Journal of Universal Computer Science, vol. 5, no. 4 (1999), 243-286
submitted: 12/4/99, accepted: 23/4/99, appeared: 28/4/99 © Springer Pub. Co.

Automatic Data Restructuring

Seymour Ginsburg
(Computer Science Department, University of Southern California,
Los Angeles, California, 90089
e-mail: ginsburg@pollux.usc.edu)

Nan C. Shu
(IBM Los Angeles Scientific Center,
Los Angeles,California)

Dan A. Simovici
(University of Massachusetts at Boston,
Department of Mathematics and Computer Science,
Boston, Massachusetts 02125
e-mail: dsim@cs.umb.edu)

Abstract: Data restructuring is often an integral but non-trivial part of information processing, especially when the data structures are fairly complicated. This paper describes the underpinnings of a program, called the Restructurer, that relieves the user of the "thinking and coding" process normally associated with writing procedural programs for data restructuring. The process is accomplished by the Restructurer in two stages. In the first, the differences in the input and output data structures are recognized and the applicability of various transformation rules analyzed. The result is a plan for mapping the specified input to the desired output. In the second stage, the plan is executed using embedded knowledge about both the target language and run-time efficiency considerations.
The emphasis of this paper is on the planning stage. The restructuring operations and the mapping strategies are informally described and explained with mathematical formalism. The notion of solution of a set of instantiated forms with respect to an output form is then introduced. Finally, it is shown that such a solution exists if and only if the Restructurer produces one.

Key Words: non-first normal form databases, data restructuring, instantiated forms, hierarchical structures, solution of a set of instantiated forms.
Category: H.2, E.1

1 Introduction

The rapid decline of computing costs and the desire for productivity improvement has created increased demands for computerized information processing by end users [22]. The challenge for computer professionals is to bring computing capabilities - usefully and simply - to people without special computer training. The emergence of visual programming [36, 8, 24, 38, 6] and the resurgence of automatic programming [5] represent two significant approaches to meet this challenge. FORMAL [34], a visual-directed and forms-oriented application development system, is a prototype that embodies the spirits of both. The visual programming aspects of FORMAL have been described elsewhere [34, 36, 37]. The purpose of this paper is to present the *automatic* aspects of data restruc-

turing in FORMAL and a theoretical foundation for it.[1] However, the abstract considerations presented here are not dependent on the particular implementation solutions adopted in FORMAL.

Data restructuring is often done in an ad hoc manner. What sets our method apart from other data restructuring efforts is the automatic mechanism's ability to take over the "thinking and coding" process normally associated with writing algorithms for data restructuring. For convenience this mechanism is called "the Automatic Restructurer" (or simply the Restructurer). The most important pieces of information given to the Automatic Restructurer are an unambiguous description of each input and an unambiguous structural description of the desired output. If there are two or more inputs, then the match fields used to tie the inputs together must also be given. Based on these descriptions, an executable program (using the restructuring operations originally proposed in CONVERT [31]) is generated by a two-stage process. In the first stage, the differences between the input and output data structures are recognized and the applicability of various transformational rules analyzed. The result is a plan for mapping the specified input(s) to the desired output. In the second stage, runtime efficiencies are taken into consideration, and an executable program is implemented to carry out the plan. The result is a reasonably efficient program for the restructuring task at hand. A major contribution of this paper is a detailed explanation (expressed both formally and informally) of the underpinnings of the automatic restructuring mechanism. Since the efficiency considerations and implementation details are not the primary interests of this paper, the concerns of the second (i.e., construction or coding) stage are included only to complete the exposition. The emphasis is on the mechanism underlying the first (i.e., planning) stage, namely, on the derivation of a sequence of restructuring operations aimed at producing a user-specified output from the given input.

The paper itself is organized into eight sections, the first being this introduction. Section 2 explains the notion of "data restructuring." Section 3 presents the underlying data model. Section 4 discusses the restructuring problem. Section 5 defines four types of basic operations designed specifically for data restructuring. Section 6 considers the strategies for mapping specified inputs to desired outputs, using sequences of these basic operations. Section 7 discusses the efficiency considerations. Section 8 presents a summary of the results obtained.

2 Data Restructuring

As mentioned in the Introduction, the purpose of this paper is to present a method for automatically performing data restructuring tasks and provide a theoretical basis for it. This section explains and elaborates on the meaning of "data restructuring," why it is an important part of information processing, and what is the significance of an automatic programming approach.

Historically, database reorganization is defined as "changing some aspect of the way in which a database is arranged logically and/or physically" [39] - a

[1] The facilities provided by FORMAL are a superset of those described in this paper. Here, we choose to discuss only those directly relevant to the *restructuring* of the data structures. Capabilities such as changing component names, specifying criteria for selections, case-by-case assignments, arithmetic and string operations, handling of exceptions, etc., are not germane to structural transformations, and are therefore not included in the discussion.

generic term that covers both data restructuring (changing logical structures) and reformating (changing physical structures) of database systems. With that perspective, data restructuring fell primarily in the domain of data-processing operations personnel and database researchers. Our view of data restructuring has a much broader scope. Here, the term *data restructuring* is to mean a sequence of operations aimed at producing an output with structure different from that of the corresponding input. Data restructuring may be performed for a variety of reasons, for example, as a desirable database function (to improve performance or storage utilization), or as a useful application (to support decision making or data processing).

(PERSON)											
ENO	DNO	NAME	PHONE	JC	(KIDS)		(SCHOOL)			SEX	LOC
					KNAME	AGE	SNAME	(ENROLL)			
								YEARIN	YEAROUT		
05	D1	SMITH	5555	05	JOHN	02	PRINCETON	1966	1970	F	SF
					MARY	04		1972	1976		
07	D1	JONES	5555	05	DICK	07	SJS	1960	1965	F	SF
					JANE	04					
							BERKEPRY	1965	1969		
11	D1	ENGEL	2568	05	RITA	04	UCLA	1970	1974	F	LA
12	D1	DURAN	7610	05	MARY	08				M	SF
					BOB	10					
19	D1	HOPE	3150	07	MARYLOU	10				M	SJ
					MARYANN	07					
02	D2	GREEN	1111	01	DAVID	04				M	SF
20	D2	CHU	3348	10	CHARLIE	06	HONGKONG	1962	1966	F	LA
					CHRIS	09					
					BONNIE	04	STANFORD	1967	1969		
								1972	1975		
21	D2	DWAN	3535	12			USC	1970	1974	F	SJ
43	D2	JACOB	4643	09	PAULA	07	BERKEPRY	1962	1966	M	SJ

Figure 1: (PERSON) file

Data restructuring plays an important role in information processing applications. When data is extracted from sources or new fields are created and placed in the output, the structure of the resulting output seldom resembles that of the input. To illustrate, consider the personnel information of a given company shown in the (PERSON) file described in Figure 1. Suppose a Christmas party is being planned. All employees' children will receive gifts at the party. Different gifts are planned for each age group. The organizer of the party wishes to list, for each location (LOC), by age of the children (AGE), the name of each kid (KNAME) and his/her parent name (NAME). For this seemingly simple example, all information required to produce the desired (XMASLIST) file (Figure 2) is contained in the (PERSON) file. Yet, the actual process of producing the desired (XMASLIST) involves data restructuring of a non-trivial nature (see Figure 3).

Besides showing that data restructuring is an integral part of many information processing applications, the above example also shows that query facilities (designed for simple retrieval) are not adequate to handle many of the real world situations: *programming is required.* Traditionally, programming is considered to

(XMASLIST)			
LOC	(RECIPIENT)		
	AGE	(KID)	
		KNAME	NAME
LA	04	BONNIE	CHU
		RITA	ENGEL
	06	CHARLIE	CHU
	09	CHRIS	CHU
SF	02	JOHN	SMITH
	04	MARY	SMITH
		JANE	JONES
		DAVID	GREEN
	07	DICK	JONES
	08	MARY	DURAN
	10	BOB	DURAN
SJ	07	PAULA	JACOB
		MARYANN	HOPE
	10	MARYLOU	HOPE

Figure 2: Desired (XMASLIST)

be in the professional programmers' province. But the reliance on a professional programmer often means placing a formal request to the data processing personnel and waiting on a priority list for programmers to take action. The alternative is for the end users to learn to program.

Learning to program is generally a time-consuming and often frustrating endeavor. Moreover, even after the skill is learned, writing and testing a program is still tedious and labor intensive. Writing programs to perform data transformation is no exception. To ease the task, special languages with high level operations for data restructuring have been reported in the literature. (See [1, 11, 17, 20, 21, 25, 30, 31, 32, 40, 42] for example.) These languages are high level, but they are still "procedural" and programming training is necessary. In other words, users of these languages must write out the algorithms of restructuring in a step by step manner. (See Appendix A for an example.)

In the rapidly growing end-user computing environment, many non-DP professionals simply do not have the time, interest, or skill to program. Indeed, as noted in a survey of end-user computing ([27]), two thirds of the end-user applications were developed by support personnel, programmers, and consultants. It is our contention that unless an essential part of tedious, low level programming can be made automatic, a large community of potential end users will not develop their applications. In the remainder of this paper, we present an automatic restructuring mechanism which is aimed at decreasing the tedium normally associated with writing procedural programs. For example, because of the Automatic Restructurer implemented in FORMAL, (XMASLIST) can be created from (PERSON) with the specification shown in Figure 4. In essence, the user describes the structure of the desired output (XMASLIST) and tells the system to find the required information in the source form (PERSON). The system does the rest.

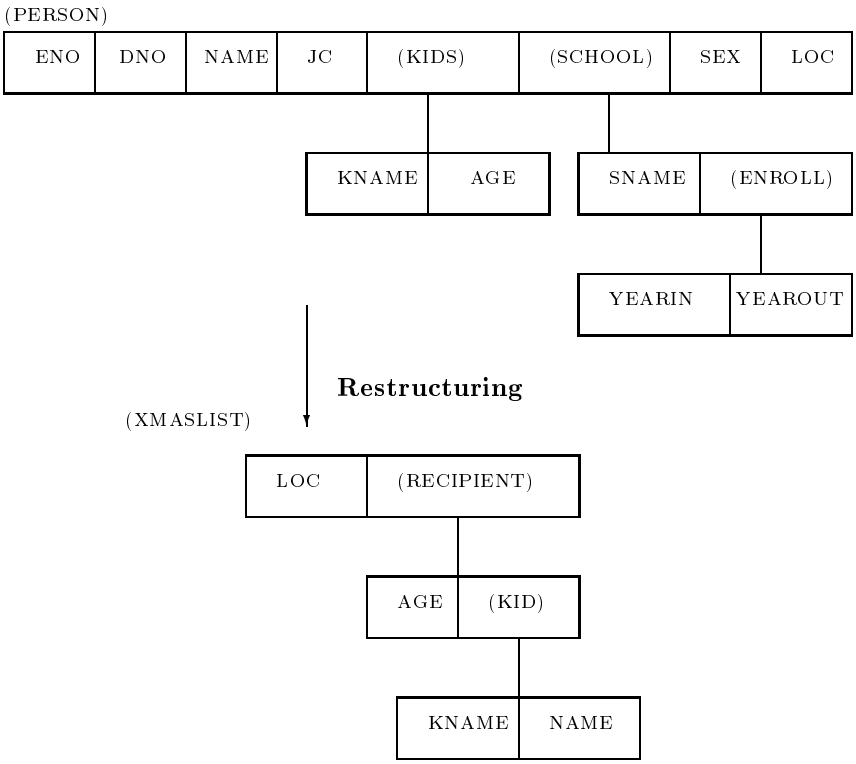


Figure 3: Restructuring of (PERSON) to (XMASLIST)

3 The Underlying Data Model

As mentioned in the Introduction, the most important information given to the Restructurer is an unambiguous description of the input and output data structures. In this section, we discuss the data structures supported by the Restructurer, and indicate several ways to describe them.

Briefly, the Automatic Restructurer handles all data structures as *hierarchies*. *Relational tables* are treated as degenerate hierarchies (i.e., hierarchies of one level), and *networks* are assumed to have been decomposed into families of hierarchies before the Automatic Restructurer is invoked. No conceptual limitations are placed on the depth or width of the hierarchies.

There are many different ways to describe hierarchical data structures. In the database community, hierarchy graphs are commonly used when data structures are discussed. As an example, Figure 3 shows the hierarchy graphs of (PERSON) and (XMASLIST). But hierarchy graphs become cumbersome when the display

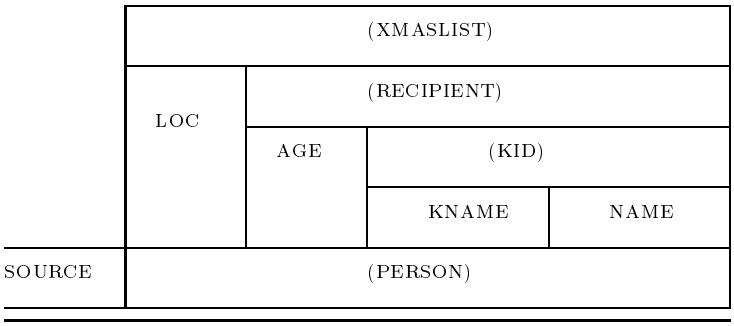


Figure 4: A specification for creating (XMASLIST) from (PERSON)

of instances is desired. The difficulty is that one "instance graph" is required to represent each occurrence, and consequently many "instance graphs" are needed to show multiple occurrences.

Data Processing personnel may or may not care about instance graphs, but for users who do not have very much data-processing training, visualization of occurrences often enhances the understanding of a problem. A "form" data model was therefore suggested in [31] as a two-dimensional representation of hierarchical data where multiple occurrences can be displayed conveniently. It was intended at that time as a visual aid to assist the users in understanding the high level operations performed on their data. The "form operation by example" [23], the nested table (NT) model [20], and more recently, the "nested normal forms", and "non first normal form relational databases" (see [1, 9, 11, 12, 18, 19, 28, 29, 30, 41], for example) use similar representations as visual aids. In these representations, hierarchical structures are implied by exhibiting sample instances.

Visual aids enhance human understanding. But, unless the representations are precise, they cannot be processed by a computer program (data restructurer included). In order to provide unambiguous descriptions (without depending on the display of sample instances) to the Restructurer, the hierarchical structures must be made *explicit.* A straightforward approach is to refine the "form headings" used in the form model so that the one-many relationships are distinguished from the one-one relationships [33, 10]. Before presenting the refined form heading as an unambiguous description of a hierarchical structure, a few definitions are in order.

A *field* is the smallest unit of data that can be referenced in a user's application. A *group* is a sequence of one or more fields and a number of subordinate groups. For example, ENO (Employee number), DNO (Department number), NAME (Employee Name), PHONE, JC (Job Code), . . . , are fields of the (PERSON) file (Figure 1), while (KIDS), (SCHOOL), and (ENROLL) are groups.

Groups are either repeating or non-repeating. A *non-repeating* group is a convenient way to refer to a *sequence* of fields *as a unit* (e.g., DATE is a non-

repeating group over MONTH, DAY and YEAR). A *repeating group,* on the other hand, represents the one-many relationship and thus exhibits the hierarchical relationship. Take (PERSON) (Figure 1) for example. An employee may have many children, and may have attended a number of schools. Thus, for an employee, (KIDS) and (SCHOOL) are repeating groups. Repeating groups can be nested (representing several levels of a hierarchy, e.g., (ENROLL) is a subordinate repeating group, nested within (SCHOOL)), or in parallel (representing several branches of a hierarchy, e.g., (KIDS) and (SCHOOL) represent two different branches). To simplify the discussion, we do not deal with the non-repeating groups in the rest of this paper.

In data processing applications, a *file* is a named collection of homogeneously structured data. The structure of a file is generally defined in terms of its components (where a *component* is either a field or a group). Component names are assumed to be *unique* for each file, i.e., there are no duplicate names among the components.

In FORMAL, a *form heading* assumes a role that is commonly known as the data structure definition in conventional programming languages. Its purpose is to unambiguously define the name of the associated file, its components, and structural relationships among the components. Parentheses are used to denote the existence of multiple (i.e. zero or more) occurrences in the associated file or group. *Thus, names of the file and its repeating groups (also called subforms) are enclosed in parentheses.* More precisely, in a form heading, the file name is placed on the top line. The names of the root or top level components are shown in columns under the file name. The names of components contained in a group, in turn, are placed in columns under the name of the containing group. A double line signals the end of a form heading. (See the lines above the double line in Figure 1).

In a form heading, the nesting and/or parallel parenthesized names serve to define explicitly a hierarchical structure. The *level* of each component is indicated by its row-position under the file name. To illustrate, the level numbers are explicitly shown in Figure 5 on the outside of the (PERSON) form heading. For example, ENO, (KIDS), and (SCHOOL) are at level 1, SNAME is at level 2 and YEARIN level 3.

The Restructurer classifies data structures (or "forms") into three distinct *shapes* : "flat", "branch", and "tree." The algorithm for determining the shape is quite simple. It is implemented by traversing a given two-dimensional form heading (or its equivalent "component description table", discussed later). During the traversal, each component in the form heading is assigned a level number according to its row position. At the end of the traversal, the shape of the data structure is determined. If the number of parenthesized names equals one, then none of the components is a repeating group (or subform). The structure is *flat* shaped. If the number of parenthesized names (say p) is greater than one, then that number (p) is compared with the largest level number (say n). Obviously, if p is larger than n, then at least one level must have more than one subform, and the shape of the form is a *tree.* (See Figure 5 for an example.) Otherwise, the structure is a *branch.* (See (XMASLIST) in Figure 4 for an example.)

The shapes of flat, branch, and tree structures are ranked 1, 2 and 3, respectively. The relative shape rankings of the input and output structures have a bearing on how the Restructurer maps the strategy for transformation (discussed in Section 6). In the rest of the paper, when the shape of the data structure

(PERSON)												level
ENO	DNO	NAME	PHONE	JC	(KIDS)		(SCHOOL)			SEX	LOC	1
					KNAME	AGE	SNAME	(ENROLL)				2
								YEARIN	YEAROUT			3

Index	IDENTIFIER	NEXTI	CHILDI	PARENTI	PRVEL	GROUP
0	PERSON	0	1	0	0	1
1	ENO	2	0	0	1	0
2	DNO	3	0	0	1	0
3	NAME	4	0	0	1	0
4	PHONE	5	0	0	1	0
5	JC	6	0	0	1	0
6	KIDS	7	10	0	1	1
7	SCHOOL	8	12	0	1	1
8	SEX	9	0	0	1	0
9	LOC	0	0	0	1	0
10	KNAME	11	0	6	2	0
11	AGE	0	0	6	2	0
12	SNAME	13	0	7	2	0
13	ENROLL	0	14	7	2	1
14	YEARIN	15	0	13	3	0
15	YEAROUT	0	0	13	3	0

Note: NEXTI, CHILDI and PARENTI are *indexes* to entries in the component description table for next component, child component,and parent component, respectively.

Figure 5: Component description table of (PERSON) form

is not the center of attention, the generic term "tree" is used interchangeably with the term "hierarchy" (discussed in the introduction to the present section).

To view the data, occurrences are displayed under the form heading. The compactness of the form heading enables the visualization of many occurrences at one time (see Figure 1).

Since the form heading represents a concrete means to unambiguously define a hierarchical structure of arbitrary depth and width, it is used as an input/output description by FORMAL. An input is defined once (regardless of the number of different applications utilizing it) by a user or a systems person, and the description is immediately available in the system catalog. There is no need for another user to define the same file again. Description of the desired output, on the other hand, is provided by the user as part of his application specification. For example, in Figure 4, the description of (XMASLIST) is provided in the form heading of the FORMAL program that produces (XMASLIST). As discussed later in Section 6, intermediate files may be needed during the restructuring process. Structures of all intermediate files are defined by the Restructurer and are of no concern to the end users.

As mentioned earlier, it is necessary that the unambiguous descriptions of input and output data structures are available to the automatic restructuring mechanism, but it is immaterial how the required information is captured. In the FORMAL implementation, a parser translates the two-dimensional form headings into *component description tables* which are used as input to the Automatic Restructurer. As an example, Figure 5 shows the component description table corresponding to the (PERSON) form heading. The component description tables are easily translatable from the form headings, and vice versa. They are really two representations of the same information, one being more convenient for the user, the other more suitable for machine manipulation.

In the formal treatment which follows, we use the term *form* to denote, in an abstracted form, the unambiguous description of a data structure. In the examples, we use form heading as the concrete representation of the abstracted notion.

In order to discuss the data restructuring process in a more abstract manner, a formal description of the hierarchical data structures is now presented.

Let $\mathbf{BA}_\infty$ be an infinite set of abstract elements called *basic attributes*. (Basic attributes correspond to the fields mentioned earlier.) For each basic attribute A, let

- $\mathrm{Dom}(A)$ (called the *domain* of A) be a nonempty set of at least two abstract elements, exactly one of which is denoted $\perp_A$ (the null symbol);
- the *basic attributes* of A, denoted $\mathbf{BA}(A)$, be the set $\{A\}$; and
- the *attributes* of A, denoted $\mathbf{Attrib}(A)$, be the set $\{A\}$.

Continuing recursively, let $B_1, \ldots, B_r$ be $r \geq 1$ attributes such that $\mathbf{BA}(B_i) \cap \mathbf{BA}(B_j) = \emptyset$ for all $i \neq j$, and *at least one of them* a basic attribute. Let $B = \langle B_1, \ldots, B_r \rangle$, where "$\langle$" and "$\rangle$" are two special symbols. Then

- $\mathrm{Dom}(B)$ is the family of *finite subsets* of the (ordered) Cartesian product $\mathrm{Dom}(B_1) \times \ldots \times \mathrm{Dom}(B_r)$;
- B is called a *form (attribute)* and a *parent* of each B_i, and each B_i is an *offspring* or *an immediate descendent* of B;
- $\mathbf{BA}(B) = \bigcup_i \mathbf{BA}(B_i)$, and is called the set of *basic attributes* of B; and
- $\mathbf{Attrib}(B) = \{B\} \cup \bigcup_i \mathbf{Attrib}(B_i)$, and is called the set of *attributes* of B.

Each attribute in $\mathbf{Attrib}(B) - \{B\}$ is called a *proper* attribute of B. Each B_i which is a form is called a *subform* of B. Recursively, each subform of a subform B_i is also called a *subform* of B. If each B_i is a basic attribute, then B is said to be a *flat form*. If B has at least one subform and B and each subform of B is of the shape $B' = \langle B'_1, \ldots, B'_s \rangle$, where at most one B'_i is a form, then B is said to be a *branch*.

To illustrate, in Figure 1, ENO, DNO, NAME, PHONE, JC, KNAME, AGE, SNAME, YEARIN, YEAROUT, SEX and LOC are basic attributes.

$$\begin{aligned}
(\mathrm{ENROLL}) &= \langle \mathrm{YEARIN}, \mathrm{YEAROUT} \rangle, \\
(\mathrm{SCHOOL}) &= \langle \mathrm{SNAME}, \mathrm{ENROLL} \rangle, \\
(\mathrm{KIDS}) &= \langle \mathrm{KNAME}, \mathrm{AGE} \rangle, \text{ and} \\
(\mathrm{PERSON}) &= \langle \mathrm{ENO}, \ldots, \mathrm{JC}, (\mathrm{KIDS}), (\mathrm{SCHOOL}), \mathrm{SEX}, \mathrm{LOC} \rangle
\end{aligned}$$

are forms.

(PERSON) is the parent of ENO, DNO, ..., and LOC. Each attribute except (PERSON) is a proper attribute of (PERSON).

Note that each proper attribute C of B has exactly one parent, denoted $\mathbf{Par}_B(C)$, or $\mathbf{Par}(C)$ when B is understood. Also, if D is a proper attribute of C and C is an attribute of B, then $\mathbf{Par}_C(D) = \mathbf{Par}_B(D)$.

C is an *ancestor* of C' if either $C = \mathbf{Par}(C')$, or $C = \mathbf{Par}(D)$ and D is an ancestor of C' (alternatively, $C = \mathbf{Par}(C')$, or C is an ancestor of D and $D = \mathbf{Par}(C')$). C' is said to be a *descendent* of C if C is an ancestor of C'. C and C' are said to be *siblings* if $\mathbf{Par}(C) = \mathbf{Par}(C')$.

If C is an ancestor of B, then C is a form.

Let C be an attribute in $\mathbf{Attrib}(F)$. If $D_1, \ldots, D_n$ are the immediate descendents of C in F, then $\mathbf{Free}_F(C) = \{D_i | 1 \leq i \leq n$ and D_i in $\mathbf{BA}(F)\}$. If C in $\mathbf{BA}(F)$, then $\mathbf{Free}_F(C) = \{C\}$. The subscript F is omitted when F is clear from the context. Note that $\mathbf{Free}_F(C) \neq \emptyset$ for each C in $\mathbf{Attrib}(F)$. Thus

$$\begin{aligned}
&\mathbf{Free}((\text{PERSON})) = \{\text{ENO}, \text{DNO}, \text{NAME}, \text{PHONE}, \text{JC}, \text{SEX}, \text{LOC}\},\\
&\mathbf{Free}((\text{SCHOOL})) = \{\text{SNAME}\} \text{ and}\\
&\mathbf{Free}((\text{ENROLL})) = \{\text{YEARIN}, \text{YEAROUT}\}.
\end{aligned}$$

Let B, B' be two attributes. B equals B' (denoted by $B = B'$) if

- B and B' are the same basic attribute, or
- $B = \langle B_1, \ldots, B_r \rangle$, $B' = \langle B'_1, \ldots, B'_r \rangle$ and $B_i = B'_i$ for $1 \leq i \leq r$.

This means that two forms that have the same direct descendents are considered identical.

Referring to the level number discussed earlier, let $F = \langle F_1, \ldots, F_r \rangle$ be a form. The *level* of each attribute F_i with respect to F is defined to be 1. Continuing by induction, suppose F' is an attribute in F of level l with respect to F. For $F' = \langle F'_1, \ldots, F'_s \rangle$ the *level* of each F'_j with respect to F is defined to be $l+1$. Each attribute of level 1 is said to be at *root level*. The level of an attribute A in $\mathbf{Attrib}(F)$ is denoted by $\mathbf{level}_F(A)$, or by $\mathbf{level}(A)$ when F is understood.

We now turn to the instance of a form. Using recursion, an *instance* of a form $F = \langle F_1, \ldots, F_r \rangle$ is a finite set $\mathcal{I}$ of functions

$$\begin{aligned}
&f : \mathbf{Free}(F) \cup \{F_j | F_j \text{ a form }\} \longrightarrow\\
&\bigcup\{\text{Dom}(A) | A \text{ in } \mathbf{Free}(F)\} \cup \{Ins(F_j) | F_j \text{ a form}\},
\end{aligned}$$

such that $f(A)$ is in $\text{Dom}(A)$ for each A in $\mathbf{Free}(F)$ and $f(F_j)$ is in $Ins(F_j)$ for each form F_j. Let $Ins(F)$ be the set of all instances of F.

Figure 1 illustrates an instance $\mathcal{I}$ of the (PERSON) form. The set $\mathcal{I}$ consists of nine functions $f_1, \ldots, f_9$, representing the information in the compartments 1 through 9 under the double line. f_1 is the function over

$$\{\text{ENO, DNO, NAME, PHONE, JC, (KIDS), (SCHOOL), SEX, LOC}\}$$

defined by

$$\begin{array}{ccc}
f_1(\text{ENO}) = 05, & f_1(\text{DNO}) = D1, & f_1(\text{NAME}) = SMITH,\\
f_1(\text{PHONE}) = 5555, & f_1(\text{JC}) = 05, & f_1(\text{SEX}) = F,\\
f_1(\text{LOC}) = SF, & f_1((\text{KIDS})) = \mathcal{I}_{11} & f_1((\text{SCHOOL})) = \mathcal{I}_{12}.
\end{array}$$

Here, $\mathcal{I}_{11}$ is an instance of (KIDS) that consists of the functions f_{111}, f_{112} defined by

$$\begin{array}{l}
f_{111}(\text{KNAME}) = JOHN,\ f_{111}(\text{AGE}) = 02,\\
f_{112}(\text{KNAME}) = MARY,\ f_{112}(\text{AGE}) = 04.
\end{array}$$

$\mathcal{I}_{12}$ is the instance of (SCHOOL) consisting of the function f_{121}, where

$$f_{121}(\text{SNAME}) = \text{PRINCETON}, f_{121}((\text{ENROLL})) = \mathcal{I}_{1211}.$$

$\mathcal{I}_{1211}$ is the instance of the form (ENROLL) consisting of the functions f_{12111} and f_{12112} over {YEARIN,YEAROUT}, where

$$f_{12111}(\text{YEARIN}) = 1966,\ f_{12111}(\text{YEAROUT}) = 1970,$$
$$f_{12112}(\text{YEARIN}) = 1972,\ f_{12112}(\text{YEAROUT}) = 1976.$$

The remaining functions $f_2, \ldots, f_9$ are defined similarly.

For each form F, let $Ins_\infty(F) = \bigcup\{\mathcal{I} | \mathcal{I} \text{ in } Ins(F)\}$, that is, $Ins_\infty(F)$ is the set of all functions which are in at least one instance of F. An *instantiated form* is a pair $(\mathcal{I}, F)$, where F is a form and $\mathcal{I}$ is an instance of F.

A set of instantiated forms $\mathcal{B} = \{(\mathcal{I}, F) | F \text{ in } \mathcal{F}\}$ will be referred to as an *instantation* of $\mathcal{F}$.

4 The Restructuring Problem

We now turn to a precise statement of the basic problem [2]. The actual formalization is rather complicated and requires a number of concepts. A central role will be played by the notions of "direct solution" and of "solution" introduced in Section 7.

We start by recalling some notions from relational database theory. Let X be a finite nonempty set of basic attributes. A *tuple* (over X) is a function f from X into $\bigcup_{A \text{ in } X} \text{Dom}(A)$ such that $f(A)$ is in $\text{Dom}(A)$ for all A in X. A *relation* is a pair (I, X), or I when X is understood, where X is a nonempty set of basic attributes and I is a finite set of tuples over X. If $(\mathcal{I}, \langle B_1, \ldots, B_n \rangle)$ is a flat instantiated form, then $B_1, \ldots, B_n$ are basic attributes and $(\mathcal{I}, \{B_1, \ldots, B_n\})$ is a relation.

For each nonempty subset Y of X, the *projection* over Y is the function Π_Y from the set of tuples over X to the set of tuples over Y defined by $\Pi_Y(f) = g$, where f is a tuple over X and g is the tuple over Y for which $g(A) = f(A)$ for all A in Y.

Our interest in tuples over a set of basic attributes stems from the following concept. By recursion, for each form $F = \langle F_1 \ldots, F_r \rangle$ and each $f \in Ins_\infty(F)$, let $K(f)$, called the *kernel* of f, be the set of all tuples g over $\mathbf{BA}(F)$ such that

(i) $g(A) = f(A)$ for each A in $\mathbf{Free}(F)$ and

(ii) g over $\mathbf{BA}(F_j)$, F_j a form, is a tuple in $K(f(F_j))$.

For each $\mathcal{I}$ in $Ins(F)$, the set $\bigcup\{K(f) | f \text{ in } \mathcal{I}\}$ denoted by $K(\mathcal{I}, F)$, is the *kernel* of $(\mathcal{I}, F)$.

Obviously, each kernel is a finite set of tuples. If F is a flat form, then $K(\mathcal{I}, F) = \mathcal{I}$. When F is clear from the context we shall write $K(\mathcal{I})$ instead of $K(\mathcal{I}, F)$.

Given a form F and a finite set R of tuples over $\mathbf{BA}(F)$ there may be more than one instantiated form $(\mathcal{I}, F)$ such that $K(\mathcal{I}, F) = R$. For example, consider the form $F = \langle A, F_1, F_2 \rangle$, where $F_1 = \langle B \rangle$ and $F_2 = \langle C \rangle$, and the set R of tuples over $\mathbf{BA}(F)$ is

[2] The authors are indebted to Dr. Marc Gyssens for helping to clarify the original incorrect formulation of the basic problem.

A	B	C
a	b_1	c
a	b_2	c
a	b_1	c_1
a	b	c_2

Both instantiated forms $(\mathcal{I}_1, F)$ and $(\mathcal{I}_2, F)$ given below have their kernels equal to R.

$(\mathcal{I}_1, F) =$

F		
A	F_1	F_2
	B	C
a	b_1	c
	b_2	
a	b_1	c_1
a	b	c_2

$(\mathcal{I}_2, F) =$

F		
A	F_1	F_2
	B	C
a	b_1	c
		c_1
a	b_2	c
a	b	c_2

The set of paths of F, denoted by $\mathbf{Paths}(F)$, is defined inductively on the number k of subforms of F. Specifically, if $k = 0$, then F is a flat form and $\mathbf{Paths}(F) = \{F\}$. Continuing by induction, let F be the form $F = \langle F_1, \ldots, F_r \rangle$. If $F_{i_1}, \ldots, F_{i_m}$ are the subforms of F at the root level, then each of them has fewer than k subforms. This allows us to define $\mathbf{Paths}(F)$ as

$$\mathbf{Paths}(F) = \{\{H_j\}_{0 \le j \le \ell} | H_0 = F \text{ and } \{H_j\}_{1 \le j \le \ell} \text{ in } \mathbf{Paths}(F_{i_p}), F_{i_p} \text{ a form in } F \text{ for } 1 \le p \le m\}.$$

Each path π in $\mathbf{Paths}(F)$ generates a (unique) branch F_π obtained by removing from F all form attributes, together with their descendants, that do not occur in π. We shall use the notation $\mathbf{BA}(\pi)$ for $\mathbf{BA}(F_\pi)$.

5 Basic Types of Restructuring Operators

As mentioned in Section 3, the data structures of interest to us are basically hierarchical structures of arbitrary depth or width. Relational tables are treated as hierarchies of one level, and networks are assumed to have been decomposed into families of trees. In this section, restructuring operations specifically designed to work on hierarchical structures are discussed.

In transforming a hierarchical structure to a different one, the Automatic Restructurer makes use of four basic types of tree operations: trimming, flattening, stretching, and grafting. When applied to a particular situation, each of these four types of operations causes a restructuring to take place - an operation that produces one output file from either one (in the case of trimming, flattening and stretching) or two (in the case of grafting) input files.

These four types of operations were called **SELECT**, **SLICE**, **CONSOLIDATE** and **GRAFT** in the CONVERT system [32]. (The names have been changed here for mnemonic purposes.) Experience has shown that the data extraction, manipulation and restructuring needs of the real world can be handled by a *combination* of these four types of operations. Similar operations were later proposed in [20, 42, 13]. Query languages designed for "nested tables" or "non

first normal form relational databases" generally have capabilities to perform functions similar to the basic operations discussed here. (See [2, 3, 10, 14, 26, 28] for example.)

The following examines the functions of the four basic types of operations.

5.1 Trimming

A trimming operation removes unwanted components from a form or a subform. Data is extracted from the input and placed in the output without any change in the hierarchical relationships. An example of trimming is presented in Figures 6 and 7. Figure 6 shows the structural transformation, and 7 the instances involved.

We now consider "trimming" in a formal manner.

A form $G = \langle G_1 \ldots G_s \rangle$ is a *trimming* of the form $F = \langle F_1, \ldots, F_r \rangle$ if $G \neq F$ and there is a one-to-one partial function h, called a *trimming operation*, from $\mathbf{Attrib}(F)$ onto $\mathbf{Attrib}(G)$ with the following properties:

(Tr 1) $h^{-1}(B) = B$ for each basic attribute B in $\mathbf{Attrib}(G)$.

(Tr 2) $h^{-1}(\mathbf{Par}(B)) = \mathbf{Par}(h^{-1}(B))$ for each proper attribute B of G.

(Tr 3) $h^{-1}(G) = F$.

Symbolically, we write $h(F) = G$.

Notice that each permutation of siblings at the root level of a form F, or at the root level of some subform of F, is a trimming of F. Also, the composition of two trimmings is a trimming; that is, if G is a trimming of F (by h_1) and H is a trimming of G (by h_2), then H is a trimming of F (by h_2h_1, where h_2h_1 is the function obtained by first applying h_1 and then h_2).

We note the following result, whose proof is given in Appendix B.

Proposition 1. (Trimming Proposition) *Let $G = \langle G_1 \ldots G_s \rangle$ be a trimming of $F = \langle F_1, \ldots, F_r \rangle$ by h. Let j be such that $h(F_j)$ exists. Then*

(a) There exists $k(j)$ such that $h(F_j) = G_{k(j)}$.

(b) $G_{k(j)}$ is a basic attribute if and only if F_j is a basic attribute, in which case $G_{k(j)} = F_j$.

(c) $h(\mathbf{Attrib}(F_j)) = \mathbf{Attrib}(G_{k(j)})$.

(d) $G_{k(j)}$ is a trimming of F_j by h.

Now let $G = \langle G_1, \ldots, G_s \rangle$ be a trimming of $F = \langle F_1, \ldots, F_r \rangle$ by h. Using (a), (b) and (d) of the Trimming Proposition, let $h^{Ins(F)}$ be the mapping from $Ins_\infty(F)$ to $Ins_\infty(G)$ where, for each f in $Ins_\infty(F)$, $h^{Ins(F)}(f) = g$, g defined as follows:

(1) $g(A) = f(A)$ for all A in $\mathbf{Free}(G) = \{G_k | G_k \text{ a basic attribute}\}$.

(2) For each subform F_j such that $h(F_j)$ exists, $h(F_j) = G_{k(j)}$ is a subform of G and $G_{k(j)}$ is a trimming of F_j by h. Recursively, $h^{Ins(F_j)}$ maps $Ins_\infty(F_j)$ into $Ins_\infty(G_{k(j)})$. Let $g(G_{k(j)}) = h^{Ins(F_j)}(f(F_j))$. Clearly, $g(G_{k(j)})$ is in $Ins(G_{k(j)})$.

The following result is also shown in Appendix B.

Theorem 2. (Trimming Theorem) *Let G be a trimming of F by h. Then $K(h^{Ins(F)}(\mathcal{I})) = \Pi_{BA(G)}(K(\mathcal{I}))$ for each $\mathcal{I}$ in* $\mathrm{Ins}(F)$.

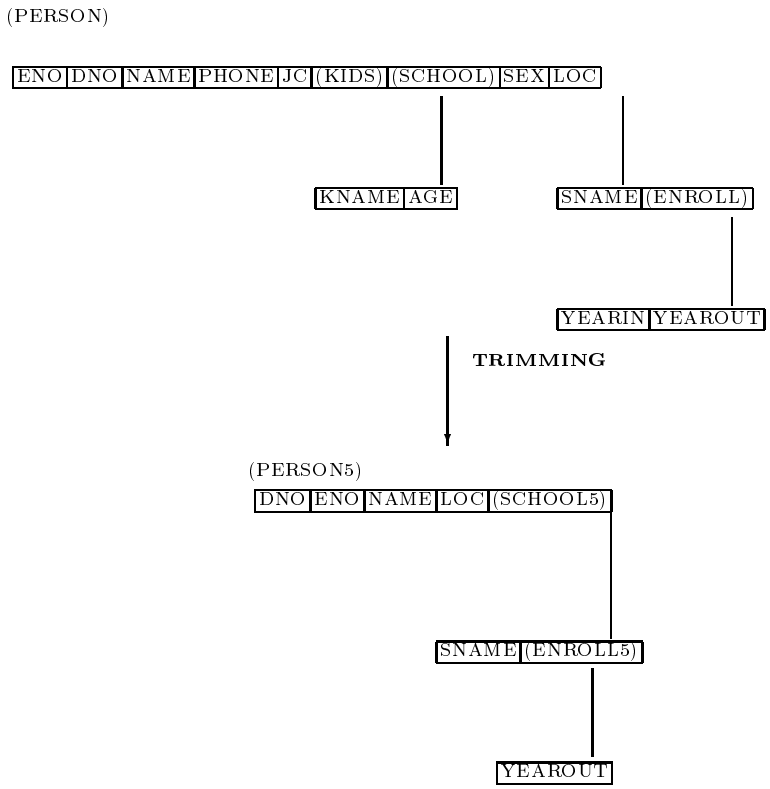


Figure 6: Structural transformation by **TRIMMING**

5.2 Flattening

A flattening operation produces a flat form from a branch of an input by extracting all information at the level of the values of the basic attributes of the branch. The relative positioning of these attributes in the output need not be the same as that in the input. An example of flattening is shown in Figures 8 and 9.

From a theoretical standpoint, it is simpler to restrict the shape of the input to be a branch. In case the input has the shape of a tree, it is not difficult to

(PERSON)											
ENO	DNO	NAME	PHONE	JC	(KIDS)		SCHOOL)			SEX	LOC
					KNAME	AGE	SNAME	(ENROLL)			
								YEARIN	YEAROUT		
05	D1	SMITH	5555	05	JOHN MARY	02 04	PRINCETON	1966 1972	1970 1976	F	SF
07	D1	JONES	5555	05	DICK JANE	07 04	SJS BERKELEY	1960 1965	1965 1969	F	SF
11	D1	ENGEL	2568	05	RITA	04	UCLA	1970	1974	F	LA
		⋮									

TRIMMING

(PERSON5)					
DNO	ENO	NAME	LOC	(SCHOOL5)	
				SNAME	(ENROLL5)
					YEAROUT
D1	05	SMITH	SF	PRINCETON	1970 1976
D1	07	JONES	SF	SJS BERKELEY	1965 1969
D1	11	ENGEL	LA	UCLA	1974
		⋮			

Figure 7: TRIMMING with instance displayed

envisage a two-step process: First apply a trimming operation to obtain an intermediate form containing only the relevant branch with the desired attributes. Then apply a flattening operation on the intermediate form to produce the desired flat output[3]. Thus, in the formal treatment that follows, we assume that the input form has the shape of a branch, and a trimming operation has already been applied if necessary.

A form $G = \langle G_1, \ldots, G_s \rangle$ is a *flattening* of the form F if $G \neq F$, F is a branch and there is a one-to-one partial function h (called a *flattening operation*) from $\mathbf{Attrib}(F)$ onto $\mathbf{Attrib}(G)$ with the following properties:

(Fl 1) Each G_i is a basic attribute, i.e., G is a flat form.

(Fl 2) $\mathbf{BA}(G) = \mathbf{BA}(F)$.

(Fl 3) $h^{-1}(G_i) = G_i$ for each G_i.

(Fl 4) $h^{-1}(G) = F$.

By (Fl 3) and (Fl 4), $h(C)$ is defined only for C in $\{F\} \cup \mathbf{BA}(G)$.

Theorem 3. (Flattening Theorem) *If G is a flattening of F by h and $\mathcal{I}$ is an instance of F, then $K(h^{\mathrm{Ins}(F)}(\mathcal{I}), G) = K(\mathcal{I}, F)$.*

[3] From the software point of view, however, this two-step process is not as efficient as directly applying a flattening operation to a branch of a tree (i.e., without first trimming the tree). Therefore, the implementation enables the direct application of a flattening operation to a branch regardless of whether the input form is tree-shaped or branch-shaped.

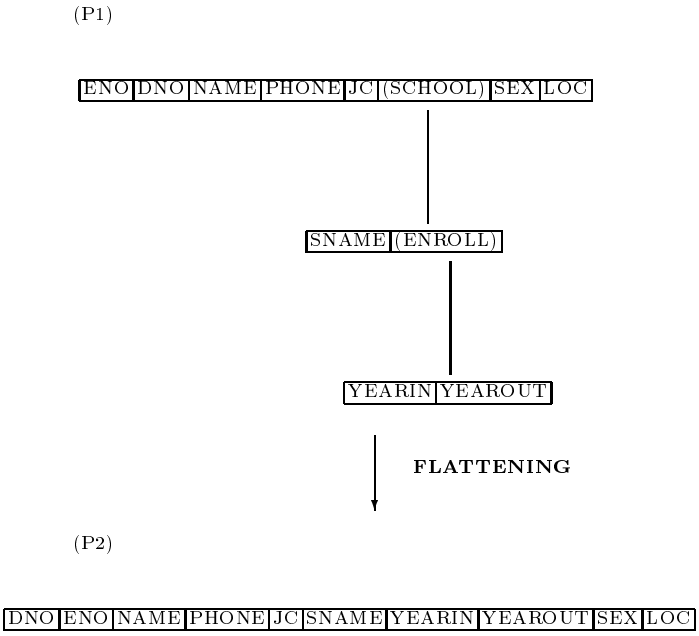


Figure 8: Structural transformation by flattening

(P1)

ENO	DNO	NAME	PHONE	JC	(SCHOOL)			SEX	LOC
					SNAME	(ENROLL)			
						YEARIN	YEAROUT		
05	D1	SMITH	5555	05	PRINCETON	1966	1970	F	SF
						1972	1976		
07	D1	JONES	5555	05	SJS	1960	1965	F	SF
					BERKELEY	1965	1969		
11	D1	ENGEL	2568	05	UCLA	1970	1974	F	LA
		⋮							

FLATTENING

(P2)

DNO	ENO	NAME	PHONE	JC	SNAME	YEARIN	YEAROUT	SEX	LOC
D1	05	SMITH	5555	05	PRINCETON	1966	1970	F	SF
D1	05	SMITH	5555	05	PRINCETON	1972	1976	F	SF
D1	07	JONES	5555	05	SJS	1960	1965	F	SF
D1	07	JONES	5555	05	BERKELEY	1965	1969	F	SF
D1	11	ENGEL	5555	05	UCLA	1970	1974	F	LA
	⋮								

Figure 9: Flattening with instance displayed

Proof. Let G be a flattening of F by h. For each f in $Ins_\infty(F)$, let $h^{Ins(F)}(f) = \Pi_{\mathbf{BA}(G)}(K(f))$. For each $\mathcal{I}$ in $Ins(F)$, let $h^{Ins(F)}(\mathcal{I}) = \bigcup\{h^{Ins(F)}(f) | f \text{ is in } \mathcal{I}\}$. Thus, we obtain $h^{Ins(F)}(\mathcal{I}) = \Pi_{\mathbf{BA}(G)}(K(\mathcal{I}))$. Clearly, $h^{Ins(F)}(\mathcal{I})$ is in $Ins(G)$. Since G is a flat form, $K(\mathcal{J}) = \mathcal{J}$ for each $\mathcal{J}$ in $Ins(G)$. Hence,

$$K(h^{Ins(F)}(\mathcal{I})) = h^{Ins(F)}(\mathcal{I}) = \Pi_{\mathbf{BA}(G)}(K(\mathcal{I})).$$

5.3 Stretching

A stretching operation produces an output with more hierarchical levels than the input. The effect is to factor out the common values in the occurrences. In terms of (hierarchical) trees, this operation forms a taller tree. The additional hierarchical levels must be formed from attributes *existing at the root level of the input.* The names of the new subforms are specified by the user (in the description of the output structure, i.e., the output form heading) or by the Restructurer (when an intermediate form is required). An example of stretching is shown in Figure 10 where the output (PERSON7) has two more levels than the input (PERSON). The new levels are formed by factoring out the common values of DNO and, within each unique DNO, the common values of LOC. (BRANCHES) and (EMPLOYEE) are new subforms.

In more detail, a form $G = \langle G_1, \ldots, G_s \rangle$ is said to be a *stretching* of the form $F = \langle F_1, \ldots, F_r \rangle$ if there exists a subform $G' = \langle G'_1, \ldots, G'_q \rangle$ of G with the following properties:

(St 1) The forms in the set $\{F_1, \ldots, F_r\}$ coincide with the forms in the set $\{G'_1, \ldots, G'_q\}$.

(St 2) The siblings of G', and the siblings of each ancestor of G', together with the basic attributes in the set $\{G'_1, \ldots, G'_q\}$, coincide with the basic attributes in the set $\{F_1, \ldots, F_r\}$, i.e., coincide with $\mathbf{Free}(F)$.

From (St 1) and (St 2), it follows that $\mathbf{BA}(F) = \mathbf{BA}(G)$.

The transformation of F into G is said to be a *stretching operation.* Expressed otherwise, G is said to be a stretching of F by a stretching operation.

To simplify the discussion of instances, we present a slightly different formulation of stretching.

Proposition 4. (Stretching Proposition) *A form G is a stretching of the form $F = \langle F_1, \ldots, F_r \rangle$ if and only if there exists a sequence $G^l = \langle G^l_1, \ldots, G^l_{s(l)} \rangle$, $0 \leq l \leq t$, where $t > 0$ and $G^0 = G$, with the following properties:*

(a) The forms in $\{G^t_1, \ldots, G^t_{s(t)}\}$ coincide with the forms in $\{F_1, \ldots, F_r\}$.

(b) For each l, $0 \leq l < t$, exactly one G^l_i is a form, say $G^l_{i(l)}$, and $G^l_{i(l)} = G^{l+1}$.

(c) The basic attributes in $\{F_1, \ldots, F_r\}$ coincide with the basic attributes in $\{G^i_j |$ for all $i, j\}$.

Proof. Indeed, suppose $\langle G_1, \ldots, G_s \rangle = G = G^0 = \langle G^0_1, \ldots, G^0_{s(0)} \rangle$ is a stretching of F. Then there exists a subform $G' = \langle G'_1, \ldots, G'_q \rangle$ satisfying (St 1) and (St 2). Let $G^1 = \langle G^1_1, \ldots, G^1_{s(1)} \rangle$ be the form in $\{G^0_1, \ldots, G^0_{s(0)}\}$ such that either $G^1 = G'$ or G^1 is an ancestor of G' . By (St 2), all the siblings of G^1 are

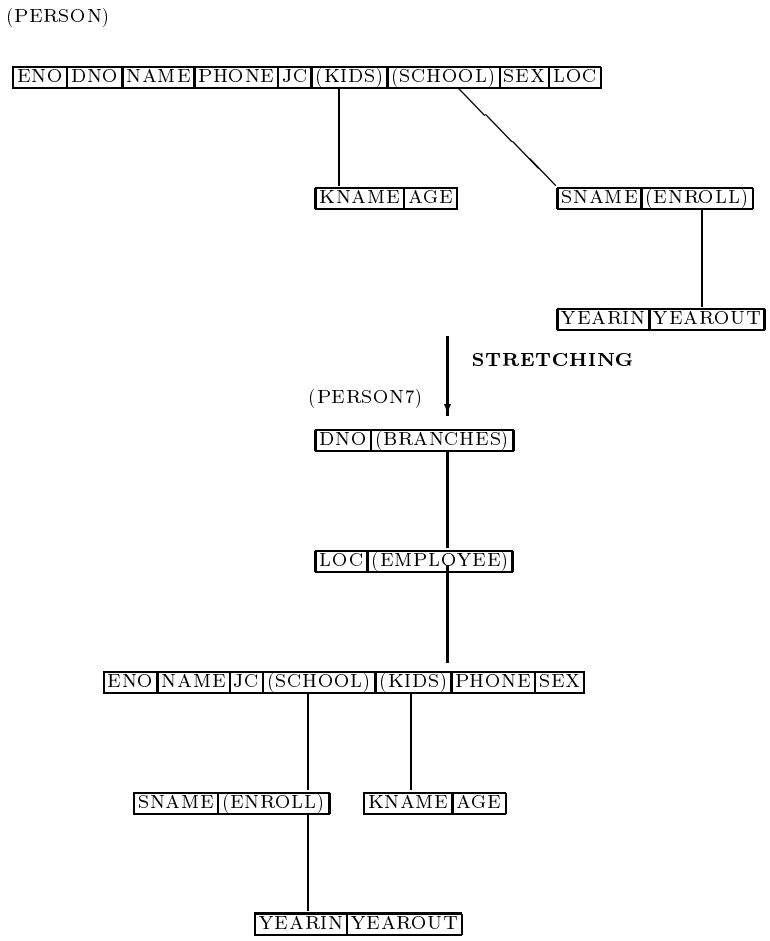


Figure 10: Structural transformation by stretching

basic attributes in $\{F_1, \ldots, F_r\}$. Now repeat the procedure for $l \geq 1$, obtaining $G^l = \langle G^l_1, \ldots, G^l_{s(l)} \rangle$ from G^{l-1}, until $G^l = G'$. Let t be the value of l for which $G^l = G'$. Clearly, $t > 0$ and the sequence $G^l = \langle G^l_1, \ldots, G^l_{s(l)} \rangle$ (where $0 \leq l \leq t$) satisfies (a) - (c).

Now suppose $G^l = \langle G^l_1, \ldots, G^l_{s(l)} \rangle$ (where $0 \leq l \leq t$, $t > 0$ and $G^0 = G$) satisfies (a) - (c). Let $G' = \langle G'_1, \ldots, G'_q \rangle = G^t = \langle G^t_1, \ldots, G^t_{s(t)} \rangle$. Clearly, conditions (St 1) and (St 2) hold. Hence, G is a stretching of F. ■

Let G be a stretching of $F = \langle F_1, \ldots, F_r \rangle$ by a stretching operation h and $G^l = \langle G^l_1, \ldots, G^l_{s(l)} \rangle$ (where $0 \leq l \leq t$) a sequence of forms satisfying (a) - (c) of the Stretching Proposition. For each function f in $Ins_\infty(F)$, let $h^{Ins_\infty(F)}(f) = g^0$, where g^0 is the function on $\mathbf{Free}(G^0) \cup \{G^0_{i(0)}\}$ defined by $g^0(A) = f(A)$ for each A in $\mathbf{Free}(G^0) \subseteq \mathbf{Free}(F)$ and $g^0(G^0_{i(0)}) = \{g^1\}$. By induction, for each l, $1 \leq l < t$, let g^l be the function on $\mathbf{Free}(G^l) \cup \{G^l_{i(l)}\}$ defined by $g^l(A) = f(A)$ for each A in $\mathbf{Free}(G^l) \subseteq \mathbf{Free}(F)$ and $g^l(G^l_{i(l)}) = \{g^{l+1}\}$. Let g^t be the function on

$$\begin{aligned}
&\mathbf{Free}(G^t) \cup \{G^t_j | G^t_j \text{ a form }\} \\
&\quad = \mathbf{Free}(G^t) \cup \{F_j | F_j \text{ a form }\}, \\
&\qquad \text{by (a) of the Stretching Proposition} \\
&\quad \subseteq \mathbf{Free}(F) \cup \{F_j | F_j \text{ a form}\}
\end{aligned}$$

defined by $g^t(A) = f(A)$ for each A in $\mathbf{Free}(G^t) \subseteq \mathbf{Free}(F)$ and $g^t(F_j) = f(F_j)$ for each F_j in $\{F_i | F_i \text{ a form }\}$. It is readily seen that :

(i) g^0 is in $Ins_\infty(G)$ and

(ii) $K(g^0)$ is the set of all functions $\bar{g}$ over $\mathbf{BA}(G) = \mathbf{BA}(F)$ such that $\bar{g}(A) = f(A)$ for each A in $\bigcup_{l=0}^{t} \mathbf{Free}(G^l)$ and $\bar{g}(F_j)$ is a tuple in $K(f(F_j))$ for each form F_j.

By definition, $K(f)$ is the set of all functions $\bar{f}$ over $\mathbf{BA}(F)$ such that $\bar{f}(A) = f(A)$ for each A in $\mathbf{Free}(F)$, and $\bar{f}(F_j)$ is a tuple in $K(f(F_j))$ for each form F_j. By (c) of the Stretching Proposition, $K(f) = K(g^0) = K(h^{Ins_\infty(F)}(f))$. From this, we immediately get:

Theorem 5. (Stretching Theorem) *If G is a stretching of F by h and $\mathcal{I}$ is an instance of F, then $K(h^{\mathrm{Ins}(F)}(\mathcal{I}), G) = K(\mathcal{I}, F)$.*

5.4 Grafting

A grafting operation combines two hierarchies horizontally to form a wider hierarchy by matching the values of the common fields at the *root (top) level* of the two inputs.[4] These common basic attributes are used as linkages between two inputs, and are called the *match attributes* for convenience of discussion. In terms of trees, one can view the operation as "grafting" one (hierarchical) tree to another in order to form a new one. An example is shown in Figure 11.

A formal description of grafting is now given. A form $G = \langle G_1, \ldots, G_s \rangle$ is said to be a *grafting* of the forms $E = \langle E_1, \ldots, E_q \rangle$ and $F = \langle F_1, \ldots, F_r \rangle$ if

(Gr 1) There exists some E_i and F_j such that $\mathbf{BA}(E_i) \cap \mathbf{BA}(F_j) \neq \emptyset$.

(Gr 2) For each E_i and F_j such that $\mathbf{BA}(E_i) \cap \mathbf{BA}(F_j) \neq \emptyset$, $E_i = F_j$ and is a basic attribute.

(Gr 3)

$$\langle G_1, \ldots, G_s \rangle = \langle E_1, \ldots, E_q, F_{u(1)}, \ldots, F_{u(s-q)} \rangle,$$

[4] If some of the common basic attributes are not at the root level, then additional operations must be performed before grafting can take place.

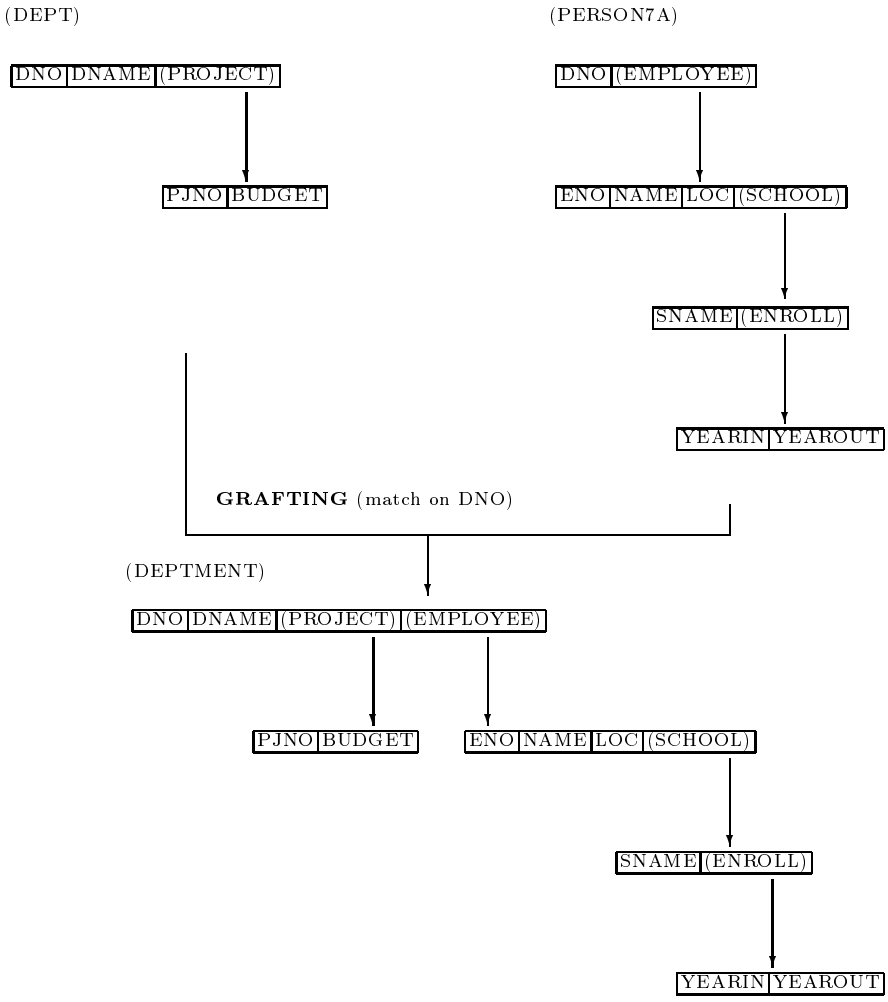


Figure 11: Structural transformation by grafting

where $F_{u(1)}, \ldots, F_{u(s-q)}$ is the *subsequence* of $F_1, \ldots, F_r$ consisting of all F_i not in the set $\{E_1, \ldots, E_q\}$.

The transformation of E and F into G is said to be a *grafting operation.* Expressed otherwise, G is said to be a grafting of E and F by a grafting operation. We denote G by $E \bowtie_{gr} F$.

Let $G = \langle G_1, \ldots, G_s \rangle$ be a grafting of

$$E = \langle E_1, \ldots, E_q \rangle \text{ and } F = \langle F_1, \ldots, F_r \rangle$$

by a grafting operation h. For each e in $Ins_\infty(E)$ and f in $Ins_\infty(F)$ such that $e(A) = f(A)$ for all A in $\mathbf{Free}(E) \cap \mathbf{Free}(F)$, let $h^{Ins(E,F)}(e, f) = g$, where g is the function in $Ins_\infty(G)$ defined as follows:

- $g(A) = e(A)$ for each A in $\mathbf{Free}(E)$.
- $g(A) = f(A)$ for each A in $\mathbf{Free}(F)$.
- $g(G_i) = e(G_i)$ if G_i is a form in $\{E_1, \ldots, E_q\}$.
- $g(G_i) = f(G_i)$ if G_i is a form in $\{F_1, \ldots, F_r\}$.

For each e in $Ins_\infty(E)$ and f in $Ins_\infty(F)$ such that $e(A) \neq f(A)$ for some A in $\mathbf{Free}(E) \cap \mathbf{Free}(F)$, let $h^{Ins(E,F)}(e, f)$ be undefined.

Consider the instance

$$h^{Ins(E,F)}(\mathcal{I}^1, \mathcal{I}^2) = \{h^{Ins(E,F)}(e, f) | e \text{ is in } \mathcal{I}^1, f \text{ is in } \mathcal{I}^2\},$$

denoted by $\mathcal{I}^1 \bowtie_{gr} \mathcal{I}^2$. The following results are established in Appendix C.

Theorem 6. (Grafting Theorem) *Let G be a grafting of E and F, and $\mathcal{I}^1$ and $\mathcal{I}^2$ be instances of E and F, respectively. Then $K(\mathcal{I}^1 \bowtie_{gr} \mathcal{I}^2, E \bowtie_{gr} F) = K(\mathcal{I}^1, E) \bowtie K(\mathcal{I}^2, F)$*

6 Mapping Strategy

In Section 5, we discussed four types of restructuring operations: trimming, flattening, stretching, and grafting. Compared to traditional programming languages, they are indeed "very high level".[5] Nevertheless, while some business applications need only one of these operations, many require sequences of them. In the following, we discuss how the restructuring programs are developed and generated by the Automatic Restructurer. In essence, the Restructurer takes over the "thinking and coding" process by (a) establishing a plan for mapping the specified input(s) to the desired output, and (b) implementing the plan for reasonably efficient execution. The (a) part is discussed in the remainder of this section, and the (b) part is addressed in Section 7.

It is important to keep in mind that the Restructurer is an automatic data restructuring facility for data processing applications, and *not* a database management system. Some concepts of theoretical interest to database management (e.g., key attributes, potential information loss or recoverability, access rights, etc.) are not necessarily important for end users' applications. For example, the ability to distinguish each entity in an entity set (i.e. the key concept) is important for database research, but not for statistical applications. As another example, the recoverability of information is an important issue to database researchers and maintainers, but not to end users. The files created by the end users' applications are generally used to produce reports, to accommodate different points of view, and to aid in the decision-making process. They are created for private use and are not part of the centralized database. These and similar considerations motivated the underlying philosophy of the Restructurer: The

[5] Expansion ratios ranging from 1:18 to 1:175 have been reported [32] for a CONVERT (see the introduction to Section 5) statement to PL/I statements.

user knows what he/she wants. As long as the descriptions of the data structures given to the Restructurer are unambiguous, the Restructurer honors the request. If it is possible to produce an output as the *user-specified*, the Restructurer proceeds to produce it. If no way is found, the Restructurer sends the user a message.

We are now ready to present the mapping strategy. The material is organized into three subsections, dealing respectively with the specifics of the mapping strategy for one, two, or more inputs.

6.1 Transformation Strategy When There Is Only One Input

To determine the strategy for transforming one input to one output, the Restructurer first examines the possibility of applying the following operation(s), in the order shown.[6] For convenience, we shall refer to these six cases as the *first order operations.*

1. trimming
2. flattening
3. flat-trim (i.e., flattening followed by trimming)
4. stretching
5. flat-stretch (i.e., flattening followed by stretching)
6. flat-trim-stretch (i.e., flattening followed by trimming, then stretching)

In other words, if the desired output can be produced by applying trimming, then trimming is chosen to be the operation. Otherwise, the Restructurer examines the next possibility, i.e. flattening. If flattening also fails the applicability test, then the third possibility is tried, and so forth. If a transformation cannot be accomplished by any of the first order operations, then "hybridization" (discussed later) is applied.

The above operations are now discussed in more detail.

Consider (1). To be "trimmable", the hierarchical relationships between the components must not be disturbed during restructuring. In other words, there must be no change of "heritage", even though positioning of components within a subform may be altered, and some of the leaves or branches of a hierarchy may be omitted. To determine whether an output is trimmable from a given source, the tests shown in Figure 12 are applied.

Now consider (2) and (3). If the output cannot be produced by trimming the given source, then the applicability of flattening is examined. When the desired output does not have a flat shape, it obviously cannot be produced by applying flattening to the input. Otherwise, the flattening operation is a natural candidate. However, the following two additional conditions must be satisfied in order for flattening to be applicable:

(a) The source components required for the creation of the output must belong to a single branch. This source branch can be either a branch-shaped input, or a branch-shaped subform within a tree-shaped input.

(b) At least one attribute must be extracted from each level along the branch.

[6] There may be more than one way to map the given input to the desired output. The ordering of the examination discussed here makes sure that the efficiency of the generated program is taken into consideration. (See Section 7.)

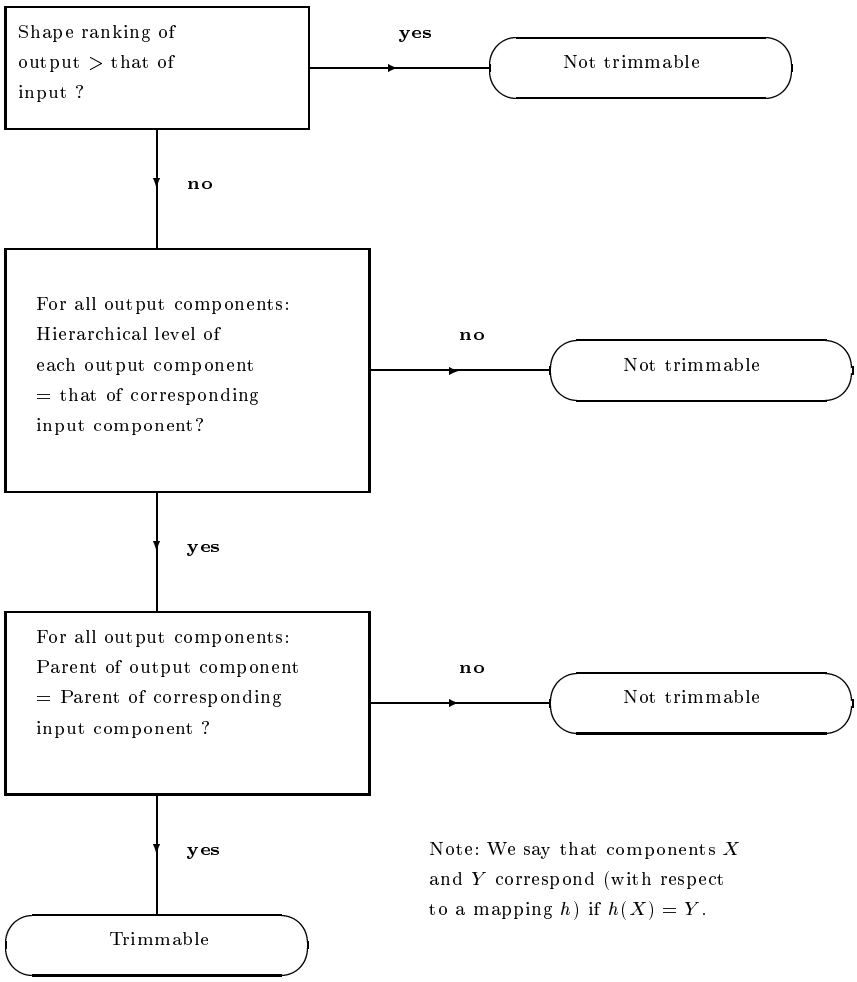
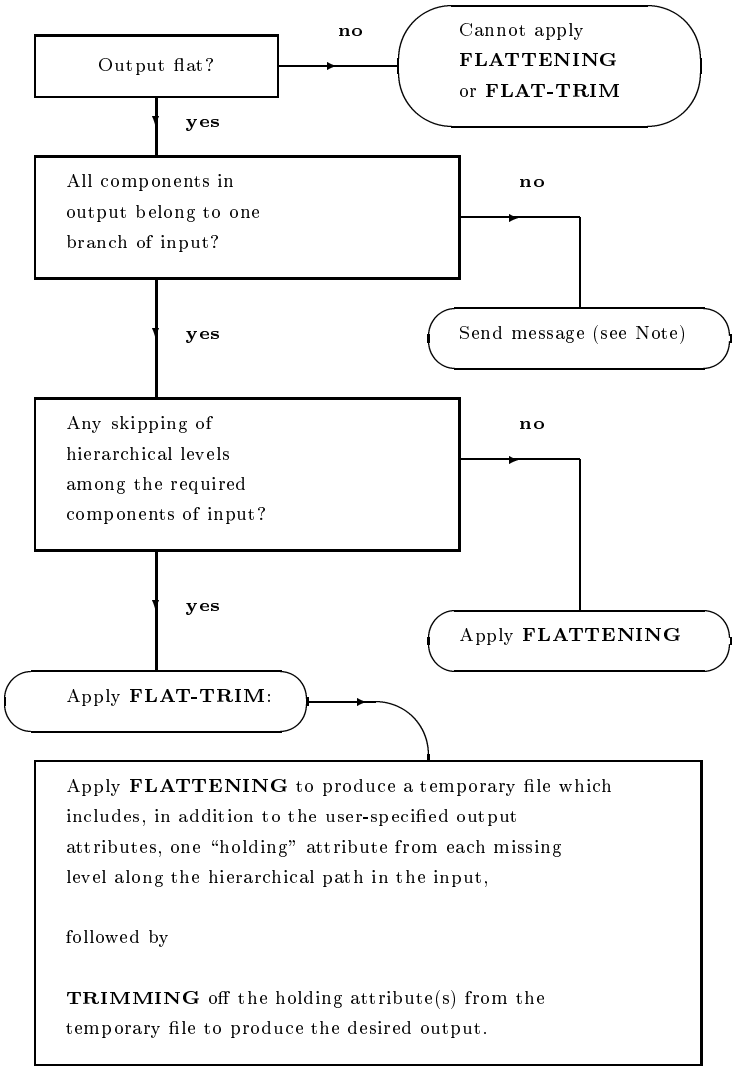


Figure 12: Determining the applicability of trimming

More specifically, the rules in Figure 13 are followed to determine whether case (2) or case (3) is applicable. Figure 14 gives an example for case (3), i.e. **FLAT-TRIM**.

Consider (4). When the output is neither a flat form nor trimmable from the source, the Restructurer explores the possibility of using stretching. To apply this operation, two rules must be observed:

(a) The expansion of new hierarchical levels occurs only from basic attributes at the root level of the source form.



Note: The data contained in each branch of a tree is independent of the data contained in other branches of that tree Therefore, simultaneous flattening of two or more branches of a tree into a flat form is not allowed.

Figure 13: The applicability of flattening and of flat-trim

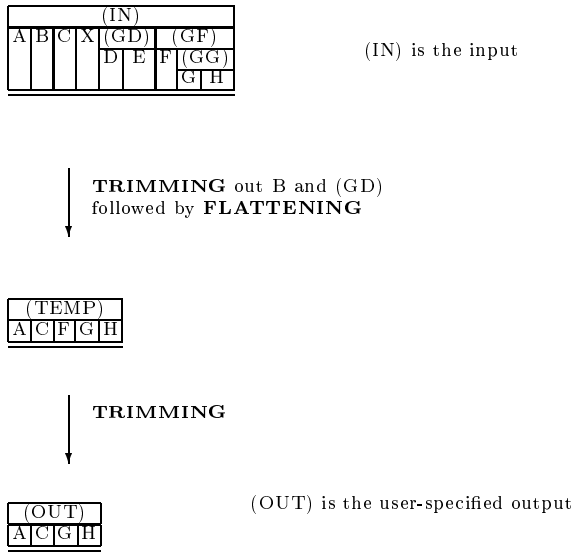


Figure 14: Example of flat-trim

(b) Except for those components participating in (a), *corresponding pairs* of the output and input components must maintain the same hierarchical relationships (i.e. if A is the parent of B in the input, then A must be the parent of B in the output). Furthermore, if a subform is carried from the input to the output, it must be carried in its entirety.

The method to determine whether an output is "stretchable" from the input is shown in Figure 15.

Finally, consider (5) and (6). When all four cases discussed above fail, the shapes of the data structures again come into consideration. Suppose the output is a branch and the source is either a branch form or all the components necessary for the creation of the output are in a single branch of the source tree. Then restructuring can be accomplished by flattening followed by stretching (see Figure 16 for an example), or flat-trim followed by stretching. Otherwise, hybridization is necessary.

The strategy discussed so far is summarized in Figure 17.

In general, if the shape of the output structure is ranked lower than that of the input, one of the first order operations can be applied. Hybridization may be required when both the output and the input are tree-shaped. An example requiring hybridization is given in Figure 18.

Hybridization is carried out in two stages, *decomposition* and *synthesis.* At the decomposition stage, the output data structure is decomposed into its

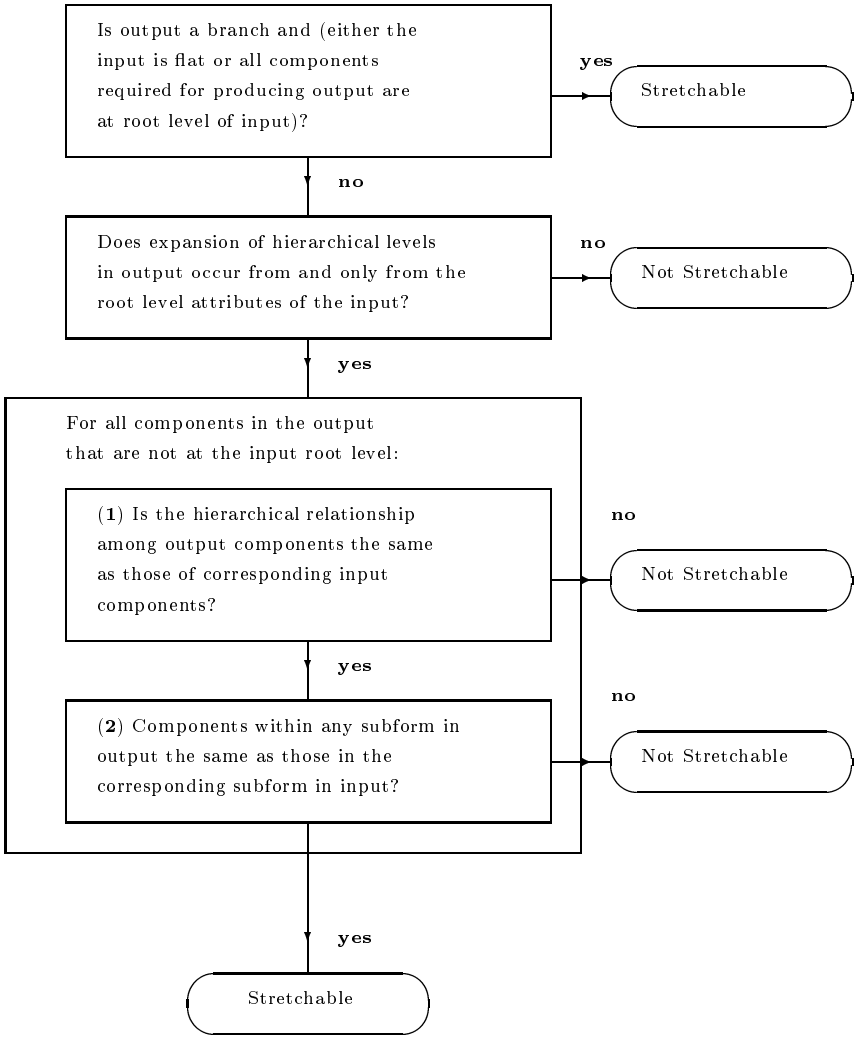


Figure 15: Determining the applicability of stretching

subtrees (i.e., subforms). Each of the subtrees concatenated with the attributes along the hierarchical path is then treated as an intermediate file and compared with its counterpart in the input. These subtrees are components of the desired output, and thus have simpler structures (than that of the output). The data structures of these intermediate files may also be simpler than the input, in which case the first order operations can be applied. If hybridization is again required for the production of any of the intermediate files, then decomposition is applied recursively. Eventually, the Restructurer succeeds in marking all intermediate

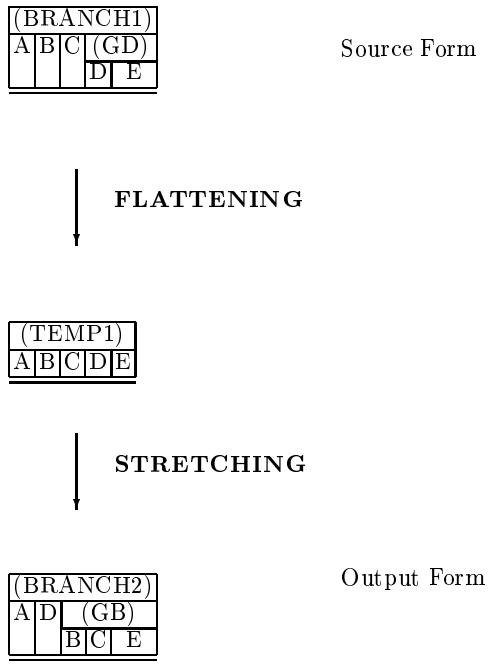


Figure 16: Restructuring of one branch form to another by FLAT-STRETCH

files with first order operations (unless there is no solution, in which case the user is given a message and the process terminates).

The synthesis stage of hybridization begins when the decompositional analysis is complete. Essentially, the marked operations are applied to produce the intermediate files from the input, in a top-to-bottom, left-to-right order. The intermediate files are then grafted pairwise, in a bottom-up, right-to-left order, until the desired output is produced. Grafting two intermediate files is the same as grafting two source (input) files. Details of the grafting operation are discussed in the following subsection.

OUTPUT FORM	INPUT FORM		
	Flat	Branch	Tree
Flat	**TRIMMING**	**TRIMMING** if all required input components are at the root level; otherwise **FLATTENING** or **FLAT-TRIM**(see Figure 13)	
Branch	**STRETCHING**	**TRIMMING** if trimmable by Figure 12; otherwise **STRETCHING** if stretchable by Figure 15 otherwise **FLATTENING** or **FLAT-TRIM** (see Figure 13) followed by **STRETCHING** (i.e. **FLAT-STRETCH** or **FLAT-TRIM-STRETCH**)	
Tree	**HYBRIDIZATION**		**TRIMMING** if trimmable by Figure 12; otherwise **STRETCHING** if stretchable by Figure 15; otherwise **HYBRIDIZATION**

Figure 17: Summary of mapping strategy for one input

6.2 Transformation Strategy When There Are Exactly Two Inputs

Whenever there are two inputs, a grafting is required. A grafting operation ties two hierarchies side by side by matching the values of the corresponding match attributes of the inputs. Except for the match attributes of the second input, the structure of each input is carried in its entirety into the output. No restructuring of any part is allowed during the grafting operation. Thus, in case the components from two source forms are intertwined in the desired output, it is necessary to use an intermediate form as the output of the grafting. A trimming is then applied to the intermediate form to rearrange the juxtaposition of the components according to the user's specification.

Aside from the fact that grafting may require the creation of an intermediate form as discussed above, there are two rules that the grafting operation must observe:

(1) The match attributes must be at the root level of both inputs.

(2) The match attributes must appear at the root level of the output.

Consider (1). If a match attribute is not at the root level of an input, then a flattening operation must be performed before the grafting.

Now, consider (2). Two cases arise:

(2a) All matching attributes appear at the root level of the output.

(2b) Some matching attributes are not at the root level of the output.

In the case of (2a), the Restructurer proceeds according to Figure 19. Briefly, the output data structure is decomposed into two subparts (each corresponding to one of the input forms), with match attributes included in both of the subparts. Each of the subparts is then compared with its corresponding source file. If the structures are the same, then the source file is used directly as input (called the Graft-source in Figure 19) to the grafting operation. Otherwise, the subpart is treated as the appropriate input to the grafting operation, and a transformation from the given source into this subpart is necessary. The problem is reduced to creating an intermediate file from only one input, and the method discussed earlier (in Section 6.1) is followed. A grafting is performed when the two Graft-sources are ready. An example is shown in Figure 20 with DIRECTRY and DEP as the given input and RESDIR as the desired output.

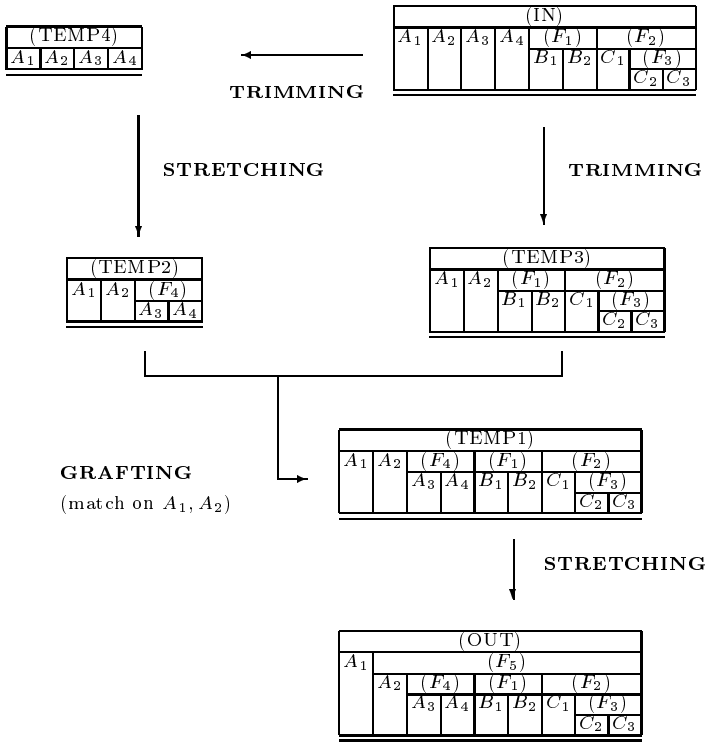


Figure 18: Restructuring of (IN) to (OUT) by hybridization

In the case of (2b), when some match attributes are *not* at the root level of the output, it is necessary to construct an intermediate file with all match attributes appearing at the root level so that a grafting operation can be applied. Take Figure 21 for example. In order to produce NEWDIR (the desired output) from DEP and DIRECTRY files, an intermediate file (TEMP2) is constructed as the output of grafting. Notice that the match field ENO in the intermediate file (TEMP2) is at the root level. The problem is now reduced to grafting where all match attributes appear at the root level of the output. The method described earlier (Figure 19) is followed. A stretching is then applied to transform the output of the grafting (i.e., the intermediate file) into the final desired form.

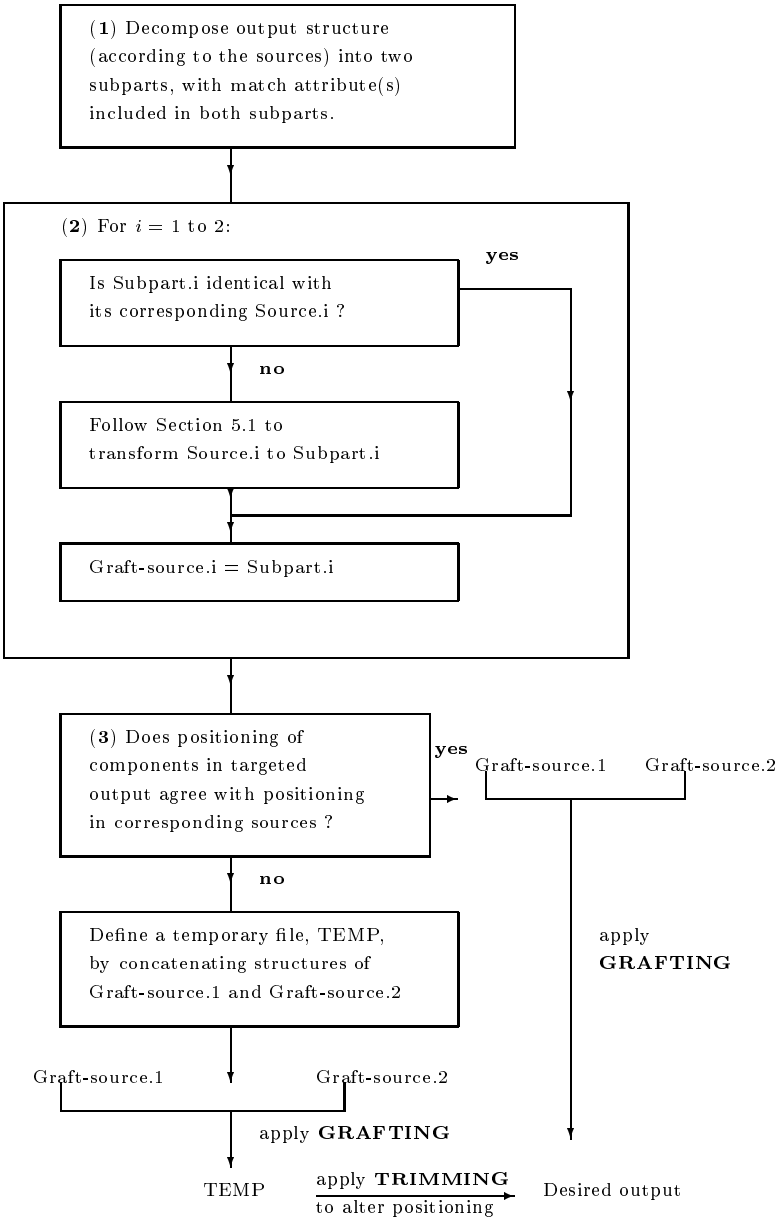


Figure 19: Grafting where all matching attributes appear at root level

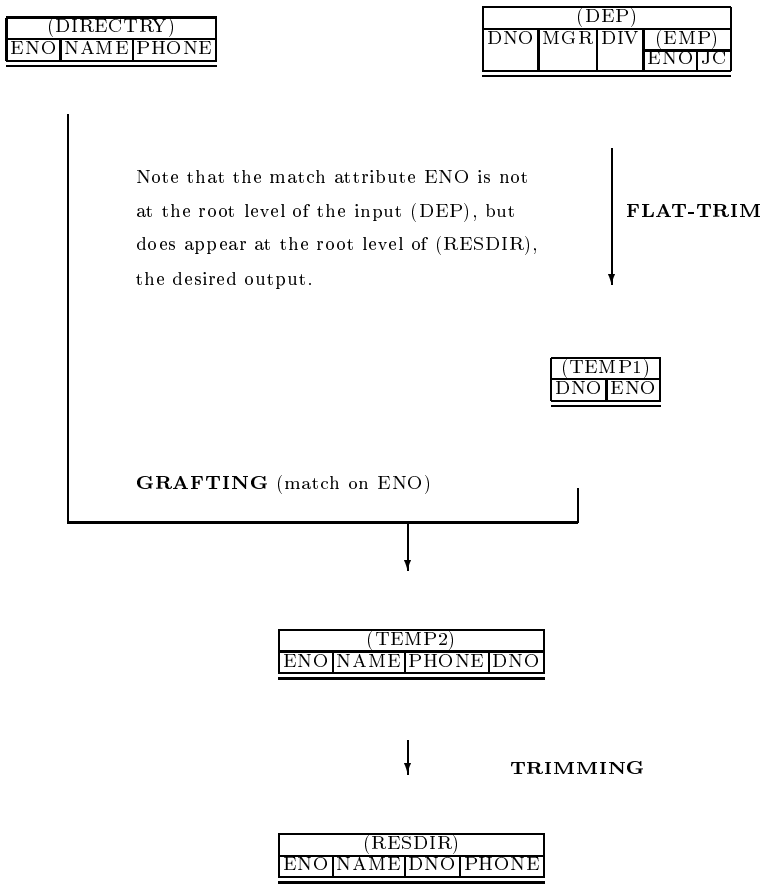


Figure 20: Producing (RESDIR) from (DIRECTRY) and (DEP)

6.3 Transformation Strategy When There Are More Than Two Inputs

The preceding subsection (6.2) discussed the transformation strategy when there are exactly two source files. In case there are more than two, a gradual pairwise build-up can be employed. For example, TFORM can be created from S1, S2, S3, S4 and S5 as shown in Figure 22.

Since the basic concepts underlying this approach have already been covered, no further discussion is necessary.

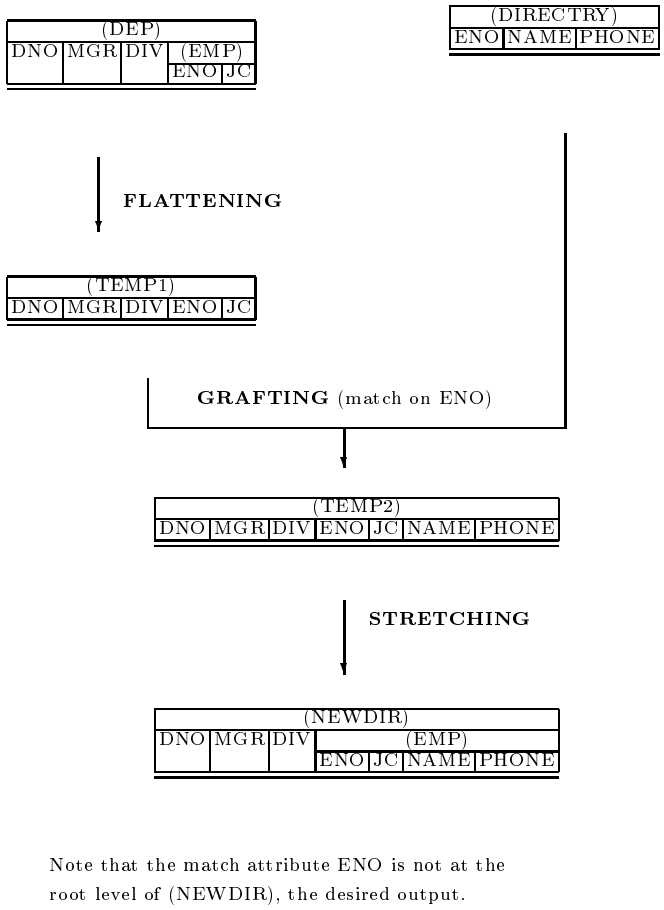


Figure 21: Producing (NEWDIR) from (DIRECTRY) and (DEP)

7 Implementing The Plan

As soon as the strategy for transforming the designated input into the desired output is completed, the second stage of automatic restructuring begins. The main function of the second stage is to produce an executable program to carry out the sequence of basic operations determined in the first stage. In other words, the main function of the first stage is *planning* and of the second stage *constructing.*

The program constructed by the Restructurer consists of (1) a declarative part which defines in target language the data structures of all the files involved in the operation, and (2) an imperative part which spells out the code sequence

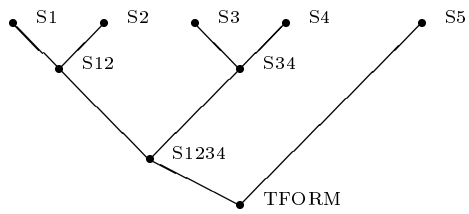


Figure 22: The computation of TFORM

necessary to perform the particular operations at hand. The declarative part of the procedure is produced from information contained in the component description tables. (The component description tables describing inputs are fetched from a catalog of predefined forms and those describing intermediate and output forms are prepared and placed in the catalog at the end of the code generation phase, i.e., the constructing stage). The imperative part is patterned after program models designed for the basic operation types trimming, flattening, stretching, and grafting.

The goal of the Automatic Restructurer is to produce an efficient program tailored for the particular situation at hand. Some efficiency considerations are taken by the Restructure at the planning stage. For example, among the first order operations discussed in Section 6.1, the possibility of applying a trimming operation is considered before all others. Having a trimming operation as the first of a sequence of operations serves two purposes: (1) unwanted instances are filtered out as early as possible (thus reducing the number of instances to be processed by subsequent operations) and (2) unwanted components are trimmed off as early as possible (thus reducing the width of the forms to be processed by subsequent operations).

The main concern at the constructing stage is to make sure that, regardless of the operation type, the generated code (i.e., *the procedure*) for each operation *passes over data once and only once.*[7] The goal is to perform necessary functions with minimum scanning and movement of data. The criterion of passing over data only once for each operation causes no problem for trimming and flattening. For these operations, one can always fetch a record, process it, and go back to fetch another one. For the other two types of operations, one pass over the data is possible only when a certain sort order (or simply *order*) requirement is enforced.[8]. In the case of stretching, input must be sorted according to the fields which form the extended hierarchical level in the output. In the case of grafting, both inputs must be sorted on the match attributes (in the same ordering sequence and the same ascending/descending directions). Thus, before producing

[7] This concern was discussed in [32]

[8] Referring to the first order operations discussed in Section 6.1, since flattening does not require a sorted order while stretching does, the applicability of flattening is examined before the applicability of stretching

a procedure for a stretching or grafting operation, the order requirement for the operation is examined and compared with the sequencing order of the input (if known). If they are not compatible, or the order information on the input is not available, then a sort operation is inserted to enforce the order requirement.

As mentioned above, the unavailability of the order information could mean an unnecessary insertion of a SORT operation (e.g., an intermediate file may already be in the required sort order for the next operation). Therefore, the order information for all intermediate forms is recorded whenever possible. It is easy to see that the trimming and flattening operations are order preserving if (1) the input to the operation is sorted and (2) all the order fields in the input are extracted and placed in the output directly (without the side effects of arithmetic or string functions). The Restructurer detects these situations and propagates the order information accordingly. In the case of the stretching and grafting operations, the resulting order of the output is determinable once the order requirement on the input is enforced. Thus, the Restructurer is able to record the order information for the output of these operations.

When the desired ordering of the user-specified output is known, the Restructurer makes the enforced ordering compatible with the desired ordering whenever possible. If that is not feasible, then a final SORT (and/or a sort within parent) is included in the generated program. In this manner, the transformation strategy determined in the first stage is implemented in the second stage as an executable program (consisting of a set of properly sequenced procedures and SORT commands) to carry out the plan.

Let $\mathcal{B}$ be a finite set of instantiated forms. The notion of $\mathcal{B}$-computation introduced next is intended as a formalization of the automatic restructuring process.

A sequence of instantiated forms $(\mathcal{I}_1, F_1), \ldots, (\mathcal{I}_n, F_n)$ is said to be a *$\mathcal{B}$-computation* of length n if for every i, $1 \leq i \leq n$, one of the following cases occurs:

1. $(\mathcal{I}_i, F_i)$ is in $\mathcal{B}$;
2. there exists an integer j, $1 \leq j < i$ such that $(\mathcal{I}_i, F_i)$ is obtained from $(\mathcal{I}_j, F_j)$ by stretching, flattening, or trimming; or
3. there exist integers j_1, j_2, $1 \leq j_1 \neq j_2 < i$, such that $(\mathcal{I}_i, F_i)$ is obtained from $(\mathcal{I}_{j_1}, F_{j_1})$ and $(\mathcal{I}_{j_2}, F_{j_2})$ by grafting.

If $(\mathcal{I}_1, F_1), \ldots, (\mathcal{I}_n, F_n)$ is a $\mathcal{B}$-computation and $(\mathcal{I}, F) = (\mathcal{I}_n, F_n)$, then the instantiated form $(\mathcal{I}, F)$ is said to be *$\mathcal{B}$-computable*.

Let $\mathcal{F}$ be a finite set of forms. A form G is said to be *$\mathcal{F}$-compatible* if for every π in $\mathbf{Paths}(G)$, there is a subset $\{F_1, \ldots, F_n\}$ of $\mathcal{F}$ and a set of paths $\{\pi_1, \ldots, \pi_n\}$ such that π_i in $\mathbf{Paths}(F_i)$ for $1 \leq i \leq n$ and

$$\mathbf{BA}(\pi) \subseteq \bigcup \{\mathbf{BA}(\pi_i) | 1 \leq i \leq n\}.$$

If $G = F_1 \bowtie_{gr} F_2$, then G is $\{F_1, F_2\}$-compatible by the definition of grafting. Similarly, direct applications of the definitions of the remaining basic restructuring operations show that if G is obtained from F_1 by either stretching, flattening, or trimming, then G is $\{F_1\}$-compatible.

To introduce the notion of solution we need the following concept. A finite set of forms $\mathcal{F}$ is *directly solvable* if $\mathbf{BA}(F) \cap \mathbf{BA}(F') = \mathbf{Free}(F) \cap \mathbf{Free}(F') \neq \emptyset$ for all distinct forms F and F', $F \neq F'$ in $\mathcal{F}$.

Note that every set $\mathcal{F}$ with exactly one form is directly solvable. Furthermore, $\mathcal{F}$ is directly solvable if $\mathcal{F} = \{F_1, F_2\}$ and $F_1 \bowtie_{gr} F_2$ is defined.

Proposition 7. *Let $\mathcal{F} = \{F_1, \ldots, F_n\}$, $n \geq 2$, be a directly solvable set of forms. Then $G = ((F_1 \bowtie_{gr} F_2) \bowtie_{gr} \cdots) \bowtie_{gr} F_n$ is defined, $\mathbf{BA}(G) = \bigcup\{\mathbf{BA}(F_i) | 1 \leq i \leq n\}$, and $\mathbf{Free}(G) = \bigcup\{\mathbf{Free}(F_i) | 1 \leq i \leq n\}$.*

Proof. The argument is by induction on n. The case $n = 2$ is obvious. Now suppose that the statement holds for directly solvable sets of forms with fewer than n elements. Let $\mathcal{F} = \{F_1, \ldots, F_n\}$ be a directly solvable set of forms. Clearly, $\mathcal{F}' = \mathcal{F} - \{F_n\}$ is a directly solvable set of forms, $G' = ((F_1 \bowtie_{gr} F_2) \bowtie_{gr} \cdots) \bowtie_{gr} F_{n-1}$ is defined, $\mathbf{BA}(G') = \bigcup\{\mathbf{BA}(F_i) | 1 \leq i \leq n-1\}$, and $\mathbf{Free}(G') = \bigcup\{\mathbf{Free}(F_i) | 1 \leq i \leq n-1\}$. Then

$$\begin{aligned}
\mathbf{BA}(F_n) \cap \mathbf{BA}(G') &= \mathbf{BA}(F_n) \cap \bigcup\{\mathbf{BA}(F_i) | 1 \leq i \leq n-1\} \\
&= \bigcup\{\mathbf{BA}(F_n) \cap \mathbf{BA}(F_i) | 1 \leq i \leq n-1\} \\
&= \bigcup\{\mathbf{Free}(F_n) \cap \mathbf{Free}(F_i) | 1 \leq i \leq n-1\} \\
&= \mathbf{Free}(F_n) \cap \bigcup\{\mathbf{Free}(F_i) | 1 \leq i \leq n-1\} \\
&= \mathbf{Free}(F_n) \cap \mathbf{Free}(G').
\end{aligned}$$

Thus, the grafting between G' and F_n is defined and the desired form $G = F_n \bowtie_{gr} G'$ can be constructed. Since $\mathbf{BA}(G) = \mathbf{BA}(F_n) \cup \mathbf{BA}(G')$ and $\mathbf{Free}(G) = \mathbf{Free}(F_n) \cup \mathbf{Free}(G')$, the inductive hypothesis implies the desired conclusion. ■

Theorem 8. *Let $\mathcal{F} = \{F_i | 1 \leq i \leq n\}$ be a finite, directly solvable set of forms and F be an $\mathcal{F}$-compatible form. Then for every instantiation $\mathcal{B} = \{(\mathcal{I}_i, F_i) | 1 \leq i \leq n\}$ of $\mathcal{F}$ there exists a $\mathcal{B}$-computable instance $(\mathcal{I}, F)$ of F such that*

$$K(\mathcal{I}, F) = \Pi_{\mathbf{BA}(F)}(\bowtie \{K(\mathcal{I}_i, F_i) | 1 \leq i \leq n\}).$$

Proof. Suppose that F is a branch. Let $\overline{F}$ be the form obtained from F by flattening. Since $\mathbf{BA}(F) = \mathbf{BA}(\overline{F})$, the $\mathcal{F}$-compatibility of F implies the $\mathcal{F}$-compatibility of $\overline{F}$.

Let $G = ((F_1 \bowtie_{gr} F_2) \bowtie_{gr} \cdots) \bowtie_{gr} F_n$ be the form whose existence was established in Proposition 7. Let $\mathcal{B} = \{(\mathcal{I}_1, F_1), \ldots, (\mathcal{I}_n, F_n)\}$ be an instantiation of $\mathcal{F}$. The corresponding instance $(\mathcal{J}, G)$ is given by

$$(\mathcal{J}, G) = ((\mathcal{I}_1, F_1) \bowtie_{gr} (\mathcal{I}_2, F_2) \bowtie_{gr} \cdots) \bowtie_{gr} (\mathcal{I}_n, F_n).$$

Therefore

$$K(\mathcal{J}, G) = K(\mathcal{I}_1, F_1) \bowtie \cdots \bowtie K(\mathcal{I}_n, F_n)$$

by a repeated application of the Grafting Theorem. The $\mathcal{F}$-compatibility of $\overline{F}$ implies $\mathbf{BA}(\overline{F}) \subseteq \mathbf{BA}(G)$, so there is an instance $(\overline{\mathcal{I}}, \overline{F})$ of $\overline{F}$ such that

$$\overline{\mathcal{I}} = K(\overline{\mathcal{I}}, \overline{F}) = \Pi_{\mathbf{BA}(\overline{F})}(K(\mathcal{I}_1, F_1) \bowtie \cdots \bowtie K(\mathcal{I}_n, F_n)).$$

The branch $(\mathcal{I}, F)$ can be obtained from $(\overline{\mathcal{I}}, \overline{F})$ by stretching. The Stretching Theorem implies

$$K(\mathcal{I}, F) = \Pi_{\mathbf{BA}(F)}(K(\mathcal{I}_1, F_1) \bowtie \cdots \bowtie K(\mathcal{I}_n, F_n)),$$

as needed. The argument also shows that $(\mathcal{I}, F)$ is $\mathcal{B}$-computable by a sequence of restructuring operations involving flattening, grafting and stretching.

Now suppose that F is a tree. The branches $G_1, \ldots, G_\ell$ of F can be extracted by trimming. The $\mathcal{F}$-compatibility of F implies the $\mathcal{F}$-compatibility of each of its branches. By the previous argument, for every instantiation $\mathcal{B}$ of $\mathcal{F}$ and for every integer k, $1 \leq k \leq \ell$, there is an instance $(\mathcal{J}_k, G_k)$ of G_k such that

$$K(\mathcal{J}_k, G_k) = \Pi_{\mathbf{BA}(G_k)}(\bowtie \{K(\mathcal{I}_i, F_i) | 1 \leq i \leq n\}).$$

The grafting of the instances $(\mathcal{J}_k, G_k)$ yields an instance of $(\mathcal{I}, F)$. By the Grafting Theorem we then have

$$\begin{aligned} K(\mathcal{I}, F) &= K(\mathcal{J}_1, G_1) \bowtie \cdots \bowtie K(\mathcal{J}_\ell, G_\ell) \\ &= \Pi_{\mathbf{BA}(G_1)}(\bowtie \{K(\mathcal{I}_i, F_i) | 1 \leq i \leq n\}) \bowtie \cdots \\ &\quad \bowtie \Pi_{\mathbf{BA}(G_\ell)}(\bowtie \{K(\mathcal{I}_i, F_i) | 1 \leq i \leq n\}) \\ &= \Pi_{\mathbf{BA}(F)}(\bowtie \{K(\mathcal{I}_i, F_i) | 1 \leq i \leq n\}), \end{aligned}$$

which gives the desired equality for trees.

The existence of the $\mathcal{B}$-computation for $(\mathcal{I}, F)$ is implicit in this argument. Specifically, this computation extends the computations of the instances of the branches of F and grafts these instances to obtain an instance of F. ■

The instantiated form $(\mathcal{I}, F)$ in Theorem 8 will be referred to as an *F-direct solution* of $\mathcal{B}$, or simply as a *direct solution* of $\mathcal{B}$ when F is clear from the context. The previous theorem shows that if $\mathcal{F}$ is a directly solvable set of forms and F is an $\mathcal{F}$-compatible form, then for every instantiation $\mathcal{B}$ of $\mathcal{F}$, the Restructures can produce a direct solution.

We now extend the notion of direct solution. Let $\mathcal{B}$ be a finite set of instantiated forms and F a form. An *F-solution* of $\mathcal{B}$ is defined inductively as follows:

1. Every instantiated form $(\mathcal{I}, F)$ in $\mathcal{B}$ is a solution of $\mathcal{B}$.
2. Every direct solution of $\mathcal{B}$ is a solution of $\mathcal{B}$.
3. If $(\mathcal{I}_1, F_1), \ldots, (\mathcal{I}_n, F_n)$ are solutions of $\mathcal{B}$, $\{F_1, \ldots, F_n\}$ is a directly solvable set of forms and F is $\{F_1, \ldots, F_n\}$-compatible, then every F-direct solution $(\mathcal{I}, F)$ of $\{F_1, \ldots, F_n\}$ is a solution of $\mathcal{B}$.

We shall refer to an F-solution of $\mathcal{B}$ as a *solution of* $\mathcal{B}$ when F is understood from the context.

The next theorem shows that the Restructurer produces a solution if and only if a solution exists.

Theorem 9. *Let $\mathcal{B}$ be a finite set of instantiated forms.*

If $(\mathcal{I}, F)$ is a $\mathcal{B}$-computable form, then $(\mathcal{I}, F)$ is a solution of $\mathcal{B}$.

Conversely, if $\mathcal{B}$ is a finite set of forms and F is a form such that there is an F-solution of $\mathcal{B}$, then there exists a $\mathcal{B}$-computation that generates an instantiated form $(\mathcal{I}, F)$.

Proof. Suppose that $\mathcal{B}$ is a finite set of instantiated forms and $(\mathcal{I}, F)$ is a $\mathcal{B}$-computable form. Then there exists a $\mathcal{B}$-computation $(\mathcal{I}_1, F_1), \ldots, (\mathcal{I}_n, F_n)$ such that $(\mathcal{I}, F) = (\mathcal{I}_n, F_n)$. We prove by induction that $(\mathcal{I}, F)$ is a solution of $\mathcal{B}$.

If $n = 1$ then $(\mathcal{I}, F)$ belongs to $\mathcal{B}$ by definition. Suppose that the theorem holds for sequences of length less than n. Several subcases of the inductive case need to be considered, depending on the basic restructuring operation applied in constructing $(\mathcal{I}_n, F_n)$.

Suppose that $(\mathcal{I}_n, F_n) = (\mathcal{I}_{j_1}, F_{j_1}) \bowtie_{gr} (\mathcal{I}_{j_2}, F_{j_2})$, where $j_1, j_2 < n$. By the inductive hypothesis, $(\mathcal{I}_{j_1}, F_{j_1})$ and $(\mathcal{I}_{j_2}, F_{j_2})$ are solutions of $\mathcal{B}$. The definition of the grafting operation implies that $\{F_{j_1}, F_{j_2}\}$ is directly solvable, and that $(\mathcal{I}_n, F_n)$ is an F_n-direct solution of $\{(\mathcal{I}_{j_1}, F_{j_1}), (\mathcal{I}_{j_2}, F_{j_2})\}$. We leave to the reader the argument of the remaining subcases when $(\mathcal{I}_n, F_n)$ is obtained from $(\mathcal{I}_j, F_j)$, $j < n$, by stretching, flattening, or trimming.

Conversely, suppose that $(\mathcal{I}, F)$ is a solution of $\mathcal{B}$. If $(\mathcal{I}, F)$ belongs to $\mathcal{B}$, then $(\mathcal{I}, F)$ is clearly $\mathcal{B}$-computable. Also, if $(\mathcal{I}, F)$ is a direct solution of $\mathcal{B}$, then Theorem 8 implies that $(\mathcal{I}, F)$ is $\mathcal{B}$-computable. Finally, suppose that $(\mathcal{I}, F)$ is a direct solution of the set $\mathcal{B}' = \{(\mathcal{I}_1, F_1), \ldots, (\mathcal{I}_n, F_n)\}$, where $(\mathcal{I}_i, F_i)$ is a solution of $\mathcal{B}$ for $1 \leq i \leq n$, $\{F_1, \ldots, F_n\}$ is a directly solvable set of forms, and F is $\{F_1, \ldots, F_n\}$-compatible. Suppose that each instantiated form $(\mathcal{I}_i, F_i)$ is $\mathcal{B}$-computable. Theorem 8 implies that $(\mathcal{I}, F)$ is $\mathcal{B}'$-computable. Therefore, by concatenating the n $\mathcal{B}$-computations of $(\mathcal{I}_1, F_1), \ldots, (\mathcal{I}_n, F_n)$ with the $\mathcal{B}'$-computation of $(\mathcal{I}, F)$ one obtains a $\mathcal{B}$-computation of $(\mathcal{I}, F)$, so $(\mathcal{I}, F)$ is $\mathcal{B}$-computable. ■

8 Summary

Most information-processing applications involve data restructuring of various degrees. The effort of writing application programs can be substantially reduced when the effort of producing data restructuring code is carried out automatically. This paper discusses the Automatic Restructurer, a mechanism that relieves the user from the tedium of writing a substantial amount of code.

The Restructurer was implemented at the IBM Los Angeles Scientific Center as part of an application development system called FORMAL [34] The capability of generating an executable program for the desired transformation depends mainly on the unambiguous descriptions of input and output data structures. The process of generating code is accomplished in two stages. In the first, the goal of an application is recognized, the differences between the input and output data structures examined, and the applicabilities of the rules and methods for transformation analyzed. The result is a plan for converting the input into the output. In the second stage, construction begins. Embedded knowledge of the target system is utilized to implement the plan efficiently. The result is a tailored executable program capable of transforming the input to the desired output.

Given this automatic data restructuring capability, complicated information-processing applications can be specified in a simple and truly non-procedural fashion [34, 36] and the non-trivial task of data restructuring can be accomplished in a systematic manner.

We introduce the notion of a solution of a set of instantiated forms with respect to an output form. This allows us to characterize those instantiated forms that can be obtained as a result of recasting the information contained in

a set of instantiated forms. Furthermore, we prove that if such a solution exists, then the Restructurer will always produce a solution. (If the Restructurer states that it is unable to restructure data contained by the set of forms, then no solution exists for that set of forms with respect to the desired output form.)

9 Acknowledgements

S. Ginsburg's contribution was supported in part as a consultant to the IBM Los Angeles Scientific Center and in part by the National Science Foundation under grant CCR-8618907.

John Kepler and Jim Jordan's management support is very much appreciated. The authors are indebted to Dr. Marc Gyssens for helping to clarify the original incorrect formulation of the basic problem.

References

[1]. Abiteboul, S. and Bidoit, N.: *Non First Normal Form Relations to Represent Hierarchically Organized Data*, Proc. ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, April 1984, pp. 191-200.

[2]. Abiteboul, S. and Bidoit, N.: *Non First Normal Form Relations: An Algebra Allowing Data Restructuring*, Journal of Computer and System Sciences, 1986, vol. 33, pp. 361-393.

[3]. Abiteboul, S. and Hull, R.: *Restructuring Hierarchical Database Objects*, INRIA Rapports de Recherche No. 615, 1987, To appear in Theoretical Computer Science.

[4]. Atkinson, M.P., and Buneman, O.P.: *Types and Persistence in Database Programming Languages*, ACM Computing Surveys, 1987 vol. 19 (2), pp. 105-190.

[5]. Barr, A. and Feigenbaum, E.A.: *The Handbook of Artificial Intelligence* vol.2, 1982. William Kaufmann, Inc.

[6]. Burnett M. B., Ambler, A. L.: *Interactive Visual Data Abstraction in a Declarative Programming Language*, Journal of Visual Languages and Computing, 1994, vol. 5, pp. 29–60.

[7]. Carter, D.A., Collins, J.W. and Khandewal, V.K.: *Data Conversion and Restructuring at Australian Iron and Steel PTY LTD using XPRS*, Australasian Share Guide Meeting, July 1982.

[8]. Chang, S. K.: *A Visual Language Compiler for Information Retrieval by Visual Reasoning*, IEEE Transactions on Software Engineering, 1990, v. 16, pp. 1136-1149.

[9]. Colby,L.S.:*A Recursive Algebra and Query Optimization for Nested Relations*, Proceedings of 1989 ACM SIGMOD Conference on Management of Data, 1989, pp. 273-283.

[10]. Dadam, P., Kuespert, F., Andersen, H., Blanken, H., Erbe, R., Guenauer, J., Lum., V., Pistor, P. and Walch, G.: *A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies*, Proceedings of the 1986 ACM-SIGMOD Conference on Management of Data, pp. 356-357.

[11]. Deshpande, A., and Van Gucht, D.:*An implementation for Nested Relational Databases*, Proceedings of the 14th VLDB Conference, 1988, pp. 76-87.

[12]. Fisher, P., Thomas, S.: *Operators for Non-First-Normal-Form Relations*, Proceedings of COMPSAC, 1983, pp. 464-475.

[13]. Guting, R.H., Zicari, R., and Chopy, D.M.: *An Algebra for Structured Office Documents*, IBM Research Report RJ5559, 1987.

[14]. Hull, R. *A Survey of Theoretical Research on Typed Complex Database Objects* in *Databases* (edited by J. Paredaens), Academic Press, 1987, London, pp. 193-256.

[15]. Hull, R. and Su, J.: *On the Expressive Power of Database Queries with Intermediate Types*, Proceedings of the ACM Symposium on Principles of Database Systems, 1988, pp. 39-51.

[16]. *Data Extraction, Processing and Restructuring System: Define and CONVERT Reference Manual*, SH20-2178, Program Number 5796-PLH, 1979, IBM Corporation.

[17]. Jaeschke, G. and Schek, H.J.: *Remarks on the Algebra of Non First Normal Form Relations*, Proc. ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1982, pp. 124-138.

[18]. Jaeschke, G.: *Nonrecursive Algebra for Relations with Relation-Valued Attributes*, IBM Heidelberg Scientific Center Technical Report TR 85.03.001, March 1985.

[19]. Jaeschke, G.: *Recursive Algebra for Relations with Relation-Valued Attributes*, IBM Heidelberg Scientific Center Technical Report, TR 85.03.002, March 1985.

[20]. Kitagawa, H., Kunii, T. and Ishii, Y., *Design and Implementation of a Form Management System APAD Using ADABAS INQ DBMS*, Proc. COMPSAC 81, Nov.1981, pp. 324-334.

[21]. Kitagawa, H., Gotoh, M., Misaki, S. and Azuma, M.: *Form Document Management System SPECDOQ - Its Architecture and Implementation*, Proc. of the Second ACM Conf. on Office Information Systems, June 1984, pp. 132-142.

[22]. Leitheiser, R.L., and Wetherbe, J.C.: *Approaches to End-User Computing: Service May Spell Success*, Journal of Information Systems Management, Winter 1986, pp.9-14.

[23]. Luo, D., and Yao, S.B.: *Form Operation by Example - A Language for Office Information Processing*, Proceedings of SIGMOD Conference, 1981, pp. 213-223.

[24]. Mohan L., Kashyap, R. L.: *A Visual Language for Graphical Interaction with Schema-Intensive Databases*, IEEE Transactions on Knowledge and Data Engineering, 1993, 5, pp. 843-858.

[25]. Navathe, S.B. and Fry, J.P.: *Restructuring for Large Databases: Three Levels of Abstraction*, ACM Trans. Database Systems, 1976, vol. 1 (2), pp. 138-158.

[26]. Pistor, P., and Andersen, F.: *Designing a Generalized NF^2 Model with an SQL-Type Language Interface*, Proceedings of 12th Conf. on VLDB, 1986, pp. 278-288.

[27]. Rockart, J.F., and Flannery, L.S.: *The Management of End User Computing*, Communications of the ACM, Oct 1983, vol.26 (10), pp. 776-784.

[28]. Roth, M.A., Korth, H.F., and Batory, D.S.: *SQL/NF: A Query Language for $\neg 1NF$ Relational Databases*, Information Systems, 1987, vol. 12 (1), pp. 99-114.

[29]. Roth, M.A. and Korth, H.F., *The Design of $\neg 1NF$ Relational Databases into Nested Normal Form*, Dept. of Computer Sciences, Univ. of Texas at Austin, Technical Report TR-86-27, Dec 1986.

[30]. Roth, M.A., Korth, H.F., and Silberschatz, A.: *Extended Algebra and Calculus for Nested Relational Databases*, ACM Transactions on Database Systems, Dec. 1988, vol. 13 (4), pp. 389-417.

[31]. Shu, N.C., Housel, B.C. and Lum, V.Y.: *CONVERT: A High Level Translation Definition Language for Data Conversion*, Communications of ACM, Oct 1975, vol. 18 (10), pp 557-567.

[32]. Shu, N.C., Housel, B.C., Taylor, R.W., Ghosh, S.P., and Lum, V.Y.: *EXPRESS: A Data Extraction, Processing, and Restructuring System*, ACM Trans. Database Systems, June 1977, vol. 2 (2), pp. 134-174.

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

[34]. Shu, N.C.: *FORMAL: A Forms-Oriented, Visual-Directed Application Development System*, IEEE COMPUTER, Aug. 1985, vol. 18 (8), pp. 38-49.

[35]. Shu, N.C.: *Automatic Data Transformation and Restructuring*, Proceedings of IEEE Third Data Engineering Conference, Feb., 1987, pp. 173-180.

[36]. Shu, N.C.: *Visual Programming*, Van Nostrand Reinhold Company, Inc. 1988, NewYork.
[37]. Shu, N.C.: *A Visual Programming Language Designed for Automatic Programming*, Proc. of 21st Annual Hawaii International Conf. on Systems Sciences, Jan. 1988, pp. 662-671.
[38]. Siau, K. L., Chan, H.C., Tan, K. P.: *Visual Knowledge Query Language*, IEICE Transactions on Information and Systems, 1992, E75-D, pp. 697–703.
[39]. Sockut, G.H. and Goldberg, R.P.: *Database Reorganization - Principles and Practice*, ACM Computing Surveys, Dec 1979, vol.11 (4), pp. 371-395.
[40]. Swartwout, D.E., Deppe, M.E. and Fry, J.P.: *Operational Software for Restructuring Network Databases*, Proc. National Computer Conference, June 1977, pp. 499-508.
[41]. Tansel, A.U. and Garnett, L.:*Nested Historical Relations* Proc. 1989 ACM SIGMOD Conference, 1989, pp. 284-293.
[42]. Thomas, S.J. and Fischer, P.C.: *Operators for Non First Normal Form Relations*, Proc. COMPSAC, 1983, pp. 464-475.
[43]. Ullman, J.D.: *Principles of Database and Knowledge-Base Systems. Vol.I*, Computer Science Press, 1988, Maryland.
[44]. Valduriez, P.:*Complex Objects in Relational Database Systems*, Technology and Science of Informatics, 1987, vol. 6(8), pp. 597-608.

A A Program to Produce (XMASLIST) from (PERSON)

The following is a CONVERT program [32] which produces (XMASLIST) shown in Figure 2 from (PERSON) shown in Figure 1. The Automatic Restructurer is able to generate this program from the FORMAL specification shown in Figure 4. (Recall that the SLICE and CONSOLIDATE operations used in CONVERT are equivalent to the flattening and stretching operations discussed in Section 4.)

```
/**  PROCESS NAME = XMASLIST **/

FORM PERSON   (
  ENO CH(3),
  DNO CH(2),
  NAME CH(8),
  PHONE CH(4),
  JC CH(2),
  KIDS(
      KNAME CH(8),
      AGE CH(2)
      ) RG,
  SCHOOL(
      SNAME CH(9),
      ENROLL(
          YEARIN CH(4),
          YEAROUT CH(2)
          ) RG
          (ORDERED ON (ENROLL.YEARIN ASC))
      ) RG,
  SEX CH(1),
```

```
  LOC CH(2)
  )
  :ORDERED ON (DNO ASC,ENO ASC);


$P1F2 = SLICE (
      NAME
      ,LOC
      ,KNAME
      ,AGE
      FROM PERSON);


FORM $P1F2 (
  NAME CH(8),
  LOC CH(2),
  KIDS(
      KNAME CH(8),
      AGE CH(2)
      ) NRG
  );


/* IT IS NECESSARY TO SORT $P1F2 ON CONCATENATED KEYS
   OF OUTPUT FOR CONSOLIDATE OPERATION.   */

FORM $P1F3 (
  NAME CH(8),
  LOC CH(2),
  KIDS(
      KNAME CH(8),
      AGE CH(2)
      ) NRG
  )
  :ORDERED ON (LOC ASC,KIDS.AGE ASC);


$P1F3 = SORT ($P1F2 BY
              LOC ASC
             ,AGE ASC
      );


XMASLIST = CONSOLIDATE ($P1F3);


FORM XMASLIST (
  LOC CH(2),
  RECEIVER(
      AGE CH(2),
```

```
        KID(
            KNAME CH(8),
            NAME CH(8)
            ) RG
        ) RG
        (KEY IS (RECEIVER.AGE),
         ORDERED ON (RECEIVER.AGE ASC))
    )
    :KEY IS (LOC),
     ORDERED ON (LOC ASC),
     DISPOSITION IS (INTERNAL);

  END;
```

B Proofs of the Trim Proposition and Theorem

In order to establish the Trim Proposition, we first prove:

Proposition 10. *Let G be a trimming of F by h.*

(a) Suppose B and C are in $\mathbf{Attrib}(F)$, $B = \mathbf{Par}(C)$, *and $h(B)$ and $h(C)$ exist. Then $h(B) = \mathbf{Par}(h(C))$, i.e., $h(C)$ is an offspring of $h(B)$.*

(b) Suppose $h(C)$ exists. Then $h(B)$ exists for every ancestor B of C, and is an ancestor of $h(C)$.

(c) Suppose $h(B)$ and $h(C)$ exist, and C is a descendent of B. Then $h(C)$ is a descendent of $h(B)$.

(d) Suppose $h(C)$ is a descendent of $h(B)$. Then C is a descendent of B. (Alternatively, if $h(B)$ is an ancestor of $h(C)$, then B is an ancestor of C).

Proof. Let $G = \langle G_1, \ldots, G_s \rangle$ and $F = \langle F_1, \ldots, F_r \rangle$.

(a) Suppose $h(C) = G$. Then $C = h^{-1}(h(C)) = h^{-1}(G) = F$ (by (Tr 3)), so $B = \mathbf{Par}(C) = Par(F)$. This is a contradiction, since F has no parent. Thus, $h(C) \neq G$. Therefore, $\mathbf{Par}(h(C))$ exists. Hence,

$$\begin{aligned} h^{-1}(\mathbf{Par}(h(C))) &= \mathbf{Par}(h^{-1}(h(C)))(\text{by (Tr 2)}) \\ &= \mathbf{Par}(C) = B = h^{-1}(h(B)). \end{aligned}$$

Since h is one-to-one, thus h^{-1} is one-to-one, $\mathbf{Par}(h(C)) = h(B)$.

(b) Using induction and (a), it suffices to show that $h(B)$ exists for $B = \mathbf{Par}(C)$. Suppose $\mathbf{Par}(h(C))$ does not exist. Then $h(C) = G$, since G is a form. Hence, $C = h^{-1}(h(C)) = h^{-1}(G) = F$. This contradicts the fact that C has a parent. Thus, $h(B) = h(\mathbf{Par}(C)) = \mathbf{Par}(h(C))$ exists.

(c) By (b), $h(B)$ is an ancestor of $h(C)$, i.e., $h(C)$ is a descendent of $h(B)$.

(d) This statement follows by induction from the fact that $h^{-1}(\mathbf{Par}(D)) = \mathbf{Par}(h^{-1}(D))$ for each proper attribute D of G. ■

Trimming Proposition. Let

$$G = \langle G_1, \ldots, G_s \rangle$$

be a trimming of $F = \langle F_1, \ldots, F_r \rangle$ by h. Let j be such that $h(F_j)$ exists. Then

(a) there exists $k(j)$ such that $h(F_j) = G_{k(j)}$.

(b) $G_{k(j)}$ is a basic attribute if and only if F_j is a basic attribute, in which case $G_{k(j)} = F_j$.

(c) $h(\mathbf{Attrib}(F_j)) = \mathbf{Attrib}(G_{k(j)})$.

(d) $G_{k(j)}$ is a trimming of F_j by h (restricted to $\mathbf{Attrib}(F_j)$).

Proof. (a). Since $F = \mathbf{Par}(F_j)$, $h(F) = G = \mathbf{Par}(h(F_j))$ by (a) of Proposition 10. By definition, each offspring of G is one of the G_i. Hence, $F_j = G_{k(j)}$ for some $k(j)$.

(b). By (Tr 1), $G_{k(j)} = h^{-1}(G_{k(j)}) = F_j$ if $G_{k(j)}$ is a basic attribute. Suppose $G_{k(j)}$ is a form. Then there is a descendent C of $G_{k(j)}$ which is a basic attribute. By (d) of Proposition 10, $C = h^{-1}(C)$ is a descendent of $h^{-1}(G_{k(j)}) = F_j$. Since every ancestor of a basic attribute is a form, F_j is a form.

(c) This follows immediately from (c) and (d) of Proposition 10.

(d) This follows from (c) and (Tr 1) - (Tr 3). ■

We now show:

Trimming Theorem Let G be a trimming of F by h. Then $K(h^{Ins(F)}(\mathcal{I})) = \Pi_{\mathbf{BA}(G)}(K(\mathcal{I}))$ for each $\mathcal{I}$ in $Ins(F)$.

Proof. Suppose F is a flat form. Then for each f in $\mathcal{I}$, f is a tuple over $\mathbf{Free}(F) = \mathbf{BA}(F)$ and $h^{Ins(F)}(f) = \Pi_{\mathbf{BA}(G)}(f)$. Hence,

$$\begin{aligned} K(h^{Ins(F)}(\mathcal{I})) &= K(\{h^{Ins(F)}(f) | f \text{ is in } \mathcal{I}\}) \\ &= K(\{\Pi_{\mathbf{BA}(G)}(f) | f \text{ is in } \mathcal{I}\}) \\ &= \Pi_{\mathbf{BA}(G)}(\bigcup \{K(f) | f \text{ is in } \mathcal{I}\}), \text{ since } f \text{ is a tuple over } \mathbf{BA}(F) \\ &= \Pi_{\mathbf{BA}(G)}(K(\mathcal{I})). \end{aligned}$$

Continuing by induction suppose that $K(h^{Ins(F)}(\mathcal{I}')) = \Pi_{\mathbf{BA}(G')}(K(\mathcal{I}'))$ for each subform F' of F and each instance $\mathcal{I}'$ of F', where G' is the trimming of F' by h as guaranteed by (d) of the Trim Proposition. To extend the induction, it suffices to show that $K(h^{Ins(F)}(f)) = \Pi_{\mathbf{BA}(G)}(K(f))$ for each f in $Ins_\infty(F)$.

Let $G = \langle G_1, \ldots, G_s \rangle$ and $F = \langle F_1, \ldots, F_r \rangle$. Two cases arise.

(α) Suppose e is in $K(h^{Ins(F)}(f))$. Now $h^{Ins(F)}(f) = g$, where g is the function in $Ins_\infty(G)$ defined by $g(A) = f(A)$ for each A in $\mathbf{Free}(G)$ and $g(G_{k(j)}) = h^{Ins(\langle F_j \rangle)}(f(F_j))$ for each form F_j such that $h(F_j)$ exists. By definition, $e(A) =$

$g(A) = f(A)$ for each A in $\mathbf{Free}(G)$ and e over $\mathbf{BA}(G_{k(j)})$, where $G_{k(j)}$ is a form, is a tuple in

$$\begin{aligned} K(g(G_{k(j)})) &= \Pi_{\mathbf{BA}(G_{k(j)})}(K(f(F_j))) \\ &= K(h^{Ins(F_j)}(f(F_j))), \text{ by induction.} \end{aligned}$$

Thus, e is in $\Pi_{\mathbf{BA}(G)}(K(f))$.

(β) Suppose e is in $\Pi_{\mathbf{BA}(G)}(K(f))$. Then there exists d in $K(f)$ such that $e = \Pi_{\mathbf{BA}(G)}(d)$. By definition, $d(A) = f(A)$ for each A in $\mathbf{Free}(F)$ and d over $\mathbf{BA}(F_j)$, where F_j a form, is a tuple in $K(f(F_j))$. Then $e(A) = d(A) = f(A)$ for each A in $\mathbf{Free}(G)$ and, by induction, e over $\mathbf{BA}(G_{k(j)})$ is in

$$\Pi_{\mathbf{BA}(G_{k(j)})} K(f(F_j)) = K(h^{Ins(F)}(f(F_j)))$$

for each $k(j)$ such that $G_{k(j)}$ is a form. Hence e is in $K(h^{Ins(F)}(f))$. ■

C Proof of Grafting Theorem

Grafting Theorem. Let G be a grafting of E and F, and $\mathcal{I}^1$ and $\mathcal{I}^2$ be instances of E and F, respectively. Then $K(\mathcal{I}^1 \bowtie_{gr} \mathcal{I}^2, E \bowtie_{gr} F) = K(\mathcal{I}^1, E) \bowtie K(\mathcal{I}^2, F)$

Proof. Let $E = \langle E_1, \ldots, E_q \rangle$, $F = \langle F_1, \ldots, F_r \rangle$, and let $G = \langle G_1, \ldots, G_s \rangle$ be the grafting of the forms E and F. If ℓ in $K(\mathcal{I}^1 \bowtie_{gr} \mathcal{I}^2, E \bowtie_{gr} F)$, there is g in $\mathcal{I}^1 \bowtie_{gr} \mathcal{I}^2$ such that ℓ belongs to $K(g)$. By the definition of the grafting operation, there is e in $\mathcal{I}_1$ and f in $\mathcal{I}_2$ such that $g(A) = e(A) = f(A)$ for all A in $\mathbf{Free}(E) = \mathbf{Free}(F)$, $g(E') = e(E')$ for every direct descendent E' of E that is a form and $g(F') = f(F')$ for every direct descendent F' of F that is a form. We claim that $\Pi_{\mathbf{BA}(E)}(K(g)) = \Pi_{\mathbf{BA}(E)}(K(e))$ and that $\Pi_{\mathbf{BA}(F)}(K(g)) = \Pi_{\mathbf{BA}(F)}(K(f))$. Indeed, $K(g)$ is the set of all functions k over $\mathbf{BA}(G)$ such that $k(A) = g(A)$ for all A in $\mathbf{Free}(G)$ and k over $BA(G_j)$ (where G_j is a form) is a tuple in $K(g(G_j))$. If G_j is a form that is a direct descendent of E, say $E_{i(j)}$, then k over $\mathbf{BA}(G_j)$ is a tuple in $K(g(E_{i(j)}))$. Thus, $\Pi_{\mathbf{BA}(E)}(K(g))$ is the set of all functions over $\mathbf{BA}(E)$ such that $k(A) = e(A)$ for all A in $\mathbf{Free}(E) \subseteq \mathbf{Free}(G)$ and k over $\mathbf{BA}(G_j)$ (where G_j is a direct descendent of E) is a tuple in $K(g(E_{i(j)}))$. Therefore, $\Pi_{\mathbf{BA}(E)}(K(g)) = K(e)$. Note that $K(g) = \Pi_{\mathbf{BA}(E)}(K(g)) \bowtie \Pi_{\mathbf{BA}(F)}(K(g)) = \Pi_{\mathbf{BA}(E)}(K(e)) \bowtie \Pi_{\mathbf{BA}(F)}(K(f))$, which implies ℓ in $K(\mathcal{I}^1, E) \bowtie K(\mathcal{I}^2, F)$.

Conversely, let ℓ in $K(\mathcal{I}^1, E) \bowtie K(\mathcal{I}^2, F)$. There are e, f in $\mathcal{I}^1, \mathcal{I}^2$, respectively such that $\ell(A) = e_1(A)$ for all A in $\mathbf{BA}(E)$ and $\ell(B) = f_1(B)$ for all B in $\mathbf{BA}(F)$ for some e_1 in $K(e)$ and f_1 in $K(f)$. Let g be defined by $g(A) = e(A) = f(A)$ for every A in $\mathbf{Free}(G) = \mathbf{Free}(E) = \mathbf{Free}(F)$, $g(E') = e(E')$ for all direct descendents of E and $g(F') = f(F')$ for all direct descendents F' of F. Clearly, g belongs to $\mathcal{I}^1 \bowtie \mathcal{I}^2$ and $K(g) = K(e) \bowtie K(f)$. Since ℓ in $K(e) \bowtie K(f) = K(g) \subseteq K(\mathcal{I}^1) \bowtie K(\mathcal{I}^2)$, we obtain ℓ in $K(\mathcal{I}^1 \bowtie_{gr} \mathcal{I}^2, E \bowtie_{gr} F)$. ■