ALGORITHMS FOR THE SYNTHESIS OF IMPLEMENTATION STRUCTURES

Lawrence A. Rowe*
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 833-5233

Fred M. Tonge
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 833-6357

Technical Report #91

- - - - - - - - - - -

*Current Address:
Computer Science Division, EECS Department
University of California, Berkeley
Berkeley, CA 94720
(415) 642-5117

Abstract

This paper presents an approach to translating the data associated with a problem-solving procedure into efficient implementations. The approach involves reduction of problem-domain data structures to implementations by way of intermediate modelling structures. Formalisms are introduced for describing modelling structures -- abstract representations of data characteristics and relationships, independent of any specific implementation -- and implementation structures -- machine-processable representations. Based upon these formalisms, algorithms are presented for recognizing known modelling structures, for synthesizing implementations for modelling structures not recognized, and for combining several implementations according to structure membership and variable binding relationships. Design considerations influencing these formalisms and algorithms are discussed.

Keywords: data structure, data structure formalisms, modelling structure recognition, implementation structure choice, implementation structure synthesis, data abstraction.

1. Introduction

There has been continued interest in computer science in different methods for representing the data associated with and acted upon by computer programs. This interest has ranged from consideration of appropriate data structures for inclusion in higher-level languages designed to detailed evaluation of specific implementations in storage of simple data structures. Most recently, the emphasis in research on data structures has been concerned with larger groupings (often called abstract data types) which treat a data structure and certain operations upon it as a unit, with formal treatment of the properties of such abstract units, and with the selection among alternative implementations of such data types [PR76].

In this paper we present an approach to automating the translation of data representations in a program into efficient implementations.

While this approach shares many of the same problems and concerns as current work in data structures, it differs in underlying motivation and so in terms of the way many intermediate problems are treated. We first present the view of the problem-solving/programming process that motivates this approach, followed by discussion of specific emphases (design decisions) which have shaped the research. Next we summarize the formalisms developed for stating modelling structures and implementations structures. Following that we present three major algorithms for the generation of alternative implementation structures which are central to this approach.

In the final two sections we discuss the relationship of this work to other current research in data abstractions and summarize and conclude this paper.

2. A View of Data Representation

Programming problems arise in a diverse collection of problem domains, and in those problem domains the original problem data is expressed in a form dictated by the needs of the problem itself and of some general approach to a problem-solution procedure. We call this original data structure a problem structure. Eventually, as the natural result of the programming process, this problem-solution procedure and its associated data are reduced to a machine-computable implementation. We call the data reperesentation derived at this level an implementation structure. Often in the problem-solving/programming process the data representation and associated procedures are expressed in terms of some intermediate structures. Typically, such intermediate structures are abstract representations of the characteristics and relationships represented in the problem domain, suitable for analysis and expression of algorithms by human problem-solvers (programmers), and largely independent of the restrictions derived from specific implementations. After D'Imperio, we call such structures modelling structures [DI69].

Thus, the programming process includes reduction of problem structures to implementation structures utilizing one or more layers of modelling structures.

Modelling structures are useful because they provide a level of formal representation common to many problem domains and yet independent of the restrictions and commitments of specific implementations. Thus, they provide a locus for collecting expert knowledge and techniques about data representation, an opportunity for expressing problem-solving procedures with a clarity often obscured by the need for efficient implementations, and an opportunity to carry out substantial debugging and correctness-proving at a level closer to the problem logic than to concerns of machine efficiency.

Note that, as we use the term, implementation structures may be expressed at any level of language, if that is the level at which the final program is written. In some cases, implementation structures are expressed in machine or assembly language; in others, in a higher-level language. For example, the abstract notion of an "array" as modelling structure may be implemented as a FORTRAN array, an ALGOL array, a BASIC array, and so forth, each of the latter being a different implementation structure with different properties. Or, the modelling structure "tree" may be implemented in FORTRAN, using FORTRAN arrays, in several distinctly different ways, each a different implementation structure for that modelling structure.

In practice, the neat separation between modelling structures and implementation structures does not exist. Typically, programs are written partly at the modelling level (program control structure and some operations) and partly at the

implementation level (data structures and related operations). Because the representations available for expressing information relationships and data in programming languages actually represent specific implementation structures, the programmer is forced to select implementation structures at the time the problem-solution procedure is modelled.

Forcing the programmer to choose an implementation structure at this stage of the programming process leads to lessened clarity in the program, diminished opportunity to analyze and select among different implementations, and considerable reprogramming as efficiency considerations are gradually recognized. In this research we are interested in providing a representation of information relationships and data at the modelling structure level during the programming process, then using the computer itself to assist the programmer in producing an efficient implementation. A system for generating and selecting among alternative structures would thus be another tool in a programming environment, one of a wide variety of facilities to aid the problem-solver/programmer in his task.

3. Design Emphasis

Here we list seven design emphases which have shaped this research.

First, we consider only the problem of developing implementation structures from already specified modelling structures, leaving aside the less well-specified and certainly more difficult area of dealing with problem structures.

Second, the modelling structure formalism (discussed in Section 4) is intended to be a consistant framework for expressing classes or types of elements and their relationships. It should be capable of expressing modelling structures currently in use (e.g., graphs, trees, lists, and arrays) and also more general statements of relationship. Those users who know which specific modelling structure they want may specify it by name. Those users who are not aware of the full catalog of specific modelling structures, who do not know exactly which structure they want, or who want a structure not provided in the catalog, may define such a structure using a suitable formalism. Thus, we envision a system based on a well-defined, coherent foundation for modelling structures.

Third, the modelling structure formalism should capture that information needed for generating efficient implementations. This is in contrast to other approaches to abstract data types where the emphasis is on conciseness of description and minimizing representational bias.

Fourth, the implementation structure formalism (discussed in Section 5) is intended to be a consistant framework for expressing storage implementations of modelling structures so as (among other things) to be able to compare alternative implementations and to combine implementations of several modelling structures. For this initial report, we restrict our attention and the scope of the formalism to implementations in first level storage (primary storage) as describable in a typical assembly language.

Fifth, the system should allow representation and use of known implementations of known modelling structures (i.e., expert knowledge) where that exists.

Sixth, as a general principle the implementation synthesis and generation system should allow user interaction at all major decision points, permitting the programmer to override system choices or temporarily change the ordering of alternatives. This consideration has not entered strongly into the research so far, since it is more related to how the various pieces of the system fit together than with the pieces themselves, but we have taken each step with that criterion in mind.

Finally, we concentrate initially on only certain parts of a total system, leaving unspecified for present those areas on which there is a great deal of other current research. For example, we do develop an algorithm for synthesis of implementation structures for "unrecognized" modelling structures; we do not work now on the data flow techniques which supply necessary information for efficient choice among implementation structures. As a corollary of this decision, we do not develop a new programming language with modelling structure capabilities, but rather consider a more abstract capability that could be adapted to many languages.

In particular, we concentrate on the problems of developing several alternative implementations for a given modelling structure, of combining implementations for the several modelling structures used in a program, and on the selection of the most desirable set of implementations with respect to some criterion. The

following sections summarize the modelling structure and implementation structure formalism we have adopted and then present three algorithms central to this overall task.

4. Summary of Modelling Structures Formalism

Modelling structures are abstract objects (and associated operators) which provide an intermediate stage in the mapping from problem to implementation. As such, they are of value only if they facilitate that mapping. Modelling structures have proven to be of value in the problem-solving/programming process for two major reasons. First, they break a large, ill-structured problem into two smaller, relatively independent problems, (mapping problem structures to modelling structures and the resulting modelling structures to implementation structures) thus typically reducing the effort in problem-solving. And second, certain useful modelling structures have taken on an existence of their own, carrying over from one programming situation to another independent of the particular initial problem or resulting implementation. Consequently, modelling structures have become a focus for technical knowledge and expertise in their own right.

The goals of our modelling structures formalism are to provide a means of expressing such structures independent of choice of implementation, to capture the information necessary for efficient choice of implementation, to provide a notation in which the useful and commonly accepted modelling structures can be expressed easily and directly, and to provide a convenient notation for expressing abstractly the objects and their

interrelations of problem structures not captured in the common modelling structures.

Questions of the completeness and adequacy of a formalism for the modelling structures domain are less easily answered than for the other domains of interest here. For problem structures and implementation structures, there are existing realities against which to measure the formalism (in the former case, that part of the "real world" in which the problem arises; in the latter, the programming language or machine storage in which the implementation must be expressed). But modelling structures are abstract creations of the mind, with no corresponding concrete reality against which to measure their scope. Thus, we can only judge such a formalism as satisfactory or not in satisfying the above goals, which are by nature open-ended, and at the same time as satisfactory or not in terms of the elegance and closure we ask of any abstract system, in some sense balancing between conflicting demands for open-endedness and closure.

The abstract entities treated in the modelling structures formalism are either primitive (not decomposible) or structured entities (composed of elements which are themselves instances of modelling structures). Primitive entities can be of whatever types are appropriate to the problem at hand and for which appropriate operators are available. They are not discussed further here.

Structured entities are characterized by six properties of their component elements, relations, and the operations upon them.

1. Replication. Elements in the structures may or may not be repeated.

2. Ordering. Elements in the structure may be linearly ordered according to a specified predicate, this "built-in" ordering being preserved by operations on the structure.

3. Distinguished Elements. One or more distinguished elements may be defined by predicates, typically upon the relations of a structure. Distinguished elements refer to that single element in the structure, if any, which satisfies the predicate.

4. Referencing Methods. Elements in the structure may be referenced by some or all of the following access methods, as specified.

   a. Distinguished element, bound to an element based on structural relationships (e.g., top of a stack or root of a tree).
   b. External access, bound (through the structure) to a particular element.
   c. Selection, by either element name or element number. Element numbers may be treated as ordered or simply as a primitive set of element names. Element name selection is similar to that for "structures" in PL/1 or "records" in COBOL.
   d. Quantification, either universal or existential.

5. Relations. Elements in the structure may be related by (possibly several) relations, each characterized by the following information.

   a. Degree (one-one, one-many, many-one, many-many).

   b. Scope of domain, may be total (all elements included), unique (all but a single unique element included), or partial (none or more elements included).

   c. Scope of range (same possible values as domain).

   d. Connected or not connected (that is, transitive closure via the relation and its assumed algebraic inverse).

   e. Reflexive.

   f. Symmetric.

6. Operations. The structure may be operated upon by one or more of the following operations: read an element, replace an element with another, insert an element, delete an element, assign a value to a reference, relate two elements (may imply insertion), unrelate two elements, find the element(s) related to an element, read-attribute value of relation, store-attribute value of a relation, and create-an-access to an element. One or more of the parameters of an operation may be bound to particular values (for example, distinguished elements).

The complete definitions of the primitive operations (see RO76) must take into account the properties of the structure and particularly of the declared relations. While there are some 288 possible combinations of the characteristics of relations given above, only 45 of these are in fact consistant and so realizible, and of these only 36 (16 connected and 20 not connected) can be constructed using the primitive operations defined above. Operations must unambiguously result in a legal structure, or they are treated as an error. The details of these definitions (particularly relate and unrelate) are direct but complicated, and depend on the local context of a sequence of primitive operations to produce an unambiguous result. The resulting (one relation) structures fall naturally into three groups, list-like, tree-like, and graph-like, depending on the degree of the relation.

Examples of several commonly-used modelling structure definitions using this formalism are given in Figure 4.1. These are to some extent our own personal arbitrary definitions, in that there are no agreed upon definitions in computer science for such structures as set, stack, tree, or array. Furthermore, there are many ways to capture the same abstract behavior in this formalism. We do not propose these as definitions for universal agreement, but as examples for which the reader can substitute his own preference as desired. The "*" in operation definitions means that any element or structure reference as appropriate may be substituted. The use of a distinguished element is illustrated, for example, in the definition of a stack, where element referencing is restricted to the distinguished

```
Stack:
        replication;
        no ordering;
        distinguished element:  (element such that
                                 NEXT(HEAD) is undefined);
        referencing:  distinguished element;
        relations:  NEXT (1-1, unique domain, unique
                          range, connected, not
                          symmetric, not reflexive);
        operations:  read(HEAD),
                     delete(HEAD,*),
                     relate(*,[<HEAD,NEXT,*>]);


Array:
        replication;
        no ordering;
        distinguished element:  none;
        referencing:  element number selection
                      (n-dimensional);
        relations:  none;
        operations: read(*),
                    assign(*,*);


Binary Tree:
        replication;
        no ordering;
        distinguished elements:  (element such that
                                  ANSC(ROOT) is
                                  undefined);
        referencing:  distinguished element,
                      external access;
        relations:  LEFT(1-1, partial domain, partial
                         range, not connected, not
                         symmetric, not reflexive),
                    RIGHT(1-1, partial domain, partial
                          range, not connected, not
                          symmetric, not reflexive),
                    ANSC(many-1, unique domain, partial
                         range, connected, not symmetric,
                         not reflexive),
                    ANSC is inverse of LEFT union
                     inverse of RIGHT;
        operations:  read(*),
                     delete(*,*),
                     replace(*,*,*),
                     create-an-access(*,*),
                     relate(*,[<*,*,*>]),
                     related(*,*,*);
```

Figure 4.1: Examples of Modelling Structure Definitions

element HEAD, and operation parameters are bound to that distinguished element.

The modelling structures formalism outlined here can be used in several ways to provide a test-bed for studying this approach to the generation of data structure implementations. It could be implemented directly as the data manipulation portion of a new or existing programming language. It could be embodied, probably with minor modifications, in an extensible language with data definition features, or it could be embodied in an existing programming language through addition of a set of procedures and restrictions on the use of existing facilities. As discussed in the earlier section on design emphases, we have chosen to bypass for now a full-scale implementation in favor of focussing on development of some of the algorithms necessary for generation of implementations. Thus, we have implemented only those features necessary for providing inputs to such algorithms.

5. Summary of Implementation Structures Formalism

The implementation structures formalism presented here is a notation for describing data structures at the "machine" level -- that is, at the level of main storage as addressed in machine language, assembly language, and possibly by the storage allocation mechanisms of the operating system.

An implementation structure is composed of a storage structure and an interpretation of that storage structure. Storage structures are composed of cells (holding primitive values) and/or

groupings of cells and storage structures. Groupings may be on the basis of contiguity (indexing and other forms of address arithmetic), explicit linkage (pointers), or structure-defining functions (association and hashing). The interpretation of an implementation structure reflects a particular implementation strategy, and is made up of correspondences between components of the storage structure and of the modelling structure which it implements and of procedures for using the storage structure.

A particular storage structure (grouping of cells in storage) may have several possible interpretations. An interpretation contains all of the data needed to use the storage structure--that is, all of the data needed by a translator to translate the modelling structure and the operations that manipulate it. Note that this interpretation is expressed in a mixture of both procedural and non-procedural data.

Storage structures are defined using the following syntactic constructs.

1. Composition
   - + Contiguous composition.
   - @ Linked composition.
   - ? Composition by structure-defining function.

2. Repetition
   - n Fixed number of repetitions (n an integer).
   - # Indeterminate number of repetitions.

3. Syntactic grouping.

   () Subgroups.

   : Naming.

   ! Alternation.

   ; Unspecified interconnection.

   / Distribution of operator.

4. Unary composition (space reservation)

   @ Linkage pointer.

   ? Structure-defining function input.

5. Primitive elements

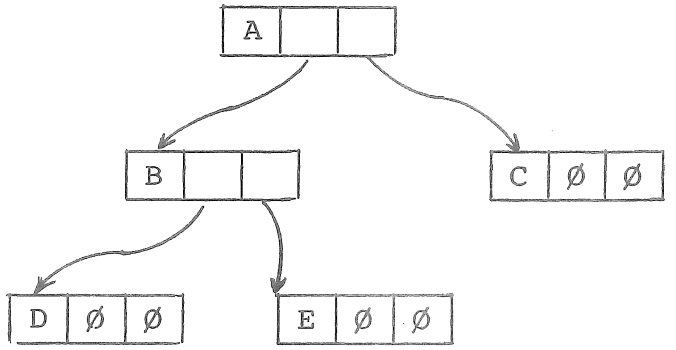
   cell.

   tag (Boolean value).

Several instances of this notation are given in Figure 5.1, together with pictorial examples (for a binary tree with data values A,B,C,D,E) of each. The symbol Ø is used to indicate a null pointer.

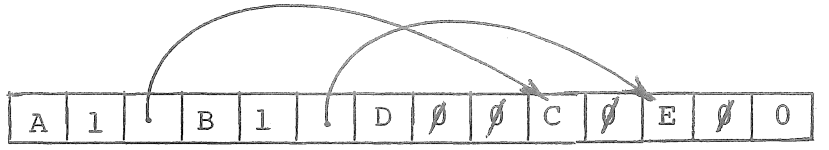
b:#(g)

g:cell+@g+@g



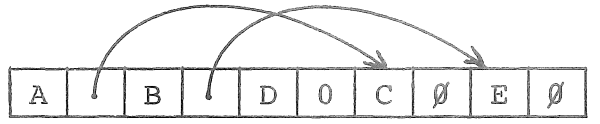
b:#(g)/+

g:cell+tag+@g



(b)

b:#(g)/+
g:cell+@g



(c)

Figure 5.1: Example Storage Structures for a Binary Tree

The interpretation of a storage structure specifies both the correspondences between components of the modelling structure and of the storage structure and also other information necessary for complimenting that storage structure. This information is organized in several groupings. (Only an example of the information included is given below.)

1. Relations. For each relation, its modelling structure name, procedure for accessing the related element(s) given an element, and several boolean flags (e.g., indicating whether the relation is explicitly stated or implied, whether the structure is ordered on this relation, etc.)

2. Structure-defining functions. For each function, procedure for finding the associated storage structure.

3. Descriptor. A set of symbols indicating what components of a run-time descriptor, if any, are to be generated. If the set is empty, a descriptor is not generated. Examples of descriptor symbols are: type -- generate run-time type information, insert flag -- a boolean variable indicating whether an element has been inserted since the structure was last ordered (for ordering on access), ext acc, elem no, and dist elem -- symbols indicating that run-time information for handling these forms of referencing should be generated, explicit size --

explicit count of elements in structure should be maintained, and struct size -- size of contiguous block in which structure is stored (applicable only if structure is stored sequentially).

4. Order. Whether ordered, and if so, on insert (order) or on access (sort).

5. Size. For each indeterminate size operator, a bounding value.

6. Elements. Indication of actual values stored (value) or pointers to values (pointer); encodings for selection methods (i.e., for structures referenced by name selection, a value or pointer indicator is maintained for each element).

7. Operations. Code generators for necessary operations, both those specified in modelling structure and other primitive actions associated with implementation (e.g., space allocation).

As was noted above, much of this information is stored procedurally (for example, as access procedures and code generators). However, it is possible to illustrate the information so expressed in a non-procedural manner. The correspondence information for example (b) in Figure 5.1 is given in Figure 5.2. We have introduced subscripts for the composition operators so that they are uniquely associated with relations. Note that the relation LEFT is bounded by $+_2\emptyset$ and RIGHT is bounded by a null pointer convention.

Storage structure:

$b : \#(g)/+_1$

$g : e +_2 \underline{tag} +_3 @\, g$

Interpretation:

the binary tree	corresponds to	b
an element	corresponds to	e
LEFT	corresponds to	$+_1 \wedge +_2\ 1$
RIGHT	corresponds to	$+_3 @$
x ANSC y	corresponds to	$\exists x((x +_1 y \wedge x +_2 1) \vee (x +_3 @ y))$
ROOT	corresponds to	$\exists x \forall y \neg((x +_1 y \wedge x +_2 1) \vee (x +_3 @ y))$

Figure 5.2: Example of Correspondence Information

6. Information Derived from a Program

This section describes the information derived from a program and used in the algorithms for modelling structure recognition and implementation synthesis presented in later sections.

The focus of the information gathering process is deducing information about distinct modelling structures. During the execution of a program many entities are created.

For example, in the program segment

```
while true do
   begin
    . . .
    x ← "create entity";
    . . .
  end
```

each iteration of the loop creates an entity. The question is which of those entities should be treated separately for implementation selection purposes (i.e., which ones should be given separate entity names)? One approach is to generate a unique name for each entity created. This would then require simulating the actual program execution to gather the necessary information, an unacceptable alternative. Another approach, the one used here, is to assume that all entities of the same type created at the same lexical point in a program are implemented in the same way. Thus, the set of entities about which information is gathered is the set of points in a program at which entities are created.

Three kinds of information are deduced from a program:

1. execution time membership relations,

2. execution time variable binding relations, and

3. properties of the modelling structure entities created during execution.

This information is collected by pseudo execution of the program.* Assuming each distinct modelling structure used in a program is given a name e_i, for i=1,2, . . .,n, the execution time membership information is represented by a relation Elem, where e_i is Elem-related to e_j if e_j may be a member of e_i (e_i must be a structured entity) during execution. In a similar fashion, the variable binding information is represented by a relation Var, where X (some variable used in the program) is Var-related to e_i if e_i may be bound to X during execution. Two additional pieces of information concerning element membership and variable binding are needed. First, can more than one instance be created at a particular place in a program? This is represented by a predicate Mcreatep on the set of entity names describing for each e_i whether multiple creations are possible. Second,

- - - - - - - - -

* Sintzoff was the first to describe this general method of analysis [SI72]. Schwartz gives a clear description of the process involved [SC75, see the paragraph beginning at the bottom of page 723]. While the membership and variable binding analysis used here is not significantly different from the analyses that Schwartz's system performs, analyzing a program to deduce modelling structure properties for recognizing known structures is new.

can an entity be inserted into a structure more than once or can more than one creation of an entity be inserted? This is represented by a predicate Minsertp on the relation Elem (treated as a set of ordered pairs). Such information can be depicted graphically, as shown in Figure 6.1. In the figure, entity e_2 may be an element of e_1 (depicted by the solid line) variables S1 and X may be bound to entities created by e_2 (depicted by the dashed lines), multiple instances of e_2 and e_3 are created (depicted by the asterisks before the entity names), and either more than one instance created at e_2 or a specific instance more than once may be inserted into e_1 (depicted by the asterisk before the solid line between e_1 and e_2).

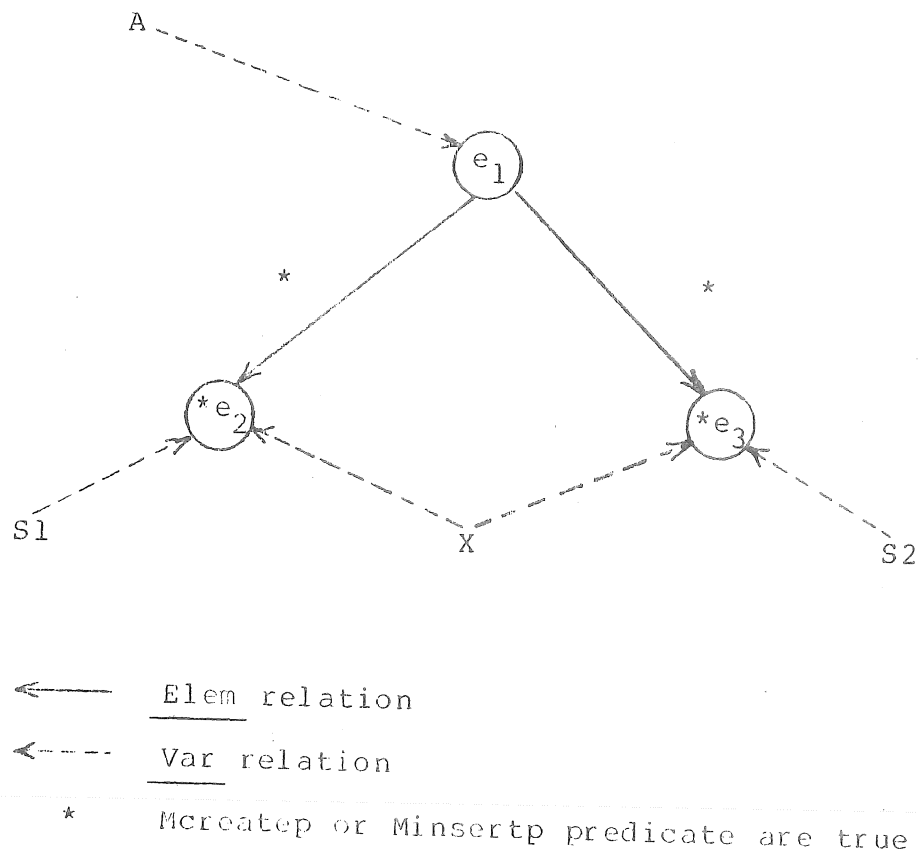


Figure 6.1: Element Membership and Variable Bound Information

The third kind of information gathered concerns properties of the modelling structures. These properties correspond to the ones defined in the modelling structures formalism (e.g., replication, ordering, referencing, and operations). For modelling structures defined by name, these properties are retrieved from a catalog of definitions and are used to check consistency of structure use (particularly with respect to the operations and referencing forms). For structures not defined by name, the pseudo execution process collects information about the static properties of the structure from the type definition (i.e., replication, ordering distinguished elements, and relations) and dynamic properties from the other part of the program (i.e., referencing mechanisms and operations).

Two features of the pseudo execution embody important aspects of the recognition process. First, for many structures, constraints on arguments to the operations or limitations on which elements in a structure can be referenced are essential identifying characteristics. A stack illustrates this point very nicely. Note in the stack description given in Figure 4.1 that arguments to each of the operations are constrained. This requires that the information gathering process retain the actual argument in some cases while substituting an indicator that any argument is allowed (or used) in other cases. This is accomplished by substituting the "any argument" indicator in all cases except where a distinguished element or a relation name is used, in which case the identifier is substituted. For example, if the operation is 'DELETE TOP FROM X, and TOP is not a distinguished

element, the operation delete (*,*) would be included as an operation performed on the entities bound to X. By contrast, if TOP is a distinguished element for X, delete (TOP,*) would be included. Here the definition of the modelling structures formalism and the mechanical recognition process are closely matched.

A second feature of interest in the pseudo execution process concerns looping constructs and procedure invocations. The execution of all looping constructs is simulated twice, once for the initial pass and once for the loop. In this way, variable bindings and element membership relations can be adjusted for both entries to the loop body. This is analogous to Howden's notion of the boundary and interior conditions of a loop [HO75]. Procedure calls are handled by simulating execution of the procedure body at each call after the appropriate argument bindings have been made. (In a language with lexical scoping, such as Algol 60, the procedure body could be executed once at the point of definition and the environment updated at the procedure call rather than simulating execution of the procedure at each call.)

Examples of the information gathering process are presented in the next section. More details on the process are available in RO76.

7. Mechanical Recognition Algorithm

This section describes the organization of a catalog of known modelling structures and the algorithm for matching structure

descriptions derived from a program with entries in the catalog. Following this, two examples of mechanical recognition are presented.

7.1 Organization of a Catalog of Known Modelling Structures

Three organizations of the catalog were considered: an unrelated collection of entries, a discrimination net of entries (descendant entries related to parent by a choice for some given modelling structure property and only leaf entries being complete structure descriptions), or an inclusion lattice (descendant entries related to parent by a more restricted set of modelling structure properties and all entries being complete structure descriptions). The inclusion lattice organization was chosen and organized so that the matching algorithm will match any structure description deduced from a program. (As a result, some inclusion relationships between individual attributes may seem arbitrary.)

An entry in the catalog is composed of a structure name, a match six-tuple, and structure-specific alternative implementation knowledge. The six fields of the match name in order are: replication, ordering, relations (mapping degree, domain scope, range scope, connectivity, reflexivity, and symmetry), distinguished elements, referencing, and operations.

The root of the catalog has the most general match name,

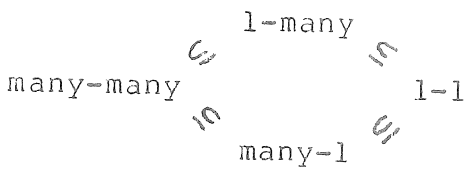
$$\langle *,*,\{*(*,*,*,*,*,*)\},\{*(*)\},\{*\},\{*\}\rangle .$$

"*" matches any property value. The containment relations for the various properties are:

no ordering $\subseteq$ ordering

no replication $\subseteq$ replication

no relations $\subseteq$ any relations



partial domain $\subseteq$ unique domain $\subseteq$ total domain

partial range $\subseteq$ unique range $\subseteq$ total range

not connected $\subseteq$ connected

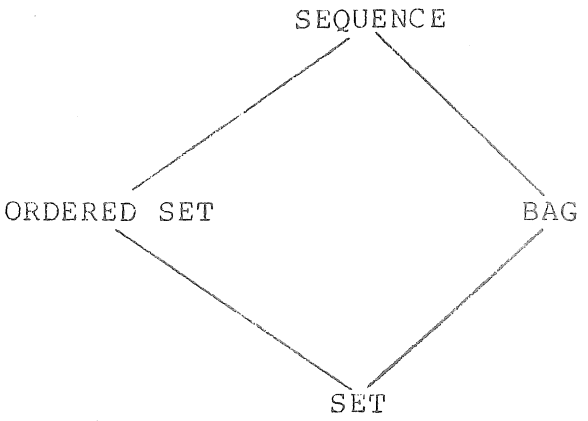
not reflexive $\subseteq$ reflexive

not symmetric $\subseteq$ symmetric

Figure 7.1 shows a small part of the catalog with match names. A bag (a structure similar to a set, except elements may be replicated [WA73], called a multiset by Knuth [KN69]) is contained in a sequence because a bag does not allow ordering and has a more restricted set of references. ("elem no(n)" means element number referencing with a n dimensional index.) A catalog of modelling structures is shown in Figure 7.2. (A list of some of these structures along with their match names is given in Figure 7.4. A complete list is available in Appendix III of reference RO76.)

The catalog shown in Figure 7.2 is not the complete catalog that would be used in an implementation structure synthesis and selection system. The catalog would be augmented with representations of other modelling structures and with other representations of the structures shown. There could also

be nodes reflecting other meaningful combinations of modelling structure properties even though they may not correspond to a meaningful structure.



where:

Sequence =
<rep,ord,ϕ,ϕ,{elem_no(1),exist_quant,univ_quant},
{read(*),insert(*,*),delete(*,*),replace(*,*,*)}>

Ordered Set =
<norep,ord,ϕ,ϕ,{elem_no(1),exist_quant,univ_quant},
{read(*),insert(*,*),delete(*,*),replace(*,*,*)}>

Bag =
<rep,noord,ϕ,ϕ,{exist_quant,univ_quant},
{read(*),insert(*,*),delete(*,*),replace(*,*,*)}>

Set =
<norep,noord,ϕ,ϕ,{exist_quant,univ_quant},
{read(*),insert(*,*),delete(*,*),replace(*,*,*)}>

Figure 7.1: Part of Catalog

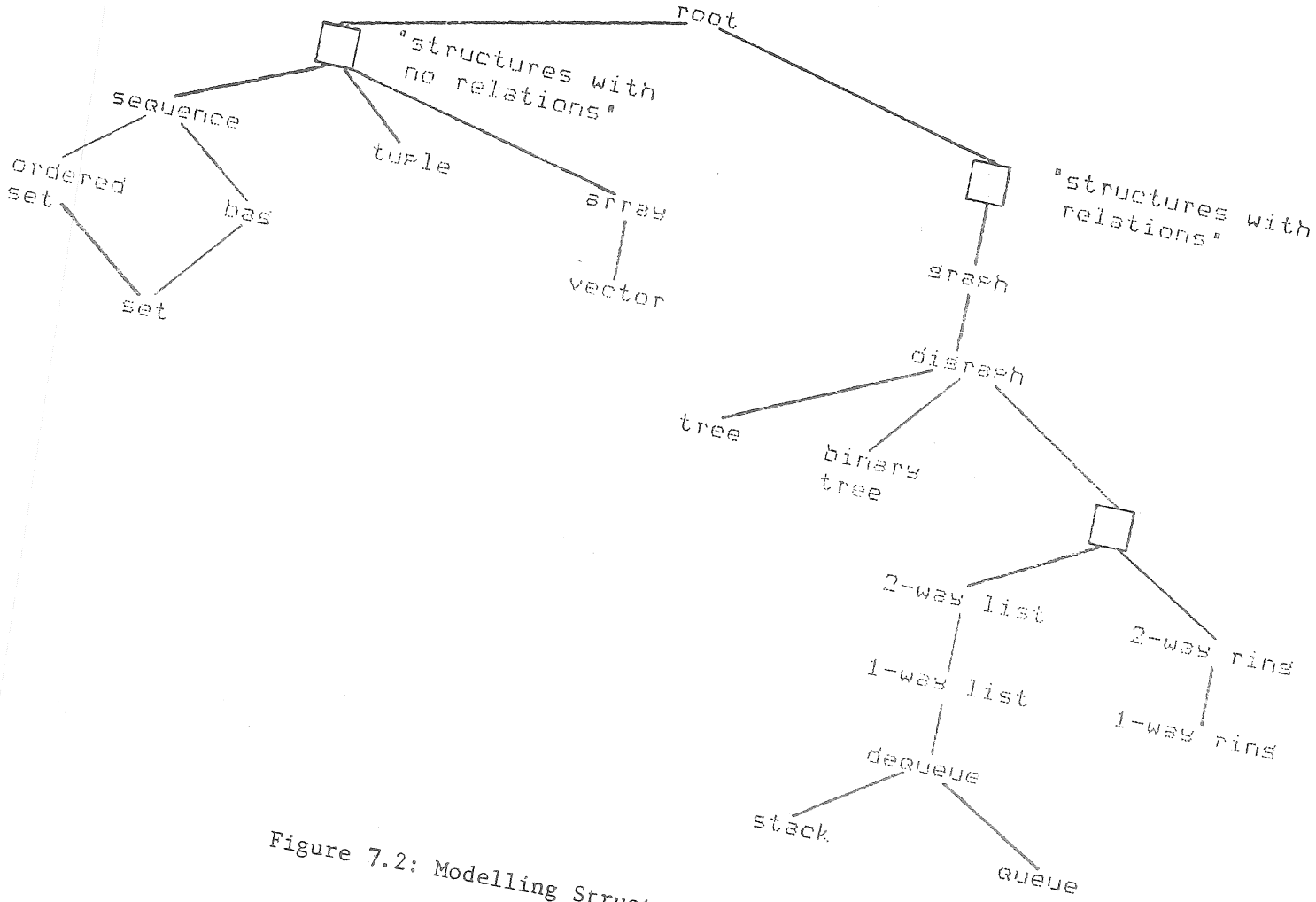


Figure 7.2: Modelling Structures Catalog

7.2 Catalog Matching Algorithm

Given a catalog as described above and the modelling structure information deduced from a program, the matching algorithm is simple. For each deduced structure, starting at the root of the lattice, compare the deduced description with the match name for each descendant of the current entry. If a descendant matches, make it the current entry and repeat; otherwise, the current entry is the structure recognized. Comparing deduced descriptions with match names uses the containment relations described above. For example, the deduced description

<rep,noord,ϕ,ϕ,{name select},{read(*),delete(*,*)}>

matches the name

<rep,ord,ϕ,ϕ,{name select,exist quant,univ quant},
{read(*),insert(*,*),delete(*,*),replace(*,*,*)}>

because no ordering is contained in ordering and the deduced description reference set and operation set are subsets of the respective sets in the match name.

When matching with descendent entries, more than one descendent may match. In that case, continue the match algorithm along all paths simultaneously. It is perfectly acceptable that the algorithm may match more than one catalog entry since this means more than one set of predefined alternatives (assuming the matched nodes have alternative implementations associated with them) are available to implement the modelling structure.

7.3 Examples of Mechanical Recognition

Two examples of mechanical recognition are described in this section. The first one is a contrived example to demonstrate the recognition of a stack while the second is an actual programming problem.

7.3.1 First Example. The program for the first example, in which the structure X is to be recognized, is shown in Figure 7.3. The modelling structure description deduced is

<rep,noord,{SUC(1-1,udom,uran,conn,noref,nosym)},
{TOP(undef(SUC(TOP)))},{dist elem},
{read(TOP),delete(TOP,*),relate(*,[<TOP,SUC,*>])}>.

This description matches the root, graph, digraph, 2-way list, 1-way list, dequeue, and finally stack entries in the catalog. The match names for these structures are shown in Figure 7.4. Thus, the structure is recognized as a stack.

In the matching process, the names for the relations and distinguished elements used in the catalog entries are formal arguments to which the actual names in the derived descriptions are bound during the matching process.

```
BEGIN
  DEFINE S.TYPE:(ELEMENTS:INT,
                 RELATION:SUC(1-1,UNIQUE DOMAIN,
                              UNIQUE RANGE),
                 DIST ELEM:TOP(NOT(DEFINED(SUC(TOP)))));
  DECLARE X:S.TYPE,Y:INT,I:INT
  BEGIN
        .
        .
    FOR I←1 TO 1Ø DO
        RELATE [SUC(TOP)=Y] IN X;
        .
        .
    DELETE TOP FROM X;
        .
        .
    IF TOP=4 THEN . . .;
        .
        .
  END
END
```

Figure 7.3: First Example Program Segment

```
Root =
  <*,*,{*(*,*,*,*,*,*)},{*(*)},{*},{*}>

Graph =
  <rep,noord,{*(*,*,*,*,*,sym)},{*(*)},{dist elem,ext acc},
   {read(*),insert(*,*),delete(*,*),replace(*,*,*),
   createaccess(*,*),relate(*,[<*,*,*>]),
   unrelate(*,[<*,*,*>]),related(*,*),readattr(*,*,*),
   storeattr(*,*,*),assign(*,*)}>

Digraph =
  <rep,noord,{*(*,*,*,*,*,nosym)},{*(*)},{dist elem,
   ext acc},{read(*),insert(*,*),delete(*,*),
   replace(*,*,*),createaccess(*,*),relate(*,[<*,*,*>]),
   unrelate(*,[<*,*,*>]),related(*,*),readattr(*,*,*),
   storeattr(*,*,*),assign(*,*)}>

2-Way List =
  <rep,noord,{"SUC"(1-1,udom,uran,conn,noref,nosym),
              "PRED"(inv("SUC"))},
   {"HEAD"(undef((inv("SUC"))("HEAD"))),
   "TAIL"(undef("SUC"("TAIL")))},{dist elem,ext acc},
   {read(*),delete(*,*),replace(*,*,*),createaccess(*,*),
   relate(*,[<*,*,*>]),related(*,*)}>

1-Way List =
  <rep,noord,{"SUC"(1-1,udom,uran,conn,noref,nosym)},
   {"HEAD"(undef((inv("SUC"))("HEAD"))),
   "TAIL"(undef("SUC"("TAIL")))},{dist elem,ext acc},
   {read(*),delete(*,*),replace(*,*,*),createaccess(*,*),
   relate(*,[<*,*,*>]),related(*,*)}>

Dequeue =
  <rep,noord,{"SUC"(1-1,udom,uran,conn,noref,nosym)},
   {"END1"(undef("SUC"("END1"))),
   "END2"(undef((inv("SUC"))("END2")))},{dist elem},
   {read("END1"),read("END2"),delete("END1",*),
   delete("END2",*),relate(*,[<"END1","SUC",*>]),
   relate(*,[<*,"SUC","END2">])}>

Stack =
  <rep,noord,{"SUC"(1-1,udom,uran,conn,noref,nosym)},
   {"HEAD"(undef("SUC"("HEAD")))},{dist elem},
   {read("HEAD"),delete("HEAD",*),
   relate(*,[<"HEAD","SUC",*>])}>
```

Figure 7.4: Catalog Entries Matched in the First Example

7.3.2 Second Example. The second example is a program that accepts a sequence of related objects (in this case strings) and lists them in topological order. The program is shown in Figure 7.5. The derived description for the only modelling structure (A) is

<norep,noord,{SUC(many-many,pdom,pran,noconn,noref,nosym)},
ϕ,{exist quant},{relate(*,[<*,SUC,*>]),related(*,SUC,*),
delete(*,*)>.

This description matches the catalog entry "structures with relations" shown in Figure 7.2, but does not match the graph entry because of the existential quantification referencing. Figure 6.1 shows the element membership and variable bound information deduced from the program.

```
                TOPOLOGICAL SORT ALGORITHM %

      % DEFINE MODELLING STRUCTURES AND VARIABLES %

DECLARE A:(ELEMENTS:STRING,NOREPLICATION,
           RELATIONS:SUC(MANY-MANY,PARTIAL DOMAIN,
                         PARTIAL RANGE,NOT CONNECTED,
                         NOT REFLEXIVE,NOT SYMMETRIC));
VARIABLE X:STRING,S1:STRING,S2:STRING;

      % MAIN ROUTINE %

BEGIN  % TOPOLOGICAL SORT %

        % READ RELATED PAIRS AND CREATE STRUCTURE
          RELATING THEM %

  WHILE NOT(EOF()) DO
    BEGIN
      READ(S1,S2); % S1 IS RELATED TO S2 %
      IF S1≠S2 THEN
        RELATE [SUC(S1)=S2] IN A
    END;

     % REMOVE MINIMAL ELEMENTS FROM A UNTIL A IS
       EMPTY OR A CYCLE EXISTS %

  WHILE SIZE(A)≠Ø DO
    BEGIN
      EXISTS X IN A SUCHTHAT SIZE(RELATED(X,SUC))=Ø;
      IF DEFINED(X)
        THEN BEGIN  % FOUND MINIMAL, PROCESS IT, REMOVE IT,
                      AND LOOP %
               PROCESS(X);
               DELETE X FROM A
             END
        ELSE BEGIN  % CYCLE EXISTS IN DATA %
               PRINT("LINEARIZATION DOES NOT EXIST BECAUSE
                      CYCLE EXISTS IN DATA.");
               STOP
             END
    END

END  % TOPOLOGICAL SORT %
```

Figure 7.5: Second Example Program

7.4 Experience with Mechanical Recognition

Given the goal of mechanically recognizing known structures, we identify four necessary characteristics of a data definition facility:

1. collections of elements treated as structures should be described explicitly,

2. references to elements in a given structure and constrained references (e.g., top of a stack or root of a tree) should be recognizable,

3. changes to a structure or its elements must be restricted (i.e., implicit changes must be controlled), and

4. relations between elements in a structure and properties of these relations should be described explictly.

These requirements clearly influence several aspects of the modelling structures formalism. While we have attempted to retain the ability to describe a wide range of structures, the need to define a formalism within which known structures can be recognized forces some representational bias. One type of representational bias introduced by the specification technique used here, is that some relations among operations defined for a modelling structure are encoded in the "structure" as opposed to being explicitly stated as in an algebraic relation specification. For example, the fact that the last element inserted is the one retrieved in accessing a stack is represented

partially by an explicitly defined relation between the elements as opposed to being a property of the operations. Consequently, using our formalism, the relation is represented explicitly in any implementation for a stack. An implementation for a stack described by algebraic relations need not represent the relation explicitly (although in many cases it will). This loss of generality must be balanced with the advantages derived from automating the implementation selection process.

A serious deficiency of this recognition process is its inability to recognize recursively defined structures. Recognizing such structures is extremely difficult because in a recursive formulation salient characteristics of the structures are typically represented procedurally rather than in declared static properties. Considerably more sophisitication than that embodied in the process used here would be needed to recognize these representations.

The program analysis methods used here can accommodate strong or weak variable typing, lexical or dynamic scoping of variables, and unconstrained forms of branching.

8. Generation of Alternative Implementation Structures

Using the information deduced from a program by the techniques discussed in section 7 (modelling structure descriptions, matching catalog entries, and execution time membership and variable binding relations), the algorithms presented in this section generate alternative implementations for each modelling structure.

The generation process produces a set of alternative implementations for each separable modelling structure. A separable modelling structure is an equivalence class of the relation created by extending the membership relation to an equivalence relation. An example will make this clear. Figure 8.1 shows 8 distinct modelling structures and a membership relation. There are 3 separable structures in this example (denoted by the boxes).

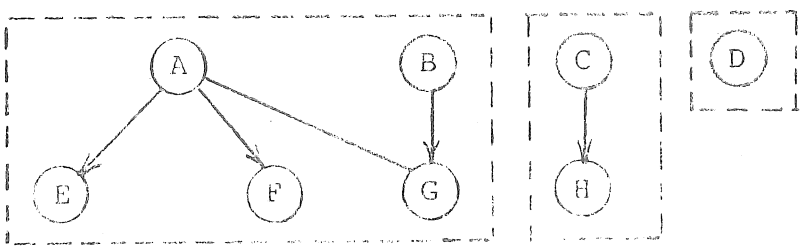


Figure 8.1: An Example of Separable Modelling Structures

There are three phases to the generation process. In the first phase alternative implementations are generated for each distinct modelling structure. For those structures which match a known modelling structure, alternative implementations are retrieved from the catalog (i.e., expert knowledge about alternative implementations). For those structures which do not match a known modelling structure, alternative implementations that can represent the required abstract behaviors are synthesized. The second phase of the process produces alternative implementations for each separable modelling structure by combining together

implementations for structures in a class according to the membership and variable binding information. The third phase adds coalescings of modelling structures to the set of alternatives.

The remainder of this section describes the synthesis and the combine algorithms, presents an example, and discusses our experiences with the algorithms. We have not worked directly on the phase which adds coalescings. Nevertheless, coalescings would likely be represented by procedural experts which recognize whether a particular coalescing applies, and, if applicable, synthesizes implementations incorporating the coalescing.

8.1 Synthesis Algorithm

Implementations are synthesized by focusing on the relations defined for a modelling structure (for a structure without relations, one is added). For each relation a set of alternative implementations for a structure with just that relation is retrieved from the implementation library. The implementation library is a database including for each realizable relation alternative implementations for a modelling structure with just that relation. An implementation is synthesized by joining together implementations for each relation.* It may not be

- - - - - - - - - -

* All implementation descriptions, whether in the modelling structures catalog, the implementation library, or synthesized by the generation process, are represented using the formalism described in Section 6 above.

possible to join together any arbitrary pair of implementations.** For example, an implementation for a structure with two relations cannot use cell contiguity to represent both relations simultaneously. After a set of implementations are synthesized, other modelling structure properties (e.g., ordering, replication, and referencing) are used to change, add, and remove possible implementations.

Before presenting the algorithm, the concept of enumeration must be defined. An enumeration relation is one which traverses, or enumerates, all elements in a structure. Many implementations for modelling structures require that there be an enumeration relation. For example, enumeration is often required in implementations for modelling structures with universal quantification referencing (to generate all elements) or when an ordering predicate is specified (to represent the ordering between elements). Some implementations do not require an enumeration relation. The synthesis algorithm decides whether an enumeration relation is necessary based on the modelling structure properties. This is

- - - - - - - - - -

** Unless one relation is derivable from another. This transformation is not attempted because we believe this to be a change in modelling structure as opposed to implementation structure. The philosophy follwed here is that if the programmer explicitly represents the relation then it will be represented in the implementation. This is an example of how representational bias is included and acted upon in this approach.

one reason why the general process cannot always generate the most efficient implementation.

For those structures requiring enumeration there may be an explicitly defined relation that can be used as the enumeration relation. In other cases an additional relation (1-1, unique domain and range, connected, not reflexive, and not symmetric), called an assumed relation, is added to the modelling structure (e.g., when no relations are defined or the structure is ordered).

Alternative implementations are synthesized for each structure by invoking the algorithm shown in Figure 8.2. The essential step in the algorithm is the call on the SYNTHESIS procedure which retrieves single relation implementations from the implementation library and joins them together. The retrieval operation is straightforward. For each relation in the structure, a set of alternative implementations is retrieved based on the relation properties (i.e., mapping degree, domain and range scope, connectivity, reflexivity, and symmetry.) Each alternative is also checked for consistency with other requirements of the modelling structure (e.g., operations performed on the modelling structure must be defined for the implementation). These single relation implementations are then joined together by repeated application of the operator JOIN(X,Y), where X and Y are implementations, to produce an implementation for the desired modelling structure. The remainder of this subsection describes how JOIN combines two implementation structure descriptions.

Synthesize Alternatives Algorithm. This algorithm synthesizes implementations for a modelling structure (MS). Relations(MS), references(MS), and operations(MS) are the sets of relations, references, and operations, respectively, defined for or performed on the modelling structure. For an implementation x, order(x) and descriptor(x) are the order field and descriptor set of the implementation description. ALT_IMP_STRUX is the set of alternaive implementations synthesized.

1.0 [Initialize.] Let ALT_IMP_STRUX ← ϕ and ASSUMED_REL ← <"",1-1,udom,uran,conn,noref,nosym>.

2.0 [Add assumed relation.] If an enumeration relation is required and either MS is ordered or an enumeration relation does not exist, insert ASSUMED_REL into relations(MS).

3.0 [Look up and join alternatives from the implementation library.] ALT_IMP_STRUX ← SYNTHESIS(MS).

4.0 [Add ordered implementations if no replication or existential referencing.] If MS not ordered and either no replication or existential referencing, for each x ∈ ALT_IMP_STRUX do
- 4.1 Let y be a copy of x.
- 4.2 Let order(y) ← order and ORDER_FLAG for the assumed relation in y be true.
- 4.3 Insert y into ALT_IMP_STRUX.

5.0 [If the structure is ordered, add two cases for each implementation -- sorted and implicit.] If MS ordered, for each x ∈ ALT_IMP_STRUX do
- 5.1 Let order(x) ← sort and ORDER_FLAG for the assumed relation in x be true.
- 5.2 Let y be a copy of x.
- 5.3 Let order(y) ← sort and ORDER_FLAG for the assumed relation in y be true.
- 5.4 Insert insert_flag into descriptor(y).
- 5.5 Insert y into ALT_IMP_STRUX.

6.0 [Mark descriptor for distinguished element, external access, or element number referencing.] If dist_elem, ext_acc, or elem_no ∈ references(MS), insert it into descriptor(x) for each x ∈ ALT_IMP_STRUX.

7.0 [Add explicit size.] If size ∈ operations(MS), then for each x ∈ ALT_IMP_STRUX do
- 7.1 Let y be a copy of x.
- 7.2 Insert explicit_size into descriptor(y).
- 7.3 Insert y into ALT_IMP_STRUX.

8.0 [Done.] Algorithm terminates.

Figure 8.2: Synthesize Alternative Implementations Algorithm

First, JOIN combines the storage structures for the implementations. All storage structure decriptions for implementations in the library must be of the form "$s:\#_{\emptyset}(x)/op$," where "$\#_{\emptyset}$" is the number of elements in the structure, x may be "e" (symbolizing the element) or the name of another construct (e.g., "f," where "f:. . ." also appears), and "op" is from the set {+, ?, @, ⌀}. Thus, the top level of the two storage structures to be joined are "$s:\#_{\emptyset}(x)/op_1$" and "$s:\#_{\emptyset}(x)/op_2$." The arguments to JOIN are switched, if necessary, so that op_1 precedes op_2 according to the ordering [+, ?, @, ⌀]. There are only 7 cases of op_1 and op_2 that can be combined: +@, +⌀, ?@, ?⌀, @@, @⌀, and ⌀⌀. In all other cases the two implementations cannot be joined. The rule for combining the two descriptions is:

> Let $s:\#_{\emptyset}(z)/op_1$ and $z:x+y+op_2z$ (do not add $+op_2z$ if $op_2=$⌀). Substitute the strings for x and y removing duplicate occurrences of e and replacing any occurrence of x or y with z. Copy all other constructs from the two storage structures replacing occurrences of x or y with z.

An example will make this clearer. Figure 8.3 shows the joining of two storage structures. Notice the removal of duplicate e's and the replacement of f by $@_2z$ in f. The same symbol, either identifier or subscripted operator, may occur in both storage structures. To resolve the naming conflict, unique identifiers

and subscripts are substituted throughout the implementation structure descriptions before joining.

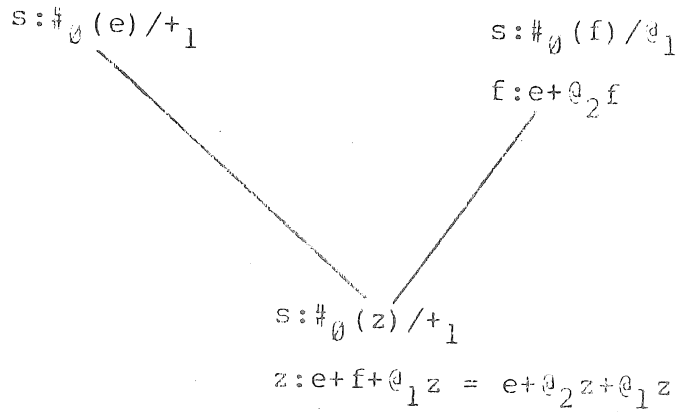


Figure 8.3: Example of Joining Storage Structures

After combining storage structure descriptions, other parts of the implementation descriptions are joined by taking the union of the relations, structure-defining functions, descriptor, and size sets. The order field of the two implementations must be no. The elements field is set to value or the name selection sequence is constructed with each name indicator set to value. Thus far, the join operator is characterized by:

1. Storage Structure: apply combine rule
2. Relations: union
3. Structure-defining Functions: union
4. Descriptor: union
5. Order: no
6. Sizes: union
7. Elements: value or [<$name_1$,value >,...]

The operations set presents a problem. For some operations, such as related, read, and read-attribute, a general code generation procedure can be defined which produces the correct operation based on other parts of the implementation description (particularly the storage structure and relation set). For other operations, such as insert and delete, a general procedure cannot be defined because implementation specific knowledge must be encoded. The problem is that knowledge represented procedurally is difficult to combine (except, for example, if the two sets can be combined one before the other, which is not true in this case). If, on the other hand, this knowledge could be represented as data, the possibilities for defining a suitable joining rule are improved. Our work with examples to date suggests that an acceptable joining rule can be devised. However, until more examples, particularly in the context of an operational system, have been studied it would be premature to state that the problem can be solved.

8.2 Hierarchically Combine Algorithm

There are three situations to be considered when hierarchically combining implementations:

1. does a structure contain more than one type of element,

2. does an entity appear in more than one structure simultaneously, more than once in the same structure, or require an existence outside a structure (called separate existence), and

3. are the elements of a sequentially implemented structure of fixed size.

The first situation requires that the elements of a structure have a run-time descriptor with a type field. The second situation is handled by using an indirect reference (a pointer) to the elements of a structure. In the third situation, for those alternatives with sequential implementations, the size of element implementations is examined and, if not all the same, the elements are changed to pointers. The algorithm shown in Figure 8.4 hierarchically combines implementations for the entities in a separable modelling structure class. Steps 1, 2, and 3 in the algorithm correspond to the three situations just discussed.

Hierarchically Combine Algorithm. This algorithm combines the entities in a separable modelling structure class according to the variable binding and element relations deduced from the program. SM is the set of entities in the class to be combined. For each x $\in$ SM, I(x)={alternative implementations for x}, member(x)={y|<x,y> $\in$ Elem, i.e., entities which may be elements of x}, parent(x)={y|<y,x> $\in$ Elem, i.e., structured entities of which x may be an element}, and variable(x)={y|<y,x> $\in$ Var, i.e., variables which may be bound to x}. For an implementation z, descriptor(z) and element(z) are the descriptor set and element field, respectively.

1.0 [Modify implemetations for those entities which require type information.] For each x $\in$ SM if there is more than one entity in member(x), do
  1.1 for each y $\in$ member(x) insert type into descriptor(z) for all z $\in$ I(y).

2.0 [Modify implementations for those entities which require pointers to elements.] For each x $\in$ SM do
  2.1 If parent(x) has more than one element or variable(x) $\neq \phi$ (i.e., x needs a separate existence), for each y $\in$ parent(x), let elements(z) $\leftarrow$ pointer for all z $\in$ I(y).
  2.2 For each y $\in$ member(x) such that Mcreatep(y)=true, let elements(z) $\leftarrow$ pointer for all z $\in$ I(x).
  2.3 If Mcreatep(x)=true, let elements(z) $\leftarrow$ pointer for all z $\in$ i(x) and for all z $\in$ I(y) where y $\in$ parent(x).

3.0 [Check combinations for variable sized elements of sequential implementations.] For each z in an implementation set for the entities in SM for which the top level operator in the storage structure is + do
  3.1 If the sizes of the implementations for the entities in member(z) (there may be more than one element entity) are not the same, let elements(z) $\leftarrow$ pointer. Otherwise, if the sizes are not the same, but are fixed, insert struct size into the descriptor of the implementations for the entities in member(z).

4.0 [Done.] Algorithm terminates.

Figure 8.4: Hierarchically Combine Algorithm

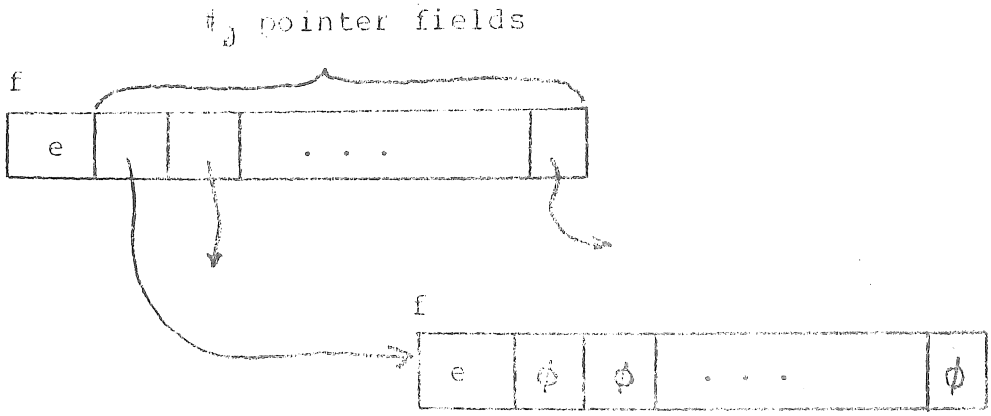
8.3 Example of Implementation Structure Synthesis

The example presented below was processed by the synthesis algorithm implemented in UCI Lisp [BO73]. The example uses the modelling structure derived from the topological sort program in figure 7.5. The structure description input to the program was

```
<norep,noord,{SUC(many-many,pdom,pran,conn,noref,nosym)},
 ϕ,{exist quant},{insert(*,*),delete(*,*),
   relate(*,[<*,SUC,*>]),related(*,SUC,*)}>.
```

Because of the existential quantification referencing, an enumeration relation was added to the structure. Thus, the structure had two relations when the SYNTHESIS procedure was called.

The implementation library included two representations for a 1-1 relation (sequential or linked implementation) and five representations for a many-many relation (three are shown in Figures 8.5-8.7; the other two are like those shown in Figures 8.5-8.6 except that the f's are stored sequentially).

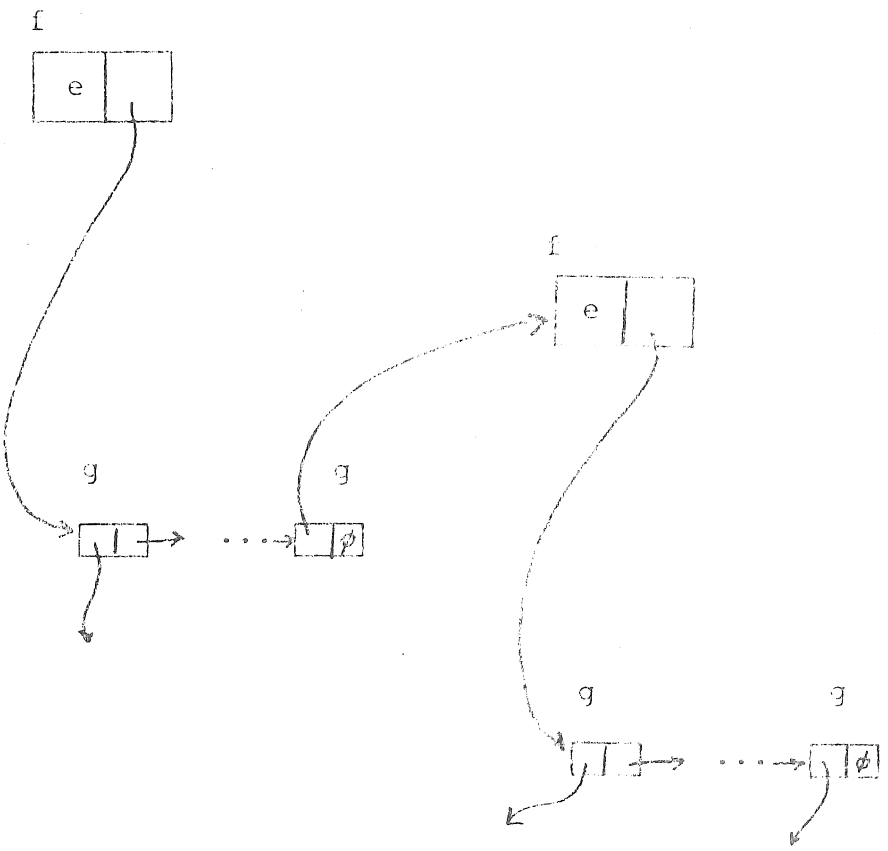


$s:\#_J(f)$

$f:e+_1 g$

$g:\#_J(@f)/+_2$

Figure 8.5: Pointers to Related Elements Stored Sequentially with Each Element

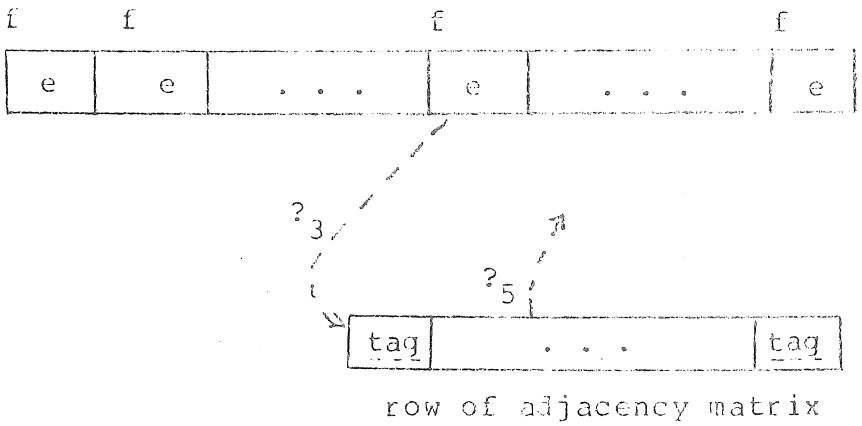


s:#$_{g}$(f)

f:e+$_{1}$$_{@g}$

g:#(@f)/@

Figure 8.6: Explicit Link from Each Element to a Linked List of Pointers to Related Elements



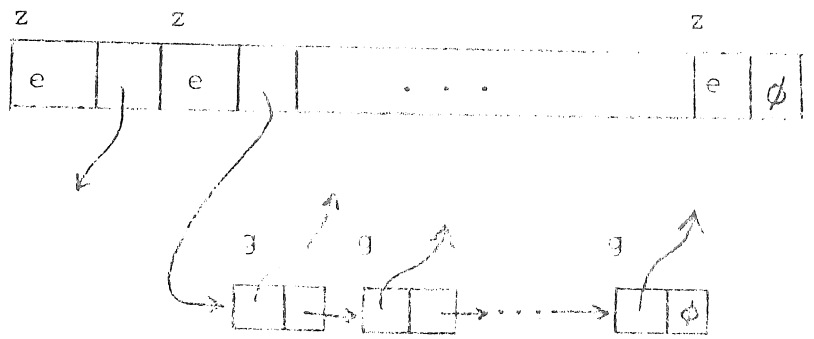
$s: \#_{\emptyset}(f)/+_1$

$f: e +_2 ?_3 g$

$g: \#_{\emptyset}(\underline{tag} +_4 ?_5 f)/+_6$

Figure 8.7: Sequentially Stored Elements with a Structure-defining Function Link to the Row of a Boolean Adjacency Matrix

SYNTHESIS returned 7 implementations for the modelling structure (the other 3 could not be joined). Figure 8.8 shows one of the implementations returned. This implementation represents the enumeration relation by sequentially storing the z's. Associated with each z is a pointer to a linked list of pointers to elements related to this one by the relation SUC. Storage structures for the other implementations returned by SYNTHESIS are shown in Figure 8.9. The implementation depicted at the top of the figure also represents the enumeration relation in sequential storage. The SUC relation is represented by $\#_{\emptyset}$ pointer fields stored with each element. The other implementations are variations of these based on whether the elements are stored contiguously or linked and how the connection relation SUC is stored. The last implementation shown uses an adjacency (or connection) matrix to represent SUC with structure-defining functions to get from the set of elements to the matrix and back.



(case 1) $S:\#_{\emptyset}(z)/+_1$

$z:e+_2@g$

$g:\#(@z)/@$

Figure 8.8: Example Implementation Returned by SYNTHESIS

(Case 1) $s:\#_{\emptyset}(z)/+_1$

$z:e+_2@g$

$g:\#(@z)/@$

(Case 2) $s:\#_{\emptyset}(z)/+_1$

$z:e+_2g$

$g:\#_{\emptyset}(@z)/+_3$

(Case 3) $s:\#_{\emptyset}(z)/@$

$z:e+_1g$

$g:\#_{\emptyset}(@z)/+_2$

(Case 4) $s:\#_{\emptyset}(z)/+_1$

$z:e+_2g+_3@z$

$g:\#_{\emptyset}(@z)/+_4$

(Case 5) $s:\#_{\emptyset}(z)/@$

$z:e+_1@g$

$g:\#(@z)/@$

(Case 6) $s:\#_{\emptyset}(z)/+_1$

$z:e+_2@g+_3@z$

$g:\#(@z)/@$

(Case 7) $s:\#_{\emptyset}(z)/+_1$

$z:e+_2?_3g+_4@z$

$g:\#_{\emptyset}(\underline{tag}+_5?_6z)/+_7$

Figure 8.9: Storage Structures for Implementations Returned by SYNTHESIS

Step 4.0 of the algorithm adds 7 ordered implementations to improve existential referencing (this assumes there exists an ordering predicate for the elements of the structure, otherwise the ordered implementations are not added). In this example the ordered implementations will not necessarily be more efficient because the searching involved in the existential referencing is not related to an ordering among the elements in the structure (i.e., the strings). The algorithm terminates returning 14 possible implementations.

Figure 6.1 denotes the membership and variable binding information deduced from the program. Suppose that implementations for e_2 and e_3 were retrieved from the catalog. The hierarchically combine algorithm is invoked to combine implementations for the strings and e_1. Step 2.1 sets the elements field of each implementation for e_1 to pointer because e_2 and e_3 have separate existences (variables bound) and e_1 is a parent of e_2 and e_3. Pointers are used so that if e_2 or e_3 is bound to a variable (say X) and deleted, the entity can continue to exist in its present location without being copied. (This preference for non-copying is a heuristic which should be studied in more depth.) Finally, there are no problems combining the various implementations with top level sequential operators because the elements of e_1 are pointers which are presumed to be fixed size (this would not be true if there were different sized pointer representations, e.g., relative and absolute pointers).

Thus, 14 implementations would be generated for this modelling structure class.

8.4 Remarks on the Generation Process

The generation process described in this section imposes the following constraints on resulting implementations:

1. there must be one, and only one, entity for each symbol specified in the top level of the storage structure as occurring $\#_{\emptyset}$ times, and

2. all elements of a structure (except those with name selection referencing) must use one of the forms: values of the same type, values with descriptors giving type information, pointers to values of the same type, or pointers to descriptors.

Some such constraints are to be expected when one is developing a general procedure to replace hand-coded implementations. The question is how well does the general procedure perform? Our limited experience to date suggests that it does reasonably well in generating a set of alternatives with a wide range of space-time tradeoffs, but a more systematic evaluation should be undertaken.

There are several questions concerning the handling of the enumeration relation. First, in what cases do modelling structures not require an enumeration relation either because an alternative traversal scheme is available (perhaps as

specific implementation knowledge, e.g., binary tree traversals) or because the elements of the structures do not have to be enumerated? Which relation should be designated the enumeration relation if several candidates are available? Finally, the synthesis algorithm can be improved by recognizing those cases in which an implementation for a relation expicitly defined in the modelling structure contains a sequential or linked representation of elements which could be used to implement the enumeration relation. The current algorithm does not discover this, resulting in the synthesis of less space-efficient implementations.

A final remark concerns additional information that would improve the generation process described here. There are three specific kinds of information. First, does an entity have an existence outside of a structure? For example, suppose x is an element of S, and the variable binding relation indicates that some variable is bound to x. The methods used here cannot determine if this is binding only while x is a member of S (in which case the existence of x outside S need not be provided for). The second kind of information relates to elements occurring simultaneously in several structures. Consider the case where an entity x is a member of S and S' during execution. Now, if T_S is the set of time epoches during which x is a member of S, which of the following cases hold?

1. $T_S \cap T_{S'} = \phi$,

2. $T_S \subseteq T_{S'}$ or $T_S \supseteq T_{S'}$, or

3. $T_S \cap T_{S'} \neq \phi$.

Cases 1 and 3 require x to have a separate existence (i.e., an existence outside of the structure). Case 2, on the other hand, does not require that x have a separate existence. The last kind of information concerns the membership relation. Consider the example in Figure 6.1. Do the multiple creations of e_2 and multiple insertions of e_2 into e_1 mean that multiple occurrences of a single e_2 are inserted into e_1, or that several creations of e_2 are inserted? Because of the possibility of the former, an inefficient implementation for the latter (which is the more likely situation) must be used. It is not possible to deduce these three kinds of information using the methods described here. We feel that such information is used by programmers in selecting good implementations. Thus, in the context of the system being investigated, either meaningful interaction with the user or monitoring program execution is required if the most efficient implementations are to be generated.

9. Relationship to Other Research

In this section we relate the work reported here to other research in the specification of data abstractions and in implementation selection.

9.1 Specification

Liskov and Zilles [LI75] survey five classes of specification techniques using:

1. a fixed domain of formal objects.
2. an appropriate, but otherwise arbitrary formal domain,
3. a state machine model,
4. an implicit definition in terms of axioms, or
5. an implicit definition in terms of algebraic relations.

These classes are listed in order of increasing abstractness, e.g., a specification based on a fixed domain of formal objects generally includes more representational detail (information and constraints) than one based on algebraic relations. More detail may imply limitations on the range of possible implementations that can be generated. As a result of this observation and considerations of whether relationships between operations on a data type appear explicitly or implicitly in the specification [GU76], some researchers argue that data specifications should be represented using the more abstract techniques.

However, we chose to use a specification technique based on a fixed domain of formal objects because of the amount and kind of detailed information needed to make intelligent choices

among alternative implementation structures. Some examples of this kind of information are:

1. whether an operator modifies one of its arguments or makes a copy of the argument and modifies it,

2. whether an operator modifies the membership relation between its arguments (i.e., does an operator insert or delete an entity from a structure),

3. whether an access to a structure is restricted (e.g., root of a tree or top of a stack), and

4. whether an entity can be a member of more than one structure at a time.

This kind of information is difficult to infer mechanically from a program level description using either a more abstract specification technique (unless additional conventions on the interpretation of the specifications are made or supplimentary information is included with the specification)* or a less abstract specification technique (e.g., a PL/1, ECL, or PPL program level description).

- - - - - - - - - -

* This relates to the more general question of whether a uniform specification technique is used for the entire program development cycle in which later stages merely add more detail or whether different specification techniques are used in the different stages.

9.2 Selection

Early research on implementation structure selection systems proposed programming languages and translation environments in which implementation is decoupled from data specification and a user can separately specify or select an implementation [BA67, NA74]. Recently, work has concentrated on how data specification techniques are embedded in programming languages to enhance reliability [LI74, WU74], how to verify concrete realizations for abstract representations [HO72, SP75], and how to automate the selection of efficient implementations [GO74, LO76, SC75]. Because we have been concerned with the latter area, this section surveys only work on automatic implementation selection.

Low has developed most parts of a system which accepts programs written in SAIL, an Algol-based associative language [FE69], and produces executable code for a particular machine. In his system a fixed set of modelling structures and associated operators are provided (sets, lists, and a single ternary relation). Tompa, on the other hand, is not constrained to a fixed set of modelling structures, using instead a "substructure model" which is roughly equivalent to Codd's relational data-base model [CO70]. However, this system is intended as a design aid and thus does not accept complete program specifications nor does it produce executable program representations. The goal of our work is a system, not restricted to a fixed number of modelling structures, which accepts programs and produces executable code.

The information which Low's system uses to evaluate alternative implementations includes the relative frequency the primitive operators are performed and the expected number of elements in each structure derived either by interrogating the user or by monitoring sample executions. Tompa depends solely on user supplied data but the information used is very different (e.g., expected relative position of the element being accessed with respect to the average and maximum number of elements, expected number of times the element being searched for is the next one or is not a member of a structure, and expected number of comparisons during a binary search). Performance prediction is a serious limitation of these systems and our proposed system has the same difficulties. This problem is discussed further in the concluding section.

Another aspect of the implementation selection problem, not explicitly addressed by the others, is storage management. Often when two implementations for distinct modelling structures are chosen simultaneously by the selection algorithm, their execution costs may be less than the sum of their separate costs (e.g., in certain circumstances two stacks implemented sequentially can be organized in a region of memory to share available free space by placing them at opposite ends of the region and having them expand towards each other). We call this a coalescing. Reformulating the selection problem allows a branch and bound search algorithm to be used. This problem and solution are discussed in more detail in another paper [TO76].

Low used a hill-climb, or incremental search, selection algorithm which, depending on the particular problem, may not find the optimal solution. Tompa proposed a branch and bound algorithm similar to that used here but without coalescing [TO75].

Schwartz discusses the analysis and optimization of SETL programs with emphasis on the information needed for automatic selection of data structures [SC75]. The techniques discussed included most traditional optimization techniques (e.g., redundant expression detection, constant propagation, and peephole optimization) and some newer techniques (e.g., interoccurrence linking, value flow tracing, and determining inclusion and membership relationships). Mechanical recognition of known modelling structures, discussed in Section 7, uses a form of value flow tracing to deduce what operations are performed on the entities of a program. Also, a method for deducing membership relationships similar to that discussed by Schwartz is used here. The work reported here does not directly address the other analysis and optimization techniques. Nevertheless, a translator designed to produce efficiently executable implementations might include some or all of these techniques.

1Ø. Summary and Conclusions

In this paper we have presented an approach to improving the process of selecting efficient implementation structures. In particular, we described formalisms for modelling and implementation structures and three algorithms concerned with the generation of alternative implementations. The principal contributions of

this work were the demonstration of the possibility of mechanical recognition of known modelling structures and the development of a general algorithm for synthesizing implementations for those structures not recognized.

Two key features of the modelling formalism that make mechanical recognition possible are: (1) distinguishing between references to elements in a structure based on structural relationships and references directed at a specific element, and (2) using binary relations to model "structure." Making explicit the distinction between these types of references has not been done in other models of data structures. The full formalism, as defined here, may be too complicated to use in a real-world programming environment. Nevertheless, we believe that a formalism like the one developed will be necessary if mechanical recognition is desired.

The implementation structure formalism may be useful in other applications. It has proven to be a convenient, concise, and (within its limits) complete formalism for expressing and manipulating implementation structures.

Although a complete system based on the approach presented has not been developed, certain conclusions about the formalisms and algorithms can be drawn.

The performance of the mechanical recognition algorithm cannot be accurately assessed because it depends on examining its

application to a large number of example programs generated by different programmers. This is one direction for future research.

The synthesis algorithm has proven quite successful at generating reasonable alternatives, assuming a carefully defined library of single relation implementations and provided that the problem of joining operation code generators can be satisfactorily resolved.

Unlike the other two algorithms, the hierarchically combine algorithm was not particularly successful in that it tends to produce inefficient implementations. This poor performance is because the detailed input required to produce efficient implementations is not easily collected, and so the existing algorithm is overly simplified. The types of information required concern the necessity of separate existences for entities, the relationship between the times during which an entity is an element of various structures, and the existence of multiple occurrences of an entity in a single structure. Assuming that one is not willing to constrain the language for describing programs, there are three possible solutions:

1. a more complicated program understanding system could be constructed to deduce the information,

2. the user could be asked for the information, or

3. the actual running of the program could be monitored to gather the information.

Each of these alternatives has some disadvantages. The first solution, given current performance of knowledge based systems, may not yield sufficient information to justify the additional processing. While the second solution is certainly possible, the volume and detail of information that is required would be both time consuming to gather and tedious to provide. And lastly, the third solution, while again certainly possible, would not be acceptable if the program for which implementations are being generated were to be executed only a few times and/or it was exceptionally costly to execute.

As a result of these observations the need for specific types of user interaction becomes more apparent. We see two distinct classes of users who would use such a system in different ways. Those users with programs which are prohibitively expensive to execute will likely want to interact with the system to provide the information needed to generate efficient implementations. On the other hand, those users with programs which are not prohibitively expensive to execute with inefficient implementations, but which will be run many times will likely want to monitor actual executions to gather the necessary information. Notice that whichever approach is taken, user interaction is not just desirable but necessary.

Finally, any progress on the problem of automating the selection of efficient implementations depends on the ability to predict a program's execution performance. Progress to date in this area has been limited. As a result, one should not expect to

see practical use of such systems in the near future. We believe, however, that economically viable systems allowing user interaction to choose alternatives from an implementation library are feasible and can be developed for practical use.

References

BA67 Balzer, R. Dataless Programming. Proceedings AFIPS FJCC, Vol. 31 (1967), pp. 534-544.

BO73 Bobrow, R. J., et. al. UCI Lisp Manual. Technical Report #21, Department of Information and Computer Science, U.C. Irvine (October 1973).

CO70 Codd, E.F. "A Relational Model of Data for large Shared Databanks." Communications of the ACM, Vol. 13, No. 6 (January 1970), pp. 377-387.

DI69 D'Imperio, M.E. Information Structures: Tools in Problem Solving. ACM SIGMOD FDT NEWSLETTER, Vol. 1, No. 2 (1969).

FE69 Feldman, J.A. and P. Rovner. An Algol-Based Associative Language. Communications of the Acm, Vol. 12, No. 8 (August 1969), pp. 439-449.

GO74 Gotlieb, C.C. and F.W. Tompa. Choosing a storage Schema. ACTA Informatica, Vol. 3 (1974), pp. 297-319.

GU76 Guttag, J. Abstract Data Types and the Development of Data Structures. Supplement to Proceedings of Conference on Data: Abstraction, Definition and Structure, SIGPLAN Notices, Vol. 8, No. 2 (March 1976).

HO72 Hoare, C.A.R. Proof of Correctness of Data Representations, ACTA Information, Vol. 1 (1972), pp. 271-281.

HO75 Howden, W.E. Methodology for the Generation of Program Test Data. IEEE Transactions on Computers, Vol. C-24, No. 5 (May 1975), pp. 554-560.

KN69 Knuth, D.E. The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Reading: Addison-Wesley 1969.

LI74 Liskov B.H. and S.N. Zilles. Programming with Abstract Data Types. Proceedings Symposium on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4 (April 1974), pp. 50-59.

LI75 Liskov, B.H. and S.N. Zilles, Specification Techniques for Data Abstraction. IEEE Transactions on Software Engineering, Vol. SE-1, No. 1 (March 1975), pp. 7-19.

LO76 Low, J. and P. Rovner. Techniques for the Automatic Selection of Data Structures. Conference Record of the Third ACM Symposium on Principals of Programming Languages (January 1976), pp. 58-67.

NA74 Nadkarni, P.M. An Implementation Facility for High Level Level Data Structures. Ph.D. Dissertation, Department of Electrical Engineering, Princeton University (1974).

PR76 Proceedings of Conference on Data: Abstraction, Definition, and Structure. SIGPLAN Notices, Vol. 8, No. 2 (March 1976).

RO76 Rowe, L.A. A Formalization of Modelling Structures and the Generalization of Efficient Implementation Structures. Ph.D. Dissertation, Department of Information and Computer Science, U.C. Irvine (June 1976).

SC75 Schwartz, J.T. Automatic Data Structure Choice in a Language of Very High Level. Communications of the ACM, Vol. 18, No.12 (December 1975), pp. 722-728.

SI72 Sintzoff, M. Calculating Properties of Programs by Valuations on Specific Models. ACM SIGPLAN Notices, Vol. 7, No. 1 (January 1972), pp. 203-207.

TO75 Tompa, F.W. Evaluating the Efficiency of Storage Structures. Research Report CS-75-16, Department of Computer Science, University of Waterloo (May 1975).

TO76 Tonge, F.M. and L.A. Rowe. A Selection Algorithm for Coalesced Implementation Structures. In preparation.

WA73 Waldinger,R.J. and R.N. Levitt. Reasoning about Programs. Artificial Intelligence, Vol.5 (Fall 1974) pp. 235-316.

WU74 Wulf, W.A. Alphard: Toward a Language to Support Structured Programming. Computer Science Department, Carnegie-Mellon University (April 1974).

SP75 Spitzen, J. and B. Wegbreit. The Verification and Synthesis of Data Structures. ACTA Informatica, Vol. 4 (1975), pp. 127-144.