermore, sorting of hierarchical data bases requires sorting within record groups. A record group consists of those records of the same type which depend on the same record in a hierarchical data base (see the class \texttt{Group} in our abstract syntax in [NIE-3]). It is typical of the \texttt{dse} description of any record group that its relation type is \texttt{P(\xi)} and the index associated with the record type name used in constructing the record group.

The function \texttt{choose} returns one record from a flat file, one data base tree from a hierarchical data base or a hierarchical substructure used in constructing a group. The selection is based on the sorting order indicated by \texttt{sort-table}. In other words, the \texttt{dse} argument of the function \texttt{choose} is always of type \texttt{P(\xi)} and the function gives always a data structure element with type \texttt{\xi}, i.e. it belongs to \texttt{\sim dse}.

In the definition of the function \texttt{choose} we shall apply the function \texttt{r-content} (see [NIE-1], [NIE-2]) which returns the indices of the data items of a record type (whose index is the argument of this function). When we shall apply the function we shall consider sorting of the instances (records) of that record type with which the
index \( l \) is associated. Thus the expression

\[
\bigcup_{i \in r\text{-}content(l)} \mathop{\cap} \mathsf{dom} \mathsf{content}(l) \setminus (\mathsf{f}_{dse}(i) \setminus \mathsf{dom} \mathsf{content}(l))
\]

contains the name of that data item whose values are used to sort the dse. Due to the fact that we allow record sorting on the basis of values of only one data item, the result of the above intersection is either a set consisting of one data item name or the empty set.

If the intersection above is empty it means that the records of this record type are not sorted. In this case we choose any instance of this record type. If there are several records with the same value of the sorting data item we select indeterministically any one of them. We shall apply the indeterministic selection function \( \mathcal{N}(S) \) in the definition of the function \( \text{choose} \). It selects indeterministically one element from a non-empty set \( S \).

In the definition of the function \( \text{choose} \) below the naming function of the dse is denoted by \( \mathsf{f}_{dse} \):

\[
\text{choose}(dse, \mathsf{sort}-\text{table})
\]

\[
\mathcal{N}([t \mid t \in dse])
\]

\[
= \begin{cases} 
\mathcal{N}([t \mid t \in dse]) & \text{if } \bigcup_{i \in r\text{-}content(l)} \mathop{\cap} \mathsf{dom} \mathsf{content}(l) \setminus (\mathsf{f}_{dse}(i) \setminus \mathsf{dom} \mathsf{content}(l)) = \emptyset \\
/* \text{no sorting} */ & \\
\mathcal{N}([t \mid t \in dse: \forall t_1 (t_1 \neq t \in dse: \sigma_1(t_1[f_{t_1}(x)]) \geq \sigma_1(t[f_{t}(x)]))]) & \text{where } x \in \bigcup_{i \in r\text{-}content(l)} \mathsf{f}_{dse}(i) \setminus \mathsf{dom} \mathsf{content}(l) \\
& \mathsf{f}_{t_1} \text{ and } \mathsf{f}_{t} \text{ are the naming functions associated} \\
& \text{with } t_1 \text{ and } t \\
& \text{if } \bigcup_{i \in r\text{-}content(l)} \mathop{\cap} \mathsf{dom} \mathsf{content}(l) \setminus (\mathsf{f}_{dse}(i) \setminus \mathsf{dom} \mathsf{content}(l)) = \emptyset \text{sort-table}(x) = \text{ASC} \\
& /* \text{the ascending sorting} */
\end{cases}
\]
\[ \Pi \{ t \mid t \in \text{dse} : \forall t_1 \in \text{dse}: \alpha (t_1 [f_{t_1}(x)]) \leq \alpha (t[f_{t_1}(x)]) \} \]

where \( x \in \bigcup_{i \in \text{r-content}(t)} \text{ndom}(\text{sort-table}) \);  

\( f_{t_1} \) and \( f_t \) are the naming functions associated with \( t_1 \) and \( t \), if \( \bigcup_{i \in \text{r-content}(t)} \text{ndom}(\text{sort-table}) \neq 1 \land \text{sort-table}(x) = \text{DESC} \)

/*the descending sorting*/

Next we define the function flatfile-encoding in terms of which we encode flat files. In its definition we shall apply the functions choose and record-encoding. The function selects records one after another from a flat file in a desired order and encodes them into a contiguous bit string. If \( \text{dse} \) (one of the arguments of the function) describes a flat file then in the result of the expression \( f_{\text{dse}}(t) \): \( \sim \text{dse-choose(dse, sort-table)} \) there is one record less than in the original \( \text{dse} \). The arguments of the function are:

- \( \text{dse} \) or the six-tuple description of a flat file, i.e.
  \[ |\phi_3(\text{dse})| = 1 \]
- \( \text{sort-table} \),
- \( \text{length-table} \),
- \( \text{co-table} \),
- \( p\text{-char} \) and \( j\text{-factor} \) as above.

\[ \text{flatfile-encoding}(\text{dse, sort-table, length-table, co-table, p\text{-char, j-factor})} \]

\[ \text{catenation(record-encoding(choose(dse, sort-table), co-table, length-table, p\text{-char, j-factor}), flatfile-encoding(f^{-1}(t)) : dse} \]

\[ \sim \text{dse-choose(dse, sort-table), sort-table, length-table, co-table, p\text{-char, j-factor})} \]

, if \( \sim \text{dse} \neq 1 /* \text{records being encoded left */} \]

\[ \text{record-encoding(t, co-table, length-table, p\text{-char, j-factor})} \]

where \( t \in \text{dse} \)

, if \( t \in \text{dse} = 1 /* \text{the last record is encoded} */
The result of the function flatfile-encoding is always a bit string representation of some sequential file which as such is one physical organization alternative of the flat file data model. This sequential file is also the input file for creating other organization alternatives (e.g. an indexed sequential file).

The encoding of hierarchical data bases is much more complex because we must decide the order in which the instances of different record types are stored. There are several methods of systematic ordering and storing of the records of a hierarchical data base. Next we consider those features which have an essential role in the encoding of hierarchical data bases.

- Each record type in a hierarchical data base has its unique record type code which is stored in the instances or records of this record type. During the processing of records this information can be used to identify the record type whose instance a record being processed is. This is a very typical feature in large commercial dbms's which manipulate data structures consisting of many different record types. Sometimes, instead of record type codes, counters are used in the processing of records of hierarchical data structures. The counter in a record expresses the number of records (sometimes they are called a repeating group) depending immediately on a certain record. Usually counters are used in cases containing only a few hierarchy levels.

- Preorder (it is also called the top-down left-to-right ordering [26] or the hierarchical serial ordering [18]) is a favorite method of arranging records of a hierarchical data base [18]. Preorder, inorder, postorder (see e.g. [2]) are the well-known basic orderings to store and traverse hierarchical data. In fact, a hierarchical data base is a forest which can be stored and traversed on the basis of these basic orderings in an analogous way to using them to traverse the nodes of an ordered tree. In this paper a hierarchical data base is stored in preorder form one data base tree after another.

In [NIE-3] we have considered an example on how a set of flat files can be restructured into a hierarchical data base. In Fig. 3 we illustrate one instance or data base tree according to the target schema (see Fig. 5 in [NIE-3]) of our sample case.
Fig. 3. The sample data base tree

Other storing principles (see e.g. [8]) have also been developed for hierarchical data bases. They may be in some respects more effective than the basic storing principles. However, in the existing dbms's the basic storing principles, especially preorder, have the central role. Our encoding function of hierarchically data bases is based on the preorder policy. It is obvious that we can define encoding functions for other basic storing principles in an analogous way.

Next we analyze those different encoding cases which make up our encoding function of hierarchical data bases. We shall pay attention to the type of dse being encoded (the argument dse of our encoding function below).

1) A hierarchical data base or its some part consists of substructures of the same type. This case can be illustrated graphically as follows:
When this kind of a structure is described as a dse then the type of the dse is \( P(\xi) \) where the type of each substructure is \( \xi \). For example, a hierarchical data base represents this kind of an encoding case because it consists of data base trees of the same type. In our sample data base tree above those records, to which we refer with the numbers (4) and (6), form with their substructures such a structure. In this case we apply the above function \( \text{choose} \) which selects one from the substructures in a desired order. Substructures are selected one after another until all substructures have been encoded. In our encoding function we divide the manipulation into two cases:

- We have many substructures to be chosen. In terms of the argument dse this condition can be expressed with \(|\neg \text{dse}| > 1\)
- We have only one substructure left or \(|\neg \text{dse}| = 1\).

2) When we select substructures in the case 1 they have the following structure.

As a dse description this hierarchical structure is of type \( D_1 \times D_2 \times \ldots \times D_n \times P(\xi_1) \times \ldots \times P(\xi_k) \) where \( D_1, D_2, \ldots, D_n \) are some primitive types which are associated with the data items of the root record, whereas \( P(\xi_1), P(\xi_2), \ldots, P(\xi_k) \) are associated with substructures depending on the
root record. For example, our sample data base tree is of this type. The argument dse of this hierarchical structure satisfies the following condition $f_{dse}^{-1}(<1>) \in \sigma_4(dse) \cap \sigma_3(dse) \neq \emptyset$ where $f_{dse}$ denotes the naming function of dse.

In this case we encode first the root record and after that the substructures depending on it. The root record can be expressed by dse[A] where

$$A = \{ i | i \in \sigma_5(dse) \wedge \text{elem}(i) = 1 \wedge f_{dse}^{-1}(i) \in \sigma_4(dse) \}.$$ 

Correspondingly, the hierarchical structure associated with substructures depending on the root record can be expressed by dse[B] where

$$B = \{ i | i \in \sigma_5(dse) \wedge f_{dse}^{-1}(i) \in \sigma_4(dse) \wedge i \notin A \}.$$ 

3) When we remove the root record from the structure according to case 2 then the remaining structure can be illustrated as follows.

![Diagram]

In this case the type of the argument dse or $\sigma_5(dse) = P(\xi_1) \times P(\xi_2) \times \ldots \times P(\xi_k)$. In the dse description the index $<1>$ is associated with the structure $P(\xi_1)$, the index $<2>$ with the structure $P(\xi_2)$ and so on the index $<k>$ with the structure $P(\xi_k)$. In this case we assume that there are at least two substructures. If we would have only one substructure then it would correspond to the case 1 above. In preorder we encode the substructures $P(\xi_1), P(\xi_2), \ldots, P(\xi_k)$ in the order indicated by their indices. The first substructure is separated from the argument dse by projection dse[I] where

$$I = \{ <1, \xi> | <1, \xi> \in \sigma_5(dse) \wedge f_{dse}^{-1}(<1, \xi>) \in \sigma_4(dse) \}.$$
Other substructures are included in the expression $dse[J]$

where $J=\{<j,\xi>|<j,\xi>\in\sigma_5(dse)\land j\neq 1\land f_{dse}^{-1}(<j,\xi>)\in\sigma_4(dse)\}$.

Our sample data base tree without the root record represents this structure alternative.

4) If the structure according to the case 2 has no substructures then we have a leaf record. In the $dse$ description of any leaf record the type of $dse$ or $\sigma_3(dse)=D_1 \times D_2 \times \ldots \times D_n$ where $D_1, D_2, \ldots, D_n$ are some primitive types. Likewise it holds for the $dse$ description of a leaf record that $\sigma_3(dse)\neq \emptyset$. For example, in our sample data base tree the records (2), (3), (5), (7) and (8) are leaf records.

We shall define the encoding of hierarchical data bases recursively with the $HDB$-encoding function based on the structure alternatives analyzed above. Contrary to the encoding of flat files we need record type codes, too, in the encoding of hierarchical data bases. We have introduced above the function $rta-table$ which defines a unique record type code associated with a record type. The function $HDB$-encoding has the following arguments.

- $dse$ or a hierarchical data structure. The naming function of $dse$ shall be denoted by $f_{dse}$.

- $sort-table, rta-table, length-table, ca-table, p-char$ and $justif$ from above.
HDB-encoding(dse, sort-table, rto-table, length-table, co-table, p-char, justif)

catenation(v-encoding(rto-table(\(\xi^{-1}(<1>)\)), co-table),
dse
HDB-encoding(choose(dse, sort-table), sort-table, rto-table,
-1 length-table, co-table, p-char, justif), HDB-encoding(
\(f_{dse}(<1>)\) - dse-choose(dse, sort-table), sort-table, rto-
table, length-table, co-table, p-char, justif))
, if \(\sigma_2(dse)\) is type of \(P(\xi)\land -dse>1\)
/* The case 1 above */

catenation(v-encoding(rto-table(\(f_{dse}(<1>)\)), co-table)
HDB-encoding(t, sort-table, rto-table, length-table, co-table,
p-char, justif))
where tc-dse
, if \(\sigma_2(dse)\) is of type \(P(\xi)\land -dse\leq 1\)
/* The case 1 above */

catenation(record-encoding(dse[A], co-table, length-table,
p-char, justif), HDB-encoding(dse[B], sort-table, rto-table,
length-table, co-table, p-char, justif))
where \(A = \{i | \exists \sigma_4(dse) \land elem(i) = 1 \land \neg dse(i) \in \sigma_4(dse)\}\)

\[B = \{i | \exists \sigma_5(dse) \land \neg dse(i) \in \sigma_4(dse) \land i \notin A\}\]
, if \(f_{dse}(<1>) \notin \sigma_4(dse) \land \sigma_3(dse) \neq \emptyset\) /* The case 2 above */

catenation(HDB-encoding(dse[I], sort-table, rto-table,
length-table, co-table, p-char, justif), HDB-encoding(dse[J],
sort-table, rto-table, length-table, co-table, p-char, justif))
where \(I = \{<1, \xi> | \exists \xi \in \sigma_5(dse) \land dse(<1, \xi>) \in \sigma_4(dse)\}\)

\[J = \{<j, \xi> | \exists j, \xi \in \sigma_5(dse) \land j \neq \neg dse(<j, \xi>) \in \sigma_4(dse)\}\]
, if \(\sigma_2(dse)\) is of type \(P(\xi_1) \land \ldots \land P(\xi_2)\)
/* The case 3 above */
record-encoding(dse, co-table, length-table, p-char, justif)
, if \(\sigma_3(dse) = \emptyset\) /* The case 4 above */
The **HDB**-encoding function is able, for example, to encode such hierarchically input files from which physical IMS data bases can be loaded with IMS utilities. Next we consider the encoding of IMS data bases. An physical IMS data base is described by a *Data Base Description* (DBD). A DBD consists of several statements. These statements have been dealt with widely the standard data base literature (see e.g. [13], [14], [38]) and therefore we do not consider them in detail here. We also use only those features of the statements which are relevant from the viewpoint of our example. Our sample data base consists of data base trees (data base records in the IMS terminology) which structurally resemble our sample data base tree described in Fig. 3. Our concepts data item and record type correspond to the concepts field and segment type in the DBD below. Let our sample data base have the following physical DBD.

```
DBD
   NAME=SAMPLE DB, ACCESS=HISAM/* HIERARCHICAL
   INDEXED SEQUENTIAL ACCESS METHOD (HISAM) IS
   ONE OF THE PHYSICAL ORGANIZATION OPTIONS
   AVAILABLE IN IMS */

DATASET ...

SEGM
   NAME=PRODUCTS, BYTES=28
FIELD
   NAME=(P-CODE,SEQ.U), BYTES=3, START=1
FIELD
   NAME=P-NAME, BYTES=20, START=4
FIELD
   NAME=PRICE, BYTES=5, START=24

SEGM
   NAME=SALESMEN, PARENT=PRODUCTS, BYTES=29
FIELD
   NAME=S-NAME, BYTES=21, START=1
FIELD
   NAME=STATE, BYTES=8, START=22

SEGM
   NAME=PARTS, PARENT=PRODUCTS, BYTES=25
FIELD
   NAME=(PA-CODE,SEQ.U), BYTES=4, START=1
FIELD
   NAME=PA-NAME, BYTES=18, START=5
FIELD
   NAME=QTY, BYTES=3, START=23

SEGM
   NAME=SUPPLIER, PARENT=PARTS, BYTES=51
FIELD
   NAME=SU-NAME, BYTES=20, START=1
FIELD
   NAME=SU-ADDRESS, BYTES=30, START=22
```

These statements indicate the end of the DBD definition.

> In IMS each physical data base is defined, together with its mapping to storage, by a DBD. It is worth noting that the order of the segment types in a DBD expresses preorder storage. It is typical of an IMS data base that its segment
(record) type codes can vary from 001 to 255, i.e. an IMS data base can consist of at most 255 segment types. IMS load utilities require that the segments of their input files contain also the segment type codes, which are represented as zoned decimal format [22]. The numbering of segment codes is based on their order in a DBO so that the root segment type has the segment type code 001 and so on. In our sample data base the segment types PRODUCTS, SALES MEN, PARTS and SUPPLIER have the segment type codes 001, 002, 003 and 004, respectively.

In IMS at most one sequence (key) field may be specified per segment type. Specification of a sequence field is optional and it may be either unique (U) or nonunique (M). M indicates that duplicate values of the sequence field can occur. The choose function applied in the RDB-encoding function is able to deal with both cases. The FIELD statement can also be used to specify optionally the data type of the field. Options are X(for hexadecimal data) P(for packed decimal data) and C(for character data). We did not use these options in our DDB so in this case the default value is C. This corresponds to our way of encoding data item values.

Next we consider the bit string representation of the input file (or source) from which the data base specified in our DDB is loaded. Let us suppose that the loading occurs in a typical DDB environment. In this case the bit string representation of our input file is achieved with the function RDB-encoding(dse, sort-table, rto-table, length-table, ac-table, space, left) where

- dse is the dse description of our sample data base or the following six-tuple (by"...
" we mean other data base trees in the sample data base).

```
([{...<111,CHAIN SAW,200,<{SMITH,1050.20},<DALE,980.30}>,
  <1232,STEEL CHAIN-200,200,<{HORN,1,A-STREET-1}>,<1234,
  S-MOTOR-22,20,<{DO DD,3,B-STREET-2},<TAYLOR,1,C-STREET-3}]>}
  ,P(char|char)xP(char|xreal)|P(int|char)xP(int|char|xreal)|P(int|char|xreal)|P(int|char|xreal)|P(int|char)xP(int|char|xreal))},
{PRODUCTS,SALES MEN, PARTS, SUPPLIER},{P-CODE,P-NAME,
PRICE, S-NAME, SALARY, PA-CODE, PA-NAME, QTY, SU-NAME, SU-TIME,
ADDRESS},{<1>,<1,1>,<1,2>,<1,3>,<1,4>,<1,5>,<1,4,1>,<1,4,2>,
<1,5,1>,<1,5,2>,<1,5,3>,<1,5,4>,<1,5,4,1>,<1,5,4,2>,<1,5,4,3}]
{f(PRODUCTS)=<1,f(P-CODE)=<1,1>,f(P-NAME)=<1,2>,f(PRICE)=
<1,3>,f(SALES MEN)=<1,4>,f(PARTS)=<1,5>,f(S-NAME)=<1,4,1>,
 f(SALARY)=<1,4,2>,f(PA-CODE)=<1,5,1>,f(PA-NAME)=<1,5,2>,
 f(QTY)=<1,5,3>,f(SUPPLIER)=<1,5,4>,f(SU-NAME)=<1,5,4,1>,
 f(SU-TIME)=<1,5,4,2>,f(ADDRESS)=<1,5,4,3})
```
In the description of the corresponding bit string below we use the notation deviation \( B(\text{value}) \) which means the encoded value or the bit string representation of value; \( i \) expresses the number of characters (bytes) used in the encoding of value. The EBCDIC character encoding yields the following bit string representation for our sample data base:

\[
\begin{align*}
\text{...} & B(001)B(111)B(\text{CHAIN SAW})B(200)B(002)B(\text{SMITH})B(1050.20) \\
& B(002)B(\text{DALE})B(980.30)B(003)B(1232)B(\text{STEEL CHAIN-22})B(200) \\
& B(004)B(\text{HORN})B(\text{B(A-STREET-1)})B(003)B(1234)B(\text{S-MOTOR-22}) \\
& B(004)B(\text{DOCTOR})B(3)B(\text{B-STREET-2})B(004)B(\text{TAYLOR})B(1) \\
& \text{...} \\
& B(C\text{-STREET-3}) \ldots
\end{align*}
\]

When we consider the content of our input file we note that all data, except for record type codes, are user data. It is typical of the loading phase that in it a large amount of control information specific to a dbms are created. Next we characterize briefly those tasks which are performed in the loading phase of our sample data base.

- Two physical files are created. The one is an indexed sequential file which contains the root segments and as many subordinate segments of the root segments as can fit into the fixed length physical record chosen by a data base administrator. The other physical file contains those subordinate segments that do not fit into the indexed sequential file.

- The storing policy of segments is such that no segment is split across physical records. This means that there may be unused space at the end of a physical record.

- In IMS a prefix area is created per each stored segment. It contains control information such as record type code, deletion flag, pointers and so on.
The purpose of our example above was to demonstrate that the input file required by a load utility is a meaningful interface between the conversion process and the software available in a certain system. Likewise, we demonstrated that the features of the encoding functions defined here are sufficient to create the bit string representation required by IMS load utilities from a dse description. A dse description of a hierarchical data base itself is general and independent of any dbms.

4. SPECIFICATION OF DATA REFORMATTING ASPECTS IN DATA CONVERSION

The encoding functions introduced in chapter 3 define bit string representations for flat files and hierarchical data bases. We shall apply these functions in the specification of data reformatting aspects in data conversion. In [NIE-3] we specified the data restructuring aspects. The specification of this paper and the specification given in [NIE-3] together build up one specification for general conversion of flat files and hierarchical data bases.

It is important to note that we extend the specification introduced in [NIE-3]. This means that we have available all the attributes defined in [NIE-3]. We make our extension so that we do not have to modify structural productions of the abstract syntax in [NIE-3] or definitions of its attributes. We suppose that the reader is familiar with our restructuring specification in [NIE-3].

4.1. The extension of the abstract syntax for data reformatting

Our abstract syntax defined in [NIE-3] is able to generate logical data units of the flat file or hierarchical data model. We state the following requirements for the extension of this abstract syntax:
- It must generate generally those aspects of the source and target sw/hw environments which have an effect on data encoding.
- It must generate those features specified by a user that have an essential role in data storing.

In terms of the definition of attributes we tie all these features together and specify the bit string representation of each source data structure being converted (it is
possible to have many source data structures, see [NIE-3])
and of the target data structure.

The abstract syntax for data conversion consists of the
abstract syntax defined in [NIE-3] (the structural productions
(1)-(14) and the structural productions (A)-(S) below
intended for data reformattting. In other words, the
structural productions (1)-(14) and (A)-(S) form one
abstract syntax. Next we define the structural productions
(A)-(S) which are constructed by the same principle as
the structural productions (1)-(14) in [NIE-3].

(A) pdb:Physical data base<db:Data base,e.,
Environment description ,e.g.:Environment description,
sp=1:User specification,sp=2:User specification>

(B) ei:Environment description>E-DESCRIPTION<e:Character
code, j-fact:Justification,p-char:Padding character>

(C) o:Character code >C-CODE<cp:C-pair>

(D) cp:C-pair>C-PAIR<sy:Symbol,bc:Bit code>

(E) sy:Symbol>SYMBOL >

(F) bc:Bit code>B-CODE >

(G) j-fact:Justification>JUSTIFICATION >

(H) p-char:Padding character>P-CHARACTER >

(I) sp:User specification>U-SPECIFICATION<ret:Record
code table, lt:Length-table, st:Sort-table>

(J) ret:Record code table>R-TABLE<rp:R-pair*>

(K) rp:R-pair>R-PAIR<rn:Record type name,rc:Record code>

(L) rc:Record code>RECORDCODE >

(M) lt:Length-table>L-TABLE<lp:L-pair*>

(N) lp:L-pair>L-PAIR<dn:D-name, l:Length>

(P) l:Length>LENGTH >

(Q) st:Sort-table>S-TABLE<sp:S-pair*>

(R) sp:S-pair>S-PAIR<dn:D-names, si:Sort indicator>

(S) si:Sort indicator>S-INDICATOR >
Intuitively, the *Data base* object in the structural production (A) represents the result of data restructuring or a target data structure (see the structural production 1 in [NIE-3]). In (A) the abstract objects $e_1$ and $e_2$ with their component objects describe some source or target *hw/sw* environments, respectively. The abstract object sp-1 with its component objects describe those aspects of the source data structures specified by the user which must be taken into account in data encoding. The object sp-2 with its component objects describe the corresponding aspects concerning the target data structure.

With the structural productions (B), (C), (D), (E), (F), (G) and (H) we generate a character code, a justification factor and a padding character i.e., those features which remain stable in a certain *hw/sw* environment. The structural productions (I), (J) and (K) generate record type codes for hierarchical data bases and the structural productions (M), (N) and (P) generate lengths for data items. With the structural productions (Q), (R) and (S) we generate the data items used in sorting and a sorting order.

It is important to note that a generated *Record code table* object (with its component objects) must contain the record type codes for all hierarchical data bases, provided we have many hierarchical source data bases. It is also true that generated source data structures are based on the flat file model. In this case $1(rp) = \emptyset$ is true in the context of the structural production (J). Similarly the structural productions (M), (N) and (P) generate the lengths for all data items of the different source data structures. Two or more source data structures may contain same data items. In order to abbreviate our discussion we assume that the same length has been allocated for these data items in different source data structures.

4.2 THE DEFINITION OF THE ATTRIBUTES FOR DATA REFORMATTING

The semantic and checking attributes have the same role as in our previous work [NIE-3]. We also shall utilize some attributes defined in our previous work. Next we consider informally our attributes.

The synthesized semantic attributes are *symbol*, *bitcode*, *d-name*, *rt-name*, *cc-table*, *length-table*, *sort-table*, *rtc-table*, *j-factor*, *p-char* and *bitstring*. The attributes *symbol* and *bitcode* are used to describe symbols and their bit string representations in the character code (the set-valued attribute *cc-table*) available in a generated *hw/sw* environment. The attributes *j-factor* and *p-char* express the justification factor and the padding character used by a *hw/sw* environment.
Intuitively, the attribute \texttt{d-name} contains a data item name specified by a user. The set-valued attributes \texttt{length-table} \texttt{sort-table} and \texttt{rtc-table} correspond to the functions \texttt{length-table}, \texttt{sort-table} and \texttt{rtc-table} used in the data encoding functions above. The attribute \texttt{rt-name} expresses the name of a record type.

The attribute \texttt{bitstring} defines the bit string representation of each source data structure being converted and the target data structure. In other words, with this attribute we define the results of the encoding of the source data structures and the target data structure. It is obvious that the attribute \texttt{bitstring} of the starting nonterminal \texttt{Physical data base} defines that input file from which a physical flat file or hierarchical data base can be loaded by a load utility available in the target \texttt{hw/sw} environment.

The inherited semantic attributes are \texttt{srt-table}, \texttt{len-table}, \texttt{rc-table}, \texttt{code-table}, \texttt{just-factor} and \texttt{pad-char}. Intuitively, these attributes have the same meanings and representations as the corresponding synthesized attributes \texttt{sort-table}, \texttt{length-table}, \texttt{rtc-table}, \texttt{cc-table}, \texttt{j-factor} and \texttt{p-char}.

The checking attributes \texttt{symbolchecking}, \texttt{u-rtcodes}, \texttt{legal-padding}, \texttt{legal-justification}, \texttt{length}, \texttt{legal-indicator}, \texttt{legal-sorting}, \texttt{all-lengths} and \texttt{is-function} are Boolean valued and synthesized. In other words, their values are true in each object generated by our abstract syntax. Next we characterize briefly the purposes of these attributes in our specification.

- The attribute \texttt{symbolchecking} checks that each symbol in the character code of the source system is included in the character code of the target system, too. This guarantees that each data item value of the source system can be also stored in the target system.

- The value true of the attribute \texttt{u-rtcodes} indicates that each record type of a hierarchical data base has a unique record type code.

- With the attribute \texttt{legal-padding} we check that the generated padding character is included in the character code.

- Justification factor can be \texttt{RIGHT} or \texttt{LEFT}. We define the attribute \texttt{legal-justification} to check this fact.

- The attribute \texttt{legal-length} checks that the lengths generated by the abstract syntax are positive integers ($I^+$).
The sorting order can be ascending or descending. Therefore we check with the attribute \texttt{legalindicator} that each generated sorting indicator is either ASC or DESC.

Our background assumption above was that each record type in the source data structures and in the target data structure has at most one sorting data item. This is checked by the attribute \texttt{legalsorting}.

The attribute \texttt{all-lengths} checks that some length has been allocated to each data item in the source data structures and in the target data structure.

The attribute \texttt{is-function} is associated with \texttt{Character code objects,Record code table objects,length-table objects and Sort-table objects}. The attributes \texttt{cc-table}, \texttt{length-table}, \texttt{sort-table} and \texttt{rtc-table} are sets which consist of ordered pairs. With the attribute \texttt{is-function} we check that the attributes in question are functions, too (see 2.2.). In addition we require that the attribute \texttt{cc-table} describes a bijective function, i.e. each symbol in a character code must have a unique bit string representation and vice versa.

The structural productions (2) and (3) of our abstract syntax (see [NIE-3]) generate \texttt{File} objects. Because the attribute \texttt{bitstring} is defined for the nonterminal \texttt{File} it must also be defined for \texttt{File} objects generated by the structural production (2) although we are not interested in the bit string representations of the intermediate files. Intermediate files in our conversion process (see [NIE-3]) are always flat files. However, we cannot apply the \texttt{flatfile-encoding} function defined above in the definition of the attribute \texttt{bitstring}. This is due to the fact that the sort information generated by our abstract syntax is associated with the source data structures and the target data structure—not with the generated intermediate files. Therefore, we define the \texttt{intermediatefile-encoding} function which encodes an intermediate file by choosing indeterministically its records. The arguments of this function are familiar from our previous encoding functions.

\begin{align*}
\texttt{intermediatefile-encoding}(dse, length-table, co-table, p-char, justify) &= \\
\texttt{concatenation(record-encoding}(\Pi([t | t \in dse]), co-table, length-table, p-char, justify, intermediatefile-encoding([f^{-1}_{dse}(<i>):-dse-\Pi([t | t \in dse]), length-table, co-table, p-char, justify]) \quad , \text{if} |dse|>1) \\
&\quad \text{record-encoding}(\Pi([t | t \in dse]), co-table, length-table, p-char, justify) \quad , \text{if} |dse|=1
\end{align*}
Next we give exact definitions for the semantic and checking attributes. We refer to the structural productions presented in [NIE-3] with their numbers and to the structural productions of this paper with their letters.

(A) semantic attributes

\[
\text{bitstring}(pdb) =
\begin{cases}
\text{HDB-encoding}(dse(db),\text{sort-table}(sp-2),\text{rtc-table}(sp-2),\text{length-table}(sp-2),\text{cc-table}(e_2),\text{p-char}(e_2),
\text{j-factor}(e_2))
\quad & \text{if } |e_2(dse(db))| > 1 /*The target data structure is a hierarchical data base */ \\
\text{flatfile-encoding}(dse(db),\text{sort-table}(sp-2),
\text{length-table}(sp-2),\text{cc-table}(e_2),\text{p-char}(e_2),
\text{j-factor}(e_2))
\quad & \text{if } |e_2(dse(db))| = 1 /*The target data structure is a flat file */ 
\end{cases}
\]

\text{srt-table}(db) = \text{sort-table}(sp-2)
\text{rc-table}(db) = \text{rtc-table}(sp-2)
\text{len-table}(db) = \text{length-table}(sp-1)
\text{ccode-table}(db) = \text{cc-table}(e_1)
\text{pad-char}(db) = \text{p-char}(e_1)
\text{just-factor}(db) = \text{j-factor}(e_1)

checking attributes

\text{symbolchecking}(pdb) = \text{dom}(\text{cc-table}(e_1)) \cap \text{dom}(\text{cc-table}(e_2)) /*\text{dom is defined in 2.2} */

\text{all-lengths}(pdb) = \text{dom}(\text{length-table}(sp-2)) = \sigma_4(dse(db)) /*a length has been specified for each data item in a target data structure */
u-rtcodes(\(pdb\)) =
\[
\begin{cases}
\text{dom}(\text{rtc-table}(sp-\delta)) = \sigma_3(dse(db)) \land \\
\forall x \in \sigma_3(dse(db)): \exists y(y \neq x) \in \sigma_3(dse(db)):
\text{rtc-table}(x) = \text{rtc-table}(y)
\end{cases}
\]
where \(rtc-table(sp-\delta)\) is denoted by \(rtc-table\)

, if \(|\sigma_3(dse(db))| > 1\)
true, if \(|\sigma_3(dse(db))| = 1\) /* a flat file*/

legalsorting(\(pdb\)) =
\[
\begin{cases}
\text{dom}(\text{sort-table}(sp-\delta)) \subseteq \sigma_4(dse(db)) \land \\
\forall x \in \sigma_3(dse(db)):
\quad \quad \bigcup f^{-1}(i) \cap \text{dom}(\text{sort-table}(sp-\delta)) \leq 1
\end{cases}
\]
where \(f\) is the naming function of \(dse(db)\)

, if \(\text{sort-table}(sp-\delta) \neq \emptyset\) /* Above we
also check that the sorting data items belong

to the target data structure */
true, if \(\text{sort-table}(sp-\delta) = \emptyset\) /* The

target data structure is not sorted*/

(1) Semantic attributes
\[
\begin{align*}
\text{srt-table}(f) &= \text{srt-table}(db) \\
\text{rc-table}(f) &= \text{rc-table}(db) \\
\text{len-table}(f) &= \text{len-table}(db) \\
\text{ccode-table}(f) &= \text{ccode-table}(db) \\
\text{pad-char}(f) &= \text{pad-char}(db) \\
\text{just-factor}(f) &= \text{just-factor}(db)
\end{align*}
\]

(2) Semantic attributes
\[
\begin{align*}
\text{srt-table}(f_1) &= \text{srt-table}(f) \\
\text{srt-table}(f_2) &= \text{srt-table}(f) \\
\text{rc-table}(f_1) &= \text{rc-table}(f) \\
\text{rc-table}(f_2) &= \text{rc-table}(f) \\
\text{len-table}(f_1) &= \text{len-table}(f) \\
\text{len-table}(f_2) &= \text{len-table}(f) \\
\text{ccode-table}(f_1) &= \text{ccode-table}(f) \\
\text{ccode-table}(f_2) &= \text{ccode-table}(f) \\
\text{pad-char}(f_1) &= \text{pad-char}(f) \\
\text{pad-char}(f_2) &= \text{pad-char}(f) \\
\text{just-factor}(f_1) &= \text{just-factor}(f) \\
\text{just-factor}(f_2) &= \text{just-factor}(f) \\
\text{bitstring}(f) &= \text{intermediate-file-encoding(n-form}(f), \\
\text{len-table}(f), \text{ccode-table}(f), \text{p-char}(f), \text{just-factor}(f))
\end{align*}
\]
checking attributes

\[
\text{all-lengths}(f) = \text{true}
\]

\[
\text{legalsorting}(f) = \text{true}
\]

\[
\text{u-rtcodes}(f) = \text{true}
\]

(3) semantic attributes

\[
\text{bitstring}(f) = \begin{cases}
\text{HDB-encoding}(\text{dse}(f), \text{srt-table}(f), \text{rc-table}(f), \\
\text{len-table}(f), \text{ccode-table}(f), \text{pad-char}(f), \\
\text{just-factor}(f))
\end{cases}
\]

\[
\quad, \text{if } |\sigma_3(\text{dse}(f))| > 1 /* \text{the encoding of a hierarchical source data structure} */
\]

\[
\text{flatfile-encoding}(\text{dse}(f), \text{srt-table}(f), \\
\text{len-table}(f), \text{ccode-table}(f), \text{pad-char}(f), \\
\text{just-factor}(f))
\]

\[
\quad, \text{if } |\sigma_3(\text{dse}(f))| = 1 /* \text{the encoding of a flat source file} */
\]

checking attributes

\[
\text{legalsorting}(f) = \forall x \in \sigma_3(\text{dse}(f)) : \bigcup \{ \epsilon_{\text{dse}}^{-1}(i) \cap \text{dom(\text{srt-table}(f))} | \exists 1 i \in \text{r-content}(f(x)) \}
\]

\[
\quad \text{where } f \text{ is the naming function of } \text{dse}(f)
\]

\[
\quad, \text{if } \text{srt-table}(f) \neq \emptyset
\]

\[
\quad \text{true, if } \text{srt-table}(f) = \emptyset
\]

\[
\text{u-rtcodes}(f) = \sigma_3(\text{dse}(f)) \subseteq \text{dom(\text{rc-table}(f))} \land \forall x \in \sigma_3(\text{dse}(f)) : \\
\exists y (y = x) \in \sigma_3(\text{dse}(f)) : \text{rc-table}(x) = \text{rc-table}(y)
\]

\[
\quad \text{where } \text{rc-table}(f) \text{ is denoted by } \text{rtc-table}
\]

\[
\quad, \text{if } |\sigma_3(\text{dse}(f))| > 1
\]

\[
\quad \text{true, if } |\sigma_3(\text{dse}(f))| = 1 /* \text{a flat file} */
\]

\[
\text{all-lengths}(f) = \sigma_4(\text{dse}(f)) \subseteq \text{dom(\text{len-table}(f))}
\]

(B) semantic attributes

\[
\text{cc-table}(\varepsilon) = \text{cc-table}(\varepsilon)
\]

\[
\text{j-factor}(\varepsilon) = \text{j-fact}
\]

\[
\text{p-char}(\varepsilon) = \text{p-char}
\]
checking attributes

legalpadding(\(e\)) = p-char@dom(cc-table(\(e\)))
legaljustification(\(e\)) = j-fact\{LEFT,RIGHT\}

(C) semantic attributes

cc-table(\(e\)) = \(\bigcup_{i \in \{1,2\}} cc-table(\(op[i]\))\)

checking attributes

is-function(\(e\)) = \(\exists i, j(\(\#i \in op\)), symbol(\(op[i]\)) = symbol(\(op[j]\)) \lor \text{bitcode}(\(op[i]\)) = \text{bitcode}(\(op[j]\))\)

(D) semantic attributes

cc-table(\(op\)) = \{\(<\#y, \#x>\)\}
symbol(\(op\)) = \#y
bitcode(\(op\)) = \#x

(I) semantic attributes

rtc-table(\(sp\)) = rtc-table(\(ret\))
length-table(\(sp\)) = length-table(\(lt\))
sort-table(\(sp\)) = sort-table(\(st\))

(J) semantic attributes

rtc-table(\(ret\)) = \(\bigcup_{i \in \{1,2\}} rtc-table(\(rp[i]\))\)

checking attributes

is-function(\(ret\)) = \(\exists i, j(\(\#i \in rp\)), rt-name(\(rp[i]\)) = rt-name(\(rp[j]\))\)

(K) semantic attributes

rtc-table(\(rp\)) = \{\(<\#n, \#c>\)\}
rt-name(\(rp\)) = \#n

(M) semantic attributes

length-table(\(rp\)) = \(\bigcup_{i \in \{1,2\}} length-table(\(lp[i]\))\)

checking attributes

is-function(\(lt\)) = \(\exists i, j(\(\#i \in lp\)), d-name(\(lp[i]\)) = d-name(\(lp[j]\))\)
(N) **semantic attributes**

length-table\((lp)\) = \(<dn, l>\)

d-name\((lp)\) = \(dn\)

**checking attributes**

legal-length\((lp)\) = \(l \in \mathbb{I}^+\)

(Q) **semantic attributes**

\[
\text{sort-table}(st) = \bigcup_{i \in \{sp\}} \text{sort-table}(sp[i])
\]

**checking attributes**

is-function\((st)\) = \(#i, j \in \{sp\} : d-name(sp[i]) = d-name(sp[j])\)

(R) **semantic attributes**

\[
\text{sort-table}(sp) = \{<dn, si>\}
\]

d-name\((sp)\) = \(dn\)

**checking attributes**

legal-indicator\((si)\) = \(si \in \{ASC, DESC\}\)
5. CONCLUSIONS

We have considered in this paper the conversion of flat files and hierarchical data bases where both data restructuring and data reformatting are taken into account. Our focus is the specification of the data reformatting aspects. We extended our abstract syntax [NIE-3] intended for data restructuring so that it generates also different sw/hw environments where flat files and hierarchical data bases are stored. Likewise, it generates information specified by a user such as lengths for data items, sorting data items, sorting orders etc. which are necessary in the encoding of data structures. Our specification is based on applying the attribute technique in the context of abstract syntax.

Our purpose is to provide such a specification of data conversion which is able to deal with many complex data conversion cases occurring in practice. For example, in terms of our specification we can construct a hierarchical data base from conventional (flat) files when the exchange of a computer occurs at the same time, i.e. also the hw/sw-environment changes totally. Our specification defines in detail, how the bit string representations of source data structures are transformed into the bit string representation of a target data structure. Our starting point to data reformatting was that we can utilize load and unload utilities available in the source and target systems.
CONCLUSIONS OF THE THESIS

We have studied the conversion problem of flat files and hierarchical data bases. Complex data conversion problems can contain both data restructuring and data reformatting. Therefore we considered the aspects associated with both data restructuring and data reformatting. In this thesis we emphasized data restructuring aspects because it has been widely recognized that it is the most complex part in the general data conversion process.

The exhaustive treatment of data restructuring requires that we deal with it at three abstraction levels (schema level, instance level, value level). So far, there has been a lack of such a formalism which is able to deal with the data restructuring problem at all these essential abstraction levels. In this thesis we developed such a formalism. Our formalism is based on the concept of the data structure element (dse) which allows flexible and precise interfaces between these abstraction levels. We can utilize the dse description in other data base areas, too. For example, in [30] we have shown that the dse concept affords the possibility of defining and processing the relational model precisely, generally and flexibly. Furthermore, in [29] we have specified a query language with relationally complete expressive power on the basis of the dse descriptions of relations.

In this thesis we developed a general restructuring function for hierarchical data bases which has many desirable new properties. The developed restructuring function is based on the dse description of hierarchical data bases. In the structural sense, the minimal information which is needed for data restructuring of a hierarchical data base, is the source data base and the target schema. We had this as a starting point to data restructuring.

In practice there are many restructuring cases which require the construction of the target data structure from many source data structures. Therefore we developed a formal software specification which is able to construct the target data structure from many source data structures. In this specification the developed restructuring function and its parts have a central role. The construction of the target data structure requires that certain relationships exist among the source data. Our specification contains exact conditions that check that the source data satisfy these requirements. Our specification permits that data structures being restructured can be based on the flat file data model or on the hierarchical data model. This feature has many important practical consequences.


[35] The Stored-Data Definition and Translation Task Group, 
Stored-data description and data translation: A model 
and language, Information Systems, Vol.3B, 95-148, 
1977.

data traversals and operations in application program 
to account for semantic changes of databases, ACM 

[37] R. Taylor, Generalized data base management system, 
data structures and their mapping to physical storage, 

[38] D.C. Tsichritzis and F.H. Lochovskly, Data base 

[39] P.A.S. Veloso, J.M.V. De Castilho and A.L. Furtado, 
Systematic derivation of complementary specifications, 
in proc. 7th International Conference on Very Large Data 
Bases, Mexico City, 409-421, 1981.
ERRATA

PART I: page 154: In the definition of \( f_{x_1 \times x_2} (a) \);
\[ f_{x_2} (a) \ast \max (I_{x_1}) \] should be \( f_{x_2} (a) + \max (I_{x_1}) \).

PART II: page 396: In the definition of the function partpath;
\[ \bigcup_{i \in r \text{-content} (i_{ind_2})} f^{-1}(i) \] should be \( \bigcup_{i \in r \text{-content} (i_{ind_2})} \times(i) \).

- page 399: \( \tilde{c} = \{ (a_1, \ldots) \} \) should be \( \tilde{c} = \{ (a_1, \ldots) \} \).

- page 401: \( \langle a_1, b_1, b_2, c_2, f_2, h_1, l_1, k_1, p_1 \rangle \)
should be \( \langle a_1, b_1, d_2, c_2, f_2, h_1, l_1, k_1, p_1 \rangle \).

- page 404: \( f(\emptyset) = \langle 1 \rangle \) should be \( f(\emptyset) = \langle 1 \rangle \).

- page 406: In the definition of the function h-join;
\( f^{-1}(\langle 1 \rangle) \{ (\ldots) \} \) should be \( f^{-1}(\langle 1 \rangle) \{ (\ldots) \} \).

PART III: page 441: \( a_2 (n \text{-form}(f)) f_n (a) \) should be \( a_2 (n \text{-form}(f)) f_n (a) \).

- page 442: \( r \text{-name}(r[7]) \) should be \( r \text{-name}(r[1]) \).

- page 452: In the last paragraph of chapter 5 "data restructures" should be "data restructurers".

- page 454: Definition 8: \( t_1 (x_1, X_1, RN_1, D_1, I_{x_1}, f_{x_1}) \)
should be \( t_1 = (x_1, X_1, RN_1, D_1, I_{x_1}, f_{x_1}) \).

- page 456: Definition 17: \( \exists x \) should be \( \forall x \).
Definition 21: \( t_1 \cap t_2 \) should be \( t_1 \cap t_2 \).
Definition 22: \( k(\ldots) \lor k(\ldots) \) should be \( k(\ldots) \land k(\ldots) \).