

3-

A STUDY OF DATA

by

HSAO-YING JENNIFER TIAO

**B. A., Tamkang University, 1980
Taiwan, Republic of China**

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

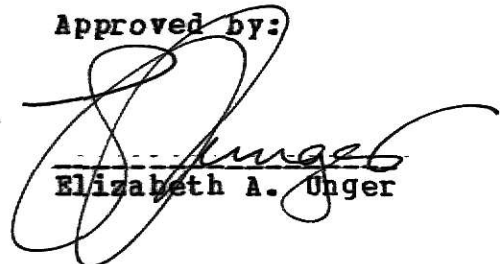
MASTER OF SCIENCE

Department of Computer Science

**KANSAS STATE UNIVERSITY
Manhattan, Kansas**

1983

Approved by:



Elizabeth A. Unger

LD
2668
R4
1983
T52
c.2

ACKNOWLEDGEMENTS

I would like to thank Dr. Elizabeth A. Unger, Dr. Rodney M. Bates, Dr. Roger T. Hartley, and Dr. Virgil E. Wallentine for serving as members of my committee. A special thanks goes to Dr. Unger for all the help and guidances she has given me throughout this reseach.

**THIS BOOK
CONTAINS
NUMEROUS PAGES
WITH DIAGRAMS
THAT ARE CROOKED
COMPARED TO THE
REST OF THE
INFORMATION ON
THE PAGE.**

**THIS IS AS
RECEIVED FROM
CUSTOMER.**

CONTENTS

	<u>Page</u>
ACKNOWLEDGEMENTS	ii
CONTENTS	iii
FIGURES	v
Chapter 1 INTRODUCTION	1
Chapter 2 DATA MODELS	
2.1 File Data Model	9
2.2 Liskov's Data Model	10
2.3 The Unger Model	21
2.4 Algebraic Specification	22
Chapter 3 THE UNGER MODEL	32
3.1 Definition of Objects	34
3.2 Definition of Structures	40
3.3 Examples of Simple Objects	42
3.4 Examples of Structure	49
3.5 Control of Computation: Definition and Example. . .	52
Chapter 4 PROBLEMS AND EVALUATION OF THE UNGER MODEL	
4.1 Strengths of the Unger Model	
4.1.1 Data Abstraction	63
4.1.2 Control and Structured Programming	66
4.1.3 Concurrency	70
4.1.4 Others	72
4.2 Problems of the Unger Model	73

Page

Chapter 5 INSIGHTS INTO SOME SOLUTIONS TO THE PROBLEMS

5.1 Problem 1	77
5.2 Problem 2	83
5.3 Problem 3	89
5.4 Problem 4	102
5.5 Problem 5	106

Chapter 6 SUMMARY

6.1 Evaluation of the Models Based on Five Criteria . . .	109
6.1.1 Evaluation on the Mealy's Model	111
6.1.2 Evaluation on the Liskov's Model	112
6.1.3 Evaluation on the Unger Model	114
6.1.4 Evaluation on the Algebraic Specification ..	115
6.2 The Role of Data Abstraction Played in Data Modelling	117
6.3 Future Work	118
BIBLIOGRAPHY	120

FIGURES

	<u>Page</u>
Figure 2.1 An Example of Procedure Taken from the Liskov's Paper	13
Figure 2.2 An Example of Cluster Taken from the Liskov's Paper	16
Figure 2.3 Stack Type in Algebraic Specification	23
Figure 2.4 Queue Type in Algebraic Specification	25
Figure 2.5 Binary Tree Type in Algebraic Specification ..	26
Figure 2.6 Binary Search Tree Type in Algebraic Specification	29
Figure 3.5-1 Ordering of An Action, Diffvar	56
Figure 3.5-2 A Construction of Diffvar from Requests for Other Actions	59
Figure 3.5-3 Examples of Stimulation	61
Figure 5.1 NEW FORM for Structure's Aggregation	87
Figure 5.2 An Example by Using NEW FORM for Aggregating A Structure	88
Figure 5.3 Sets of Longevity of A Structure Example Based on RULE 1	94
Figure 5.4 An Example of A Nested Structure on Case 2 by RULE 1	95
Figure 5.5 An Example of A Structure on Case 3 by Using NEW RULE	100

CHAPTER 1

INTRODUCTION

"We do not, it seems, have a very clear and commonly agreed upon set of notions about data -- either what they are, how they should be fed and cared for, or their relation to the design of programming languages and operating systems."

Mealy

Information in its real essence is probably too ambiguous and too subjective to be captured precisely by the deterministic and objective processes in a computer. The report is concerned with the question of how to model as much of reality as possible with computer representations of information. There are quite a number of structures available to represent information in computers such as file organization, indexed, hierarchical structures, network structures, and relational models. These structures give us a useful way to deal with information, but they do not always fit completely. Each has its strengths and weaknesses, or serves different purposes, or appeals to different users in different environments. No matter how we structure the "territory", the description is just another "map". Thus, is there such a natural model to articulate a slice of reality by constructs such as "entities (or objects)", "names", "relationships", and "attributes" to organize our cognition and discussion of information?

An information system, such as a data base, is a model of small, finite subset of the real world. Certain correspondences between constructs inside the information system and in the real world are expected. Ideally, the correspondence between these two environments would be one-to-one. However, it is not so easy to establish what construct in a computing system will be the appropriate representation. This construct might be a record, a part of it, or a group of them as a representative of the thing in reality. Take an employee who works in a certain department of a company as an example. We expect to have one record in the system of data processing for each employee's information.

In any situation of data processing, we are usually confronted with three systems [12]. The first is some part of the real world, the second is ideas about it existing in the mind, and the third is symbols on some storage medium, i.e., a machine representation of the idea. Each system is composed of entities, values, relations, procedures, and etc. The ideas would be models of the real world, and it is same that symbols or constructs in the information system are models of the ideas in the minds of users. Thus, we might draw the relationship among the reality, the ideas, and its representations.

We might say that data are fragments of the real world, and the information system accepts these fragments through a recognized representation. For example, the date of a certain day, arbitrarily chosen, "January 1, 1979" cannot be visualized or shaped out by human beings, but being expressed as an abstract or theoretical concept. There are various representations of it,

such as:

1 JAN 1979 010179 JAN. 1, 1979 2,443,874.

When we say "two", we never point at the exact and touchable integer in reality. However, we can see a number of representations, such as:

II 2 (010)₂ (2)₈ 0.2E01,

and all denote the same thing in reality with different formats or representatives. Thus, we can say, the construct or representative does not correspond to any physical object, but to the abstract idea of one thing in reality.

If a data construct is an abstraction of information in the real world, we need to formulate a description of this information so that it may be processed by computers. The abstraction is just able to model computation as it provides models of the real world. Fundamentally, characteristics of computers are deterministic, simplistic, structured, repetitious, unambiguous, unimaginative, and uncreative. Even some artificial intelligence experiments have simulated more elegant computer behaviour.

Unlike the real world, data in an information system represented perhaps by a data base has to be described both in physical aspect and in logical aspect. Physical description specifies the location, format, and organization of the data on disks, tapes, or other storage media. Logical descriptions declare the kinds of entities, the attributes, and the relationships among them. The user should know the meaning of the data in the system and the meaning of the relationships among

them. For example, a record in a data system contains "John Law" and "99" as two fields, without any clue from the system to show whether the "99" stands for his age, weight, weekly working hours, or something else. It is expected that the user knows what the fields mean; the semantics of data are important to data description.

We have not explained what is regarded as data in the system. One often tends to think of data in the system as the contents of various attributes of an entity or an object. In database, a fact is sometimes defined as an association between two fields: one giving the value of an attribute (weight --> 200 lbs.) and one identifying the entity having that attribute. The user's concern is what data in the system can be extracted, rather than how it is physically stored. The system is modelled as a set of named functions or maps, which return certain values when invoked with certain arguments. Once a user calls the function to be implemented, the function might involve simple access to stored data or complex traversal of data structure. Thus, the data content of the system is defined by this set of functions or maps, rather than physically stored data.

Before taking a view at several data models available in our fields, I will sketch a theoretical data model which is based on a number of old and obvious ideas in this field to be a prologue for this research work.

A theoretical model for data and data processing has been proposed in Mealy's work "Another Look at Data" [12]. The model is a system of sets of entities, values, data maps, and procedure

maps. The entities correspond to the objects in the real world to be recorded or computed. Attributes of these entities are correspondences between the things and the values. Alternatively speaking, attributes play as "relations", "maps", or "functions". Thus, the data maps, which assign values to attributes of the entities, are regarded as sets of ordered pairs of entities and values (i.e., data items).

Data Maps: Entities and Values

Data are supposed to record a set of facts about some set of entities, either real or abstract. Previously mentioned, a fact is sometimes defined as an association between the one giving the value of an attribute and the one identifying the entity having that attribute. Let E be a set of entities, V be a set of values, and D be a set whose members are maps of the form:

$$\theta : E \dashrightarrow V$$

where the Greek letter, θ , represents a map. And, we call D a set of "data maps". For example, a person John Law is 25 years old now, then the age-of data map, θ , would assign an age value to him. The form would be

$$\text{John Law} \xrightarrow{\theta} 25$$

or

$$\theta(\text{John Law}) = 25$$

While dealing with complex structural data, a structural map is expected, and it is defined with the form:

$$\sigma : E \dashrightarrow E$$

which is to have a single value for each argument to make the map be a function. The structural maps are maps of E into E . The

name for data items, such as (e, v) , in such a map is a "pointer".

A notion of separation of structural form from data is seen in the model. For instance, a floating-scale entity is composed of the mantissa and the characteristic. Both must be available during calculation, treated as one entity during arithmetic but as two entities during radix conversion. For consistency, it is often convenient to introduce two more auxiliary entities; each of these is a fixed scale number. We can create data types, such as arrays, lists, tables, etc., for any kind of structure required. The concept of abstraction is revealed here again that most of the detail involving the auxiliary entities is suppressed. The user only knows the description like:

```
type struct_data = array [3, 3 ] of integer
```

to describe a three-by-three array of integers. Things are set up in a way that a data map for a thing can be decomposed into a structural data map followed by a simple data map.

Data: Data Maps With Entities and Values

The information in the system is defined by a set of functions or data maps, rather than by physically stored data. Data objects exist through operations occurring in the system. We know that data maps assign values to corresponding entities. The question is whether entities, values, data maps, or all as a whole are data. What happens when data are processed? The first notion into our mind is that the procedure accepts values as arguments to produce a result which is that the values get defined, redefined, or undefined. Thus, the value set V is regarded as the data. But, the value cannot be redefined if it is the integer set. Same

reason to the entities. Thus, the data maps would be the data in the system, and would be a set of ordered pairs of entity and value, denoting as $\mu(e) = v$ or (e, v) . To redefine the value of the data item (e, v) is to redefine the map μ by removing that pair (e, v) and add a new pair (e, v') . The definition of data item is an element of a data map. A data element is a set of all data items associated with a given entity. In the conventional notion, a logical record is an example of data element, and field within that record is an example of a data item.

Procedures

The effect of a procedure is to redefine one or more data maps, or to change the value part of certain data items. As to structural data, a list processing with procedures which process structural data is suggested.

Data type is a fragment of data description, describing a portion of a system and an entity with its applicable maps. Data description describes machine data systems, representations, and organizations, and it is a specification of the maps in terms of procedures which implement the mapping and process the data items. A data type tells us what kind of data we are dealing. Usually, there are four generic data types seen in a system: string, boolean, numeric, and pointer.

Data processing takes place in the abstract realm, and its results should be representation (or machine) independent. In recent years, language design has tended to suppress the notion of representation so that the user frequently ignores the details irrelevant to him.

Over the past decade, a topic of research activity in programming language has been to explore the issues related to abstract data types (i.e., data abstraction) [14]. What abstraction (or modelling) means is to suppress irrelevant details of an object and to emphasize details significant to the user under the "current" context. Data abstraction is the application of abstraction to data objects. By the nature of abstraction, a small and finite subset of the real world is modelled (or abstracted). Correspondences between things in the real world and constructs inside of the information system must produce the expected data characteristics and behaviour.

In Chapter 2, a number of data models will be defined and introduced. The reader will find examples of the use of one of the models in Chapter 3. Chapter 4 defines the problems with this particular model and evaluates its strengths and deficiencies. Chapter 5 provides insights into some solutions to the problems uncovered in Chapter 4. And, a summary of the current state of the work and suggestive direction for future work will be stated in Chapter 6.

CHAPTER 2

DATA MODELS

In recent years, many data models have been proposed. Each data model differs in the style in which data objects and relationships between data objects are described. By a data model, the types of data structures visible to the user and the operations allowed on these structures are determined. Among different models of data, some are depicted to the readers here, and they are a) the file data model used in common programming languages [7], b) Liskov's model of "Abstract Data Types" [9], c) Unger's "A Natural Model for Concurrent Computation" [18], and d) algebraic specification [3]. The Liskov's model and algebraic specification will be discussed in detail in this chapter. The Unger model will be discussed in detail in Chapter 3.

2.1 File Data Model

The file data model, one of the forerunners of modern data models, is used by COBOL, FORTRAN, and other programming languages [7]. In the model, data are described by a declaration which lists the names of the fields of data in each record of a file, and describes the type and length of each field. First, in FORTRAN, variables are used to denote a quantity that is referred to by name rather than by its appearance as a value. For example, `I : INTEGER` states that "I" is the name of a variable which holds an object of abstract type INTEGER. Here, objects are created in conjunction with variable declaration. Secondly, people are quite

accustomed to working with large arrays of data in which the individual element of the arrays is indicated by subscripts. The use of such subscripted variables with all elements having the same type can handle very large amounts of data with a very minimum of programming effort, and one of advantages is that it is possible to manage an entire array without listing all of its elements explicitly. Thirdly, in conventional programming languages, by using a function (or procedure), a programmer is concerned only with what the function provides to him, but not concerned with the algorithm executed by the function. This has achieved a level of abstraction. The last type mentioned here is record seen in COBOL and PASCAL. A record, like an array, is a structured variable with several components which may have different types.

2.2 Liskov's Data Model

In Liskov's data model, an abstract data type defines a class of abstract objects which is completely characterized by available operations on those objects. When a programmer uses an abstract data object, he is concerned only with the behavior which the object has but not with the details how the behavior is implemented. Also, this model allows the set of built-in abstractions to augment with a new abstraction whenever discovered. Hence, in Liskov's Abstract Data Type, the set of useful abstractions is defined in advance, and a mechanism -- structured programming -- is provided, so that the abstractions, which the user requires, are constructed further.

Abstraction provides a mechanism, if as expected, to express relevant details but to suppress irrelevant details. By another programming mechanism, an abstract data type is used at one level, and realized at a lower level. Its lower level does not exist automatically by being part of the language. The realization of an abstract data type is done by writing an operation cluster, which defines the data type in terms of representation by already existing objects operations which can be performed on it.

There is a programming language given to facilitate the activity of cluster by allowing the use of an abstract data type without requiring its definition. The language supports two forms of abstraction, functional abstractions and abstract data types, by two corresponding forms of modules: procedures and operation clusters. Here, the functional abstraction denotes abstract operations which do not belong to any set of operations characterizing any abstract data type, but would be as a composition of the characterizing operations of one or more data types.

The way in which the language supports abstract data types is illustrated below. Objects can be created in combination with variable declaration. For example,

t : token

states that t is the name of a variable which holds an object of abstract type, TOKEN, but no creation of an object is made. Take an example of creation,

s : stack (token)

stating that s is the name of a variable which holds an object of

type, STACK, and a stack object is to be created and stored in s. There is more about the creation of an object later.

The language is strongly typed, and a defining operation on an abstract object is indicated by an operation call with a compound name. For example, two operations are

stack\$push(s,t)

and

token\$is_op(t)

each of which contains three parts. The first part of the compound name identifies the abstract type to which the operation belongs, STACK and TOKEN. The second part identifies the operation, PUSH and IS_OP. The last component identifies a parameter list of the operation calls, and it has at least one object of the abstract type to which the operation belongs, t or/and s. The presence of a type-name would enhance the understandability of programs. Another advantage from it is that operation calls are clearly distinguished from procedure calls.

Objects could also be created to be independent of variable declaration by the appearance of the type-name followed by parentheses. For example,

token (g, newsymb)

states that a token object, representing certain result out of g and newsymb, is to be created, and the information required to create the object is passed in a parameter list.

An example of a procedure, Polish_gen, (see Figure 2.1), which uses abstract data types illustrates the power of this concepts.

```

Polish_gen: PROCEDURE (input: infile,
                        output: outfile,
                        g: grammar);

    t: token;
    mustscan: BOOLEAN;
    s: stack(token);

    mustscan := TRUE;
    stack$push(s, token(g, grammar$eof(g)));
    WHILE stack$empty(s) DO
        IF mustscan
            THEN t := scan(input,g)
            ELSE mustscan := TRUE;
        IF token$is_op(t)
            THEN
                CASE token$prec_rel( stack$top(s), t) OF
                    "<" :: stack$push(s,t);
                    "=" :: stack$erasetop(s);
                    ">" :: BEGIN
                                outfile$out_str(output,token$symbol
                                                (stack$pop(s)));
                                mustscan := FALSE;
                            END
                OTHERWISE error;
            ELSE outfile$out_str(output, token$symbol(t));
        END
    END
    outfile$close(output);

```

```

    RETURN;
END Polish_gen

```

Figure 2.1 An Example of Procedure Taken
from Liskov's Paper. [9]

Polish_gen uses the functional abstraction, scan, to obtain a symbol of the grammar from the input string. The symbol is returned from the called procedure, scan, in the form of a type, TOKEN, as $t := \text{scan}(\text{input}, g)$, without revealing information about how the grammar represents symbols. In the procedure Polish_gen, there are five data abstractions: infile, outfile, grammar, token, and stack, and one functional abstraction, scan, is used. By the power of data abstraction, the procedure Polish_gen only needs to know what it needs, without knowing what I/O devices are being used, when the I/O actually occurs, nor how characters are represented on the devices.

For parameter input, an object of abstract type infile holds the sentence of the input language, it consists of three operations,

- <i> infile\$get, to obtain the next character;
- <ii> infile\$peek, to look at the next
character without removing; and
- <iii> infile\$eof, to recognize the end of input.

For parameter output, an object of abstract type outfile will accept a sentence of the output language; it contains two operations:

- <i> outfile\$out_str, to add a string of characters; and
- <ii> outfile\$close, to signify that the output is complete.

For parameter *g*, an object of abstract type *grammar*, which can be used to recognize symbols of the input language and determine their precedence relations, has a operation *grammar\$eof* to recognize the end of file symbol.

Other than these parameters, *Polish_gen* makes use of local variables, *s* and *t*, of abstract types *STACK* and *TOKEN*. The operations of type *STACK* and *TOKEN* shown in the procedure are

- *t*: token, to declare a variable *t* which holds an object of *TOKEN* type without object creation;
- *token(g, grammar\$eof(g))*, to create an object to be independent of variable declaration;
- *token\$is_op(t)*, to check if the object *t* of *TOKEN* type is an operand;
- *token\$prec_rel(t1, t2)*, to determine the precedence relation between two object of *TOKEN* type, *t1* and *t2*;
- *s : stack(token)*, to create a stack object and store it in *s* which is its variable name;
- *stack\$empty(s)*, to check if a stack has no more elements in;
- *stack\$push(s, t3)*, to push *t3* object onto the stack;
- *stack\$top(s)*, to return the top element of the stack;
- *stack\$pop(s)*, to obtain the top-of-stack token and expose a new top-of-stack token;
- *stack\$erasetop(s)*, to erase the top-of-stack token.

Several abstract data types are used in *Polish_gen*. And the definition of abstract data types is accomplished through the

operation cluster. The cluster of a type contains a set of characterizing operations through which a data type is defined. The abstract data type STACK used in Polish_gen will be used as an example to describe a programming object (see Figure 2.2).

```

stack: CLUSTER (element_type: TYPE)

    IS push, pop, top, erasetop, empty;

REP (type_param: TYPE) =
    (tp: INTEGER;
     e_type: TYPE;
     stk: ARRAY[ 1.. ] OF type_param;

CREATE
    s: REP(element_type);
    s.tp := 0;
    s.e_type := element_type;
    RETURN s;
END

push: OPERATION (s: REP, v: s.e_type);
    s.tp := s.tp + 1;
    s.stk[ s.tp ] := v;
    RETURN;
END

pop: OPERATION (s: REP) RETURN s.e_type;
    IF s.tp = 0 THEN error;
    s.tp := s.tp - 1;
    RETURN s.stk[ s.tp . ]

```

```

END

top: OPERATION (s: REP) RETURN s.e_type;
    IF s.tp = 0 THEN error;
    RETURN s.stk[ s.tp ];
    END

erasetop: OPERATION (s: REP);
    If s.tp = 0 THEN error;
    s.tp := s.tp - 1;
    RETURN;
    END

empty: OPERATION (s: REP) RETURN BOOLEAN;
    RETURN s.tp = 0;
    END

END stack

```

Figure 2.2 An Example of Cluster from Liskov's Paper [9].

In the cluster, a set of operations characterizes the abstract data type. A very general kind of stack object is defined here (or a prototype); the element type of the object is not defined when the cluster is defined. The type of the element would be indicated by the cluster parameter `element_type` for a particular stack object (or an instance of stack type) declared in detail later in the part of object creation. The whole cluster definition consists of four parts: cluster interface, object

representation, object creation, and a group of operations.

Cluster Interface

The cluster interface is the level which its user can see. The interface defines the name of the cluster, the parameters required to create an instance of the cluster, and a list of operations defining the abstract type which the cluster implements. In it, the actual TYPE of the elements of the abstract type is still not defined, e.g.,

```
stack : CLUSTER (element_type: TYPE)
        IS push, pop, top, erasetop, empty
```

The next three parts describe how the abstract type is actually supported.

Object Representation

In the view of users of the abstract data type, objects of that type are indivisible. However, objects are viewed to be decomposable into elements of more primitive types, such as integer, boolean, array, or record inside of the cluster. For example,

```
REP (type_param: TYPE) =
    (tp : INTEGER;
     e_type: TYPE;
     stk: ARRAY[ 1.. ]OF type_param);
```

corresponds to the form of REP description,

```
REP { ( <rep_parameters> ) } = ( <type_definition> )
```

which defines a new type of a stack object. The optional, "{}", <rep_parameters> makes it possible to delay specifying some aspects of the <type_definition> until an instance of the REP is

created. In the example, the `<type_definition>` specifies that a stack object is represented by a record with three component names `tp`, `e_type`, and `stk`. The parameter, `type_param`, specifies the type of element stored in the array `stk`. At the same time, this same type will be stored in the `e_type` component. The `tp` component represents the stack pointer to index the top element of the stack.

Object Creation

The create-code with the reserved word `CREATE` gets executed when an object of the abstract type is created, for example,

```
s : stack(token)
```

or

```
s: REP (element_type)
```

The parameters of the cluster are actually parameters of the create-code. They are accessible neither to the operations nor to the `<type_definition>` in the REP. Thus, information about the parameters must be saved explicitly through the insertion to each instance of the REP, e.g., the information about the parameter 'token' would be described through the record with components `tp`, `e_type`, and `stk`. In the create_code, an object of type REP is created, i.e., space is allocated to hold the object. Next, the initialization would be done. While the object is returned, its type is changed from type REP to the abstract type defined by the cluster. A type specification for the components of the structure, such as stacks and arrays, must be supplied before an instance can be created. Because constructs, such as stacks and arrays, define a class of types, they are called type generators. Each type in the class is supplied by its own type definition for

each of the type parameters of the type generator. Thus, a type generator defines an extensible class of types to provide the future needs.

Operations

A group of operations indicate the permissible accesses on the data types. There is always at least one parameter of type REP attached to each operation. By this parameter, an operation knows on which particular object to operate. As the parameter is passed from the caller to the operation, the type of this parameter will change from the abstract type to type REP. The type consistency, between object pushed onto a given stack and elements acceptable in the stack, is checked to achieve such a strongly typed language. On the operation push, the type of the second argument, *v*, is to be the same as the *e_type* component of the REP of the stack object which is the first argument of push.

During the implementation of cluster, a cluster or a procedure will be accepted by the compiler as a module. The module-names, then, are used to refer to data types and procedures. While the compiler processes a module, all information about the module is built into a description-unit, which includes:

- the location of the object code generated by the compiler;
- a description of the module (i.e., object) interface;
- a list of authorized modules to access the module.

The concept of abstract data type brings up a form most useful to the user to support data abstraction. By introducing a new linguistic construct — the operation cluster, abstract data types are supported. The implementation details irrelevant to a user

are hidden from him, and it is easier to lead him to an improvement in program quality, that is, programs will be more modular, and be easier to understand, to modify, and to maintain. It is believed that the abstraction-building-mechanism would be a useful feature of a very-high-level language.

2.3 The Unger Model

In the Unger model, through the use of structures, information involved in the solution of the problem is represented in terms of objects. A structure, which is an abstract notion of data, is formed from one or more objects, e.g., a record is a structure. By defining an object, a structure is described. An object is defined as a five-tuple with components: designator, attribute, representation, corporality, and value. The designator is a sequence of names, that identifies the object uniquely, including the context of creation, user-defined name, instance information, and alias sets. The attribute is a set of descriptors defining the object's logical organization: the underlying atomic representation, the internal structure, and the external relationships. The representation defines the physical organization: coding, compression, and overlay characteristic of the object. The corporality describes longevity, environment, replications, availability, and authorized uses. The final component, value, is either an atomic value or a structure of atomic values compatible with the attribute. Also see more in Chapter 3.

2.4 Algebraic Specification

With what criteria would we measure a good data type specification? A data type specification is an abstraction to define a type with rigorous definition of operations but representation-independent. Among many criteria of measuring, such as formality, extensibility, constructibility, comprehensibility, and wide range of applicability, two major criteria can be used to measure the value of a specification notation: (i) the ease of constructibility by the specification, and (ii) the ease of comprehensibility of the resulting specification. The four models will be evaluated based on a number of criteria in Chapter 6.

The particular specification technique, algebraic specifications by Guttag, Zilles, and Goguen (1975), appropriately meets the two criteria stated previously. After examining a number of examples later, the virtue of this specification would be clearly seen that a fairly complex object has been completely defined by a few lines only.

The algebraic specification technique is a method to describe a data type and to define the object using natural language and mathematical notation without concerning its eventual implementation. Since software designers usually have had programming experience, the use of a programming-like specification is suitable.

In this formalism, not many features assumed to appear in conventional programming languages are permitted. The permitted features include (a) free variable, (b) if-then-else expressions, (c) boolean expressions, and (d) recursion. In addition,

procedures are restricted to be single-valued and to have no side-effects. This approach is strongly recommended for several reasons. First, the resulting specification can clearly express the desired concept. Second, the separation of values and side effects makes a specification clearer and simpler. Third, these features allowed in it can be easily axiomized.

Here begins some examples of data type with this approach, and they include Stack, Queue, Binarytree, and Bstree (binary search tree) [11].

Type 1: Stack

In the beginning, let us look at the very simple example of a Stack data type given in Figure 2.3.

```

TYPE Stack (item)
1. DECLARE  NEWSTACK{} --> Stack
2.          PUSH(Stack,item) --> Stack
3.          POP(Stack) --> Stack
4.          TOP(Stack) --> item U { UNDEFINED }
5.          ISNEWSTACK(Stack) --> Boolean;
6. FOR ALL  s ∈ stack, i ∈ item LET
7.          ISNEWSTACK(NEWSTACK) = true
8.          ISNEWSTACK(PUSH(s,i)) = false
9.          POP(NEWSTACK) = NEWSTACK
10.         POP(PUSH(s,i)) = s
11.         TOP(NEWSTACK) = UNDEFINED
12.         TOP(PUSH(s,i)) = i
13.END

```

END Stack

Figure 2.3 Stack Type in Algebraic Specification.

The operations available to manipulate a stack are NEWSTACK, PUSH, POP, TOP, and ISNEWSTACK. In the DECLARE statement, the type of input and output of an operation is listed. These operations now we have are functions which return a single value and allow no side effects.

The notation used in this technique is stated here. All reserved words (e.g., TYPE, LET) and operation names (e.g., POP) are denoted in upper case. Type names begin with a capital letter (e.g., Stack). Free variables are written in lower case (e.g., s and i). A list of parameters following the type name or operation name is within parentheses, and these parameters might be type names or free variables whose range is the set of all types. In the example of Stack, "item" is such a variable which could be any other data type. Within the block FOR ALL and END, the equations are the axioms describing the semantics of the operations. With the structure of recursion, these axioms may be easier to be interpreted than they look. The only other stacks appearing are in the operation PUSH(s,i), where s is any stack and i is the most currently inserted item.

Type 2: Queue

The entire specification technique used in writing algebraic axioms has been seen in the first example of Stack, and the next one is to introduce conditionals (i.e., IF-THEN-ELSE) into the

right-hand sides. In Figure 2.4 of type Queue, there are six operations: NEWQ, ADDQ, DELETEQ, FRONTQ, ISNEWQ, and APPENDQ.

```

TYPE Queue (item)
1. DECLARE  NEWQ () --> Queue
2.          ADDQ (Queue,item) --> Queue
3.          DELETEQ (Queue) --> Queue
4.          FRONTQ (Queue) --> item U { UNDEFINED }
5.          ISNEWQ (Queue) --> Boolean
6.          APPENDQ (Queue, Queue) --> Queue ;
7. FOR ALL  q, r ∈ Queue, i ∈ item LET
8.          ISNEWQ (NEWQ) = true
9.          ISNEWQ (ADDQ (q,i)) = false
10.         DELETEQ (NEWQ) = NEWQ
11.         DELETEQ (ADDQ (q,i)) =
12.             IF ISNEWQ (q) THEN NEWQ
13.             ELSE ADDQ (DELETEQ (q),i)
14.         FRONTQ (NEWQ) = UNDEFINED
15.         FRONTQ (ADDQ (q,i)) =
16.             IF ISNEW (q) THEN i ELSE FRONTQ (q)
17.         APPENDQ (q,NEWQ) = q
18.         APPENDQ (r,ADDQ (q,i)) = ADDQ (APPENDQ (r,q),i)
19. END
END Queue

```

Figure 2.4 Queue Type in Algebraic Specification.

The recursion on q of the type Queue is seen in FRONTQ and DELETEQ. Taking the FRONTQ of the empty queue is UNDEFINED. Or, the returning value would be the most recently inserted item i , and the remainder of the queue is q . If q is not empty, then FRONTQ is recursively applied to q . It is the same for the operation DELETEQ.

Type 3: Binary Tree

The type binary tree is the next to be introduced. Some additional points in this type are as follows:

- Point1 - The omitted operations are excluded from a type because they make use of the operations of another data type;
- Point2 - A new approach is that a user can create a nonprimitive operation in terms of some primitive operations of a certain type.

The specification of the type Binarytree is given in Figure 2.5.

TYPE Binarytree (item)

1. DECLARE EMPTYTREE() --> Binarytree
2. MAKE(Binarytree,item,Binarytree) --> Binarytree
3. IEMPTYTREE(Binarytree) --> Boolean
4. LEFT(Binarytree) --> Binarytree
5. DATA(Binarytree) --> item U { UNDEFINED }
6. RIGHT(Binarytree) --> Binarytree
7. ISIN(Binarytree,item) --> Boolean ;

```

8. FOR ALL  l, r ∈ Binarytree, d,e ∈ item LET
9.          ISEMPYTREE(EMPTYTREE) = true
10.         ISEMPYTREE(MAKE(l,d,r)) = false
11.         LEFT(EMPTYTREE) = EMPTYTREE
12.         LEFT(MAKE(l,d,r)) = l
13.         DATA(EMPTYTREE) = UNDEFINED
14.         DATA(MAKE(l,d,r)) = d
15.         RIGHT(EMPTYTREE) = EMPTYTREE
16.         RIGHT(MAKE(l,d,r)) = r
17.         ISIN(EMPTYTREE,e) = false
18.         ISIN(MAKE(l,d,r),e) =
19.             IF d = e
20.                 THEN true
21.                 ELSE ISIN(l,e) OR ISIN(r,e)
22. END
END Binarytree

```

Figure 2.5 Binary Tree Type in Algebraic Specification.

Let us read the meaning of each operation.

- EMPTYTREE creates the empty tree.
- MAKE joins two trees together with a new root.
- LEFT returns the left subtree or the right subtree
or RIGHT of a node.
- DATA accesses data at a node, and returns an item or
"undefined".

- ISIN searches for a given item to return a boolean value.

The type of binary tree naturally reminds us of the usual traversal methods: preorder, inorder, and postorder. Should an operation be part of the specification or not? The choice of which operations to be included in a specification is arbitrary. See one example of three operations below,

INORD(Binarytree) --> Queue;

and its axioms,

FOR ALL $l, r \in \text{Binarytree}, d \in \text{item},$
 $\text{INORD}(\text{EMPTYTREE}) = \text{NEWQ}$
 $\text{INORD}(\text{MAKE}(l, d, r))$
 $= \text{APPENDQ}(\text{ADDQ}(\text{INORD}(l), d), \text{INORD}(r))$

In most applications, the tree is usually ordered. In order to extend the binary tree into an ordered one, called as binary search tree given next in Figure 2.6, a new approach is that a non-primitive INSERT operation is formed in terms of the primitive operations of type Binarytree. In general, a data type specification always defines only one type, but it may have the operations of other data types to accomplish it.

Bstree (Binary Search Tree) type is defined to be a binary tree with data items at each node that any item is alphabetically greater than any item in its left subtree and less than any in its right subtree. With some altered axioms and a new operation, the Binarytree specification is transformed into type Bstree. See Figure 2.6 for the specification of Bstree. The differences between these two types include the altered second axiom for ISIN;

a new operation INSERT, which searches for an item in a binary search tree and inserts it if it is not there. Since the only way to create a binary search tree is the operation INSERT, not MAKE any more now, the operation MAKE becomes a hidden function with a star ('*') attached in the new specification to indicate that it is not accessible any more.

```

TYPE Bstree (item)

1. DECLARE    EMPTY() --> Bstree
2.            *MAKE(Bstree,item,Bstree) --> Bstree
3.            ISEMPYTREE(Bstree) --> Boolean
4.            LEFT(Bstree) --> Bstree
5.            DATA(Bstree) --> item U { UNDEFINED }
6.            RIGHT(Bstree) --> Bstree
7.            ISIN(Bstree,item) --> Boolean
8.            INSERT(Bstree,item) --> Bstree ;
9. FOR ALL    l,r ∈ Bstree, d,e ∈ item LET
10.           ISEMPYTREE(EMPTY) = true
11.           ISEMPYTREE(MAKE(l,d,r)) = false
12.           LEFT(EMPTYTREE) = EMPTYTREE
13.           LEFT(MAKE(l,d,r)) = l
14.           DATA(EMPTYTREE) = UNDEFINED
15.           DATA(MAKE(l,d,r)) = d
16.           RIGHT(EMPTYTREE) = EMPTYTREE
17.           RIGHT(MAKE(l,d,r)) = r
18.           ISIN(EMPTYTREE,e) = false
19.           ISIN(MAKE(l,d,r),e) =
20.               IF d = e THEN true
21.               ELSE IF d < e THEN ISIN(r,e)

```

```

22.                                     ELSE ISIN(l,e)
23.      INSERT(EMPTYTREE,e)
24.      = MAKE(EMPTYTREE,e,EMPTYTREE)
25.      INSERT(MAKE(l,d,r),e) =
26.      IF d = e
27.      THEN MAKE(l,d,r)
28.      ELSE IF d < e
29.      THEN MAKE(l,d,INSERT(r,e))
30.      ELSE MAKE(INSERT(l,e),d,r)
31. END
END Bstree

```

Figure 2.6 Binary Search Tree in Algebraic Specification.

So far, we have seen four examples of the use of algebraic specification. At this point, let us view the virtues of the specification. It is criticized that the recursion forces into inefficient coding, e.g., the FRONTQ operation in Queue finds the front element of the queue by starting at the last element. However, the contents in the specification should not be viewed as describing the actual implemented program but just as a means to understand what the operation is to do.

What is the so-called constructor set? For each type, there is a subset of the basic operation set to satisfy the property that the data type can be represented using only constructor set operations. The axioms of the type, then, show how each non-

constructor-set operation behaves on all possible instances of the data type. Take the previously given example, Queue, to illustrate it. Out of those four operations, NEWQ and ADDQ are selected as the constructors. A set containing $n \geq 1$ items is given as

NEWQ

or

ADDQ(...,ADDQ(ADDQ(NEWQ,i₁),i₂),...,i_N),

where $N \geq 1$.

The item i_1 is at the front and i_N is at the rear. Then the axioms can be looked at as the rules which show how each operation of non-constructor-set (DELETEQ, FRONTQ, ISNEWQ, and APPENQ) acts on such an expression (called a "canonical form" which is defined next).

Each specified type can be represented by a single expression, and it is called the "canonical form" which contains the minimal set of operations -- constructor set. For example, the canonical form for type the integer Set is:

INSERT(INSERT(...INSERT(EMPTYSET,i₁),...,i_{N-1}),i_N)

with EMPTYSET and INSERT as the constructors. Constructibility of algebraic specification can be enhanced by a canonical form which can be derived mechanically.

In this chapter, the models have been introduced in reasonable detail, except the Unger model. The detail of and the use of the Unger model will be described in next chapter. The problems with the model will be investigated; and, formally, some solutions will be discussed.

CHAPTER 3

THE UNGER MODEL

This chapter illustrates some uses of the Unger model by means of examples. These examples will illustrate how basic properties of the model allow representation of information involved in the solution of problems through the use of objects and multi-object structures. The model utilized to create all examples is based on the pre-simplified model. The readers will see the simplification introduced right after the full work of the model. The simplified model is courageous for focus on logical properties without the loss of generality. One of advantages of simplification is to more clearly focus on the logical properties of the object. Minimal attention will be given to issues concerning implementation; for example, the representation, the third component of an object, will be ignored.

A structure is defined as one or more objects, and, therefore, a structure must be described by defining an object which is the basic component of a structure. An object is defined as a five-tuple, where the five components are designator, attribute, representation, corporality, and value.

Important definitions of the basic object of the information structures are presented in Section 3.1. The multi-object structure is defined in Section 3.2. To illustrate the use of applying definitions of objects and structures, examples of objects and those of structures are given in Section 3.3 and 3.4

respectively. Finally, the control structure of computation execution is introduced in 3.5. In this last section, action, request, ordering of the request's execution, detailing of an action, and conditions to control paths within a computation are stated.

A combination of mathematical notation and the Backus-Naur Form (BNF) was selected to define the linguistic forms used for illustrating examples. The mathematical notation of sets, tuples and mappings is employed. First of all, a number of definitions extracted from Unger's "A Natural Model for Concurrent Computation" [18] will usher us into the model.

<u>Notation</u>	<u>Meaning</u>
Upper case English letters e.g., A, B, C, ...	Set name
Lower case English letters e.g., a, b, c, ...	Set member
{ enumeration } e.g., A = { a, b, c }	Set definition
(enumeration) e.g., (a, b, c)	Tuple
Lower case Greek letters e.g., α , β , δ , ...	Map name
Post-operator e.g., $a\alpha = b$	Mapping
a^+	One or more occurrences of a
a^*	Zero or more occurrences of a
Parentheses i.e., ()	Grouping

3.1-Definition of Objects

All information involved in the solutions of problems is represented by the use of data "objects" or "structures" of objects. The objects in the model are drawn from a data universe as follows:

Definition 1. The UNIVERSE of possible objects, O , is a five tuple:

$$O = D \times A \times R \times C \times V, \text{ where}$$

D is the designator space,
 A is the attribute space,
 R is the representation space,
 C is the corporality space,
 V is the value space.

Thus, an individual object may be defined as follows:

Definition 2. An OBJECT, o , is a five tuple:

$$o = (d, a, r, c, v),$$

where the five components of the tuple are:

$d \in D$ is the designator,
 $a \in A$ is the attribute,
 $r \in R$ is the representation,
 $c \in C$ is the corporality, and
 $v \in V$ is the value.

The components of an object are expressed in terms of mappings as follows:

$$d \equiv \cdot o \delta \quad \text{where} \quad O \xrightarrow{\delta} D$$

$$a \equiv \cdot o \alpha \quad \text{where} \quad O \xrightarrow{\alpha} A$$

$$r \equiv \cdot o \rho \quad \text{where} \quad O \xrightarrow{\rho} R$$

$$c \equiv \cdot o \xi \quad \text{where} \quad O \xrightarrow{\xi} C$$

$$v \equiv \cdot o \gamma \quad \text{where} \quad O \xrightarrow{\gamma} V$$

where each of these mappings is single-valued. Two objects are equal, $o_1 = o_2$, if and only if $d_1 = d_2$, $a_1 = a_2$, $r_1 = r_2$, $c_1 = c_2$, and $v_1 = v_2$. An object is uniquely referenced by its designator, so we can use d to represent an object, e.g., $\overline{o\delta} \equiv \overline{d\delta}$ and $\overline{o} = \overline{d}$. Therefore, if $d_1 = d_2$ for two objects $(d_1, a_1, r_1, c_1, v_1)$ and $(d_2, a_2, r_2, c_2, v_2)$, then $o_1 = o_2$.

Designator

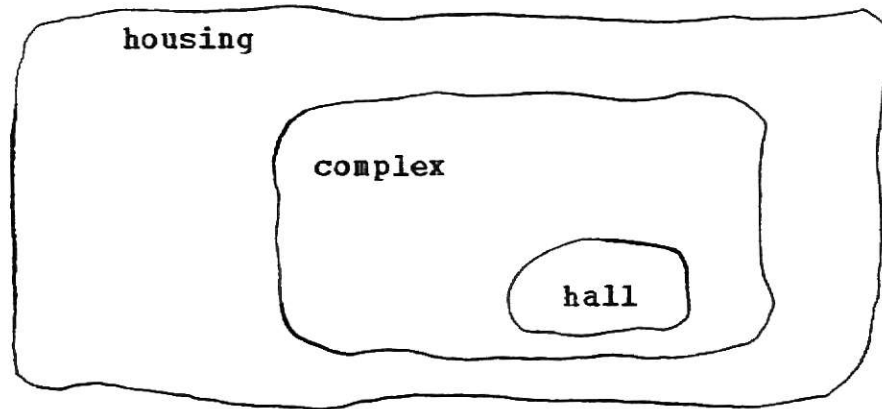
Each object within the model has a unique identifier called the designator which is a name or sequence of names. The designator identifies an object, and some of its names are user supplied or internally generated.

Definition 3. The designator of an object is a four tuple (c, u, i, a) where

- c is the context of creation for the object that is represented by a finite ordered sequence of zero or more names, i.e., (c_1, c_2, \dots, c_N) .
- u is the user name that is represented by a finite ordered list of one or more simple names, i.e., (u_1, u_2, \dots, u_N) which are created by the user.
- i is the instance of the object that is represented by zero or more names or notations indicating the specific instance of a class of objects with identical c and u names.
- a is the alias set of designators that are used to reference the object in other contexts.

An object exists if and only if its designator has been specified. The designator must exist before the other components. The rest would be in such an order, attribute, representation, corporality, then value. The existence of each component can be ascertained by the existence function (e.g., \overline{d}). An existence function has a truth value indicated by superscoring the component to determine if a component has been bound. For an object $o = (d, a, r, c, v)$, the existence for each component would be achieved by,

$\overline{o\delta} \equiv \overline{d}$: the existence of the object is equivalent



If the object is passed to an action higher in the hierarchy of actions, a wider context is created for the object. Thus, it has an environment inclusive of all actions subordinate to the wider context.

The user-name is represented by

$$\langle \text{user-name} \rangle ::= (\langle \text{name} \rangle_1) * \langle \text{name} \rangle$$

which is an ordered sequence of names instead of a single name for illustrative purposes.

The instance is composed of three subcomponents: spatial coordinates, chronological identity, and sequence position (order of the creation). Any combination among three may be specified by the user. The existence of either chronological identity or sequence position is optional.

Attributes

The attribute contains information about (1) the type of the object's value, (2) the internal structure within the object, and (3) the external relationship of this object to other objects. The type of the object's value can be any type represented naturally in hardware (i.e., "atomic") or described by an aggregation (i.e., "structure"). The details of type will be covered in the part of value later. The internal structure of an object describes the relationship between the components of the object's value which may be a simple value for an atomic object or a number of values for a structured object. The external relationship specifies the logical position of the object to other objects in the system. For example, a record might be a part of certain file.

Representation

The representation information for a given type need not be made available to all users of objects of that type, thus, the abstraction is provided for the user. In

most of cases, we will put a null, ϕ , to indicate the ignorance of representation. Otherwise, a Greek alphabet, Ω , to indicate that the representation would be the system-dependent primitive at the machine level.

Corporality

The corporality provides information about longevity, environment, replications, availability, authorization, and access of the object.

Definition 4. Corporality of an object is a four tuple (l, p, r, e) where

l is the longevity of the object and its durability in a given environment,

p is the location where the object is to be found,

r is an indication of the number of identical copies of the object, and

e is a list of authorized uses.

Nevertheless, the currently simplified model restricted the corporality to consist of two components: longevity and the availability of authorized uses. The simplification results in the conditions: (1) that the object can be found, (2) that there are no replications.

Within the model, the longevity is classified into four categories: fixed, static, dynamic, and fluid. These four categories are ordered in terms of capability to allow change. Thus, a fluid value is most accepting of change, however, a fixed value is least. If an object is not specified with a value of longevity, it will be considered as dynamic, by default. As an object passes from one context to another, its longevity may be changed under certain constraints. First, it may be changed to a more restrictive category as passing from its context of creation to other contexts. Yet, a restriction is noted that the longevity may never be made less restrictive than when it was originally defined for the object.

This tells a fact about objects with a fixed longevity that fixed objects may not change, and they exist prior to the existence of the context in which they are used. For example, the alpha-numeric characters are this sort of objects as they were created prior to the model.

Objects with longevity static are defined by actions within the model. (Note: A simple action is a three-tuple

(m, a, r) where m is the material required, expressed as a list of objects, a is the designator of the action, r is the result produced by the action, expressed as a list of objects. For example, an action QUADROOT(r1, r2 ; a, b, c) is composed of QUADROOT as its designator, r1 and r2 as its objects of result, and a, b, and c as its objects of materials required. More in Section 3.5.) The environment of a static object is smaller than that of a fixed object. Within the environment, each component of the static object is bound to the object. Once a component of this type of object comes into existence, it is bound to the object in all environments. For example, a static constant would be created within the model rather than prior to the system. For this type of object, the sequence number of its designator will not be greater than one but exactly one.

Objects with longevity dynamic are defined by actions within the model. Such an object may change its value overtime, and simultaneously, a new incarnation would be invoked. Therefore, the object must have a chronological identity, t, and/or a sequence number, o, in the user-name component. Each time when a new value associates with the object, the u (user-name) part of the designator is updated by setting the new reading of chronological identity or/and by generating a new sequence number in a monotonically increasing sequence. Thus, a new object is created. Previous incarnations of an object could be purged by a specific action. Once the dynamic object with a series of incarnations is referred to, relative reference d.{+-}n, n ≥ 0, retrieves the incarnation created n steps previously. To an unqualified reference, the most recent incarnation is retrieved. If n = 0, the current incarnation is specified.

The last category for longevity is fluid which is defined by actions within the model. Objects with fluid longevity may change value at any time with a restriction that change must occur between uses. Also, no previous incarnations are available. That is to say the sequence number, o, of the designator will not be greater than one for this kind of object.

The second component of corporality, authorization, indicates allowed uses to protect an object. It might be desired to provide a journal of uses and/or unauthorized attempted uses.

In terms of mappings, the two-tuple simplified corporality (l, e) for an object is defined in the following:

$$l = cv \text{ where } c \xrightarrow{v} \{ \text{fixed, static, dynamic, fluid} \}$$

$e = c\phi$ where $c \xrightarrow{\phi} u$, u is a set of authorizations.

Value

The value represents the informational unit for the entire object which might be a simple object or a multi-object structure. The information may be an atomic value, a complex of values obtained through the use of spatial coordinates, a collection of objects (i.e., structure), or the detail of an action, compatible with the attribute. An atomic value for an object might be boolean, integer, real, or character. As for a nonatomic type of a structured object, the whole value is defined when a structure is created via aggregation. Nonatomic values defined in the model and available for users are set, collection, ordered collection, and action.

The specification of an object is simplified without significant loss of generality. Moreover, the simplification moves the focus more on the logical properties of the object. One of simplified points is the ignorance of representation, and it ignores the concern during implementation of a specific language. The definition of an object after simplification is stated below.

Definition 5. The simplified object is a five tuple (d, a, ϕ, c, v) where

- d the designator is a three tuple (c, u, i) and i is a three tuple (s, t, o) where either t and/or o is null,
- a is the attribute, as defined previously,
- c the corporality is the tuple (l, e) where l is longevity defined previously, and e is a binary indicator of availability, and,
- v is the value of the object expressed as atomic value (integer, real, bit string), a number of atomic values specified via aggregation.

3.2 Definition of Structures

A multi-object structure is defined via a process called aggregation. As in the Liskov's abstraction of data, the components of the object composing a structure are not available to the user once the process of aggregation is done. In Liskov's

terminology, a structure is called "type". Three definitions provided in the Unger model describe a structure are extracted as follows:

Definition 6. A structure is the three tuple (o, p, c) where

- o is a structure (d, a, r, c, v) composed of N objects
 $o_i = (d_i, a_i, r_i, c_i, v_i),$
 $1 \leq i \leq N.$
- p is a set of partitions which can be referenced as a structure, and,
- c is a set of legal actions which can be performed on the structure $o.$

Definition 7. A structure is created by defining a prototype tuple (n, m, r, o, p) where
 n is the designator of the structure (i.e., type), e.g., set.

- m is the material to be supplied by the user for composing the structure (in terms of abstract representations of object, i.e., the fields of a record, $f_1, f_2, f_3, \dots, f_N$).
- r is the representation of the structure stated in terms of atomic types and available structures, e.g., a record (f_1, f_2, \dots, f_N) is an ordered collection of a finite number of objects.
- o is the list and definition of the set of pre-defined operators and operations, and,
- p is the set of names of the predefined partitions and the definitions of these partitions.

Definition 7-A. An occurrence of a structure

- (nd, md, rd, od, pd) is a tuple (d, n, m, o, p)
- d is the designator of the structure,
- n is the designator, nd , of a prototype structure tuple,
- m is the material to be used in this structure, d , of the structure nd ,
- o is the list of operators and operations defined for the instance d of nd ,
 (if o is null then od is valid, while
 if o is not null then $o \subseteq od$ and

o is valid).
 p is a set of partition names defined for
 this instance of nd (if p is null
 then pd is valid, while if p is not
 null, then p is valid if $p \subseteq \underline{pd}$).

Denoted by the BNF definition, the occurrence of a structure will be represented as follows:

```

<structure occurrence> :: =
  <designator> : structure <name> { <material> }
                uses ( <operations> )
                partitions ( <partitions> )

<list> :: = ( <designator> .)* <designator>
<material> :: = <list>
<name> :: = <designator>
<operations> :: = <list>
               and,
<partitions> :: = <list>.

```

Although a mechanism, which is defined in the model, allows structures to be created, there are four embedded structures included in the model. The detail of an action, ordered collection, unordered collection, and set are used as fundamental structures. Other structures could be created by aggregating atomic objects and/or existing structures.

3.3 Examples of Simple Object

Let us create several simple objects based on the pre-simplified definition and two structures: record and array. Supposedly, we need to have this model create objects, or structures if needed, to represent personal record of a student in a college. The information about a student included in examples are his official name, alternative name, social security number, sex, pursuing degree, birthdate, and local address. So far, we

have not reached the portion of structure creation, so, some basic structure would be assumed to exist and to be available for use in our examples below, e.g., `character-string(length : integer)` and `set` (see Definition 8 for `set`).

Definition 8. A set is an unordered collection of objects, $s = \{ s_1, s_2, \dots \}$, which are homogeneous in type of value and such that for any two elements, $s_i, s_j \in S$, $s_i \neq s_j$. A set will be denoted by enclosing it in braces, e.g., `{ William, Bill, Willy }`.

The prototype for an object is `o(d, a, r, c, v)`, and the instance of such a prototype `o`, let us say, `oi(di, ai, ri, ci, vi)` for any future occurrence, where $i = 1$ to N . There are seven objects used in the following as examples, i.e., student's name, alternative name, social security number, sex, degree, birthdate, and local address, and defined below.

Object <1>: An object `o1(d1, a1, r1, c1, v1)`, called as `NAME`, represents the name of a student. Thus `o1(NAME, char_string(25), byte, static, atomic)` is an example of an object, where

`d1 = ϕ •• NAME •• (65) . 1980`

an object without creation context, with user name `NAME`, a spatial coordinate of 65, and a chronological name of 1980.

`a1` : The attribute describes

<1> the type of the object's value, `char_string(25)`, is an occurrence of a pre-defined structure object `char-string(length)` with 25 characters long.

<2> since, here, the value of an object is a structure object, a character string, the internal structure is a sequence of character values.

<3> and, there is a logical relationship of equality with another object, introduced next, ALT_NAME. They both identify the same uniqueness.

r1 : coding of bytes,

```
c1 = (l1, p1, r11, e1)
    = (static, { Reg-Dept-FLG80F }, r11
        { create, delete } ),
where r11 (a1, s1, g1) = (true, 1, true).
```

By r11, we know this object is available, with 1 copy, and with copiability.

v1 = "Kelly S. Folley ".

The type of the value is atomic, because the structure char-string is available in the model. It is used as an object, so the user does not have to create a structure of it. If the type is not pre-defined prior to the use in the model, then the type of the value would be "structure" rather than "atomic". The atomic object is any type represented naturally in the hardware, such as integer or string.

Object <2>: Similarly for an object called ALT_NAME, there is

```
o2 = (d2, a2, r2, c2, v2)
    = (ALT_NAME, set, Ω, dynamic, structure),
where
```

```
d2 = APPLICANT . GRADSCHOOL . CSDEPT •• ALT_NAME ••
    (116) . 1980 . 3
```

It tells that an object was created in the environment APPLICANT, and passed into the environment GRADSCHOOL. Currently, CSDEPT is its local environment, with user-name ALT_NAME, a chronological name of 1980, and a sequence number of 3 indicating the third time of generating new value on the object.

a2 : The attribute describes

<1> the type of the object's value, set, is an occurrence of a pre-defined structure object, set, existed in the model. Set is one of four embedded structures included in the model, although a mechanism is defined which allows other structures to be created. See Definition 8 for the set,

<2> since the value of the object, ALT_NAME, is a structure object, set, the internal structure of it is an unordered collection of names with char-string type. The value of ALT_NAME could be an empty set,

<3> and, there is a logical relationship of equality with the object, NAME. Both identify the same uniqueness.

r₂ : It is denoted here, as a Greek alphabet, α , which means a primitive at the machine level with the coding and compacting in a specific computer system.

```
c2 = (l2, p2, r21, e2)
      = {dynamic, { CS-Dept-FLG80F }, r21, { create,
        delete, change } },
      where r21 (a2, s2, g2) = (true, 2, false).
```

By r₂₁, we know this object is available, with 1 copy, but without copiability.

```
v2 = { "Kelly A. Schneider      ",
        "Ann Schneider          " }.
```

Object <3>: Next one is the object S_S_No,

```
o3 = (d3, a3, r3, c3, v3)
      = (S_S_No, integer(9),  $\Omega$ , dynamic, atomic)
```

where

```
d3 =  $\phi$  .. S_S_No .. (4,4,4) . 1980 . 2
```

```
a3 = integer(9)
```

which takes 9 digit-long integer for the type of its value. Since a social security number is simply an integer which is an atomic value, there is no internal structure for it.

r₃ : Same as r₂.

```
c3 = (l3, p3, r31, e3)
      = {static, { 66506-Soc-Sec-Office, CS-Dept-FLG80F },
        (true, 3, true), { create, delete } }, and
```

```
v3 = 876000321
```

Object <4>: The fourth object is to distinguish the sex of a student, either male or female. Thus, for its prototype,

```
o4 = (d4, a4, r4, c4, v4)
      = (SEX, boolean,  $\Omega$ , static, atomic).
```

```
d4 =  $\phi$  .. SEX .. (40) . 1980
```


a4 = type of boolean,

r4 : Same as r2.

c4 = (static, { CS-Dept-FLG80F }, (true, 1, true),
 { create }).

v4 = 0

Note: Assign 0 (FALSE) for female, and 1 (TRUE) for male.

Object <5>: We need know the pursuing degree of a student at the college. Assign the abbreviation BS, BA, MS, MA, and PD for Bachelor of Science, Bachelor of Arts, Master of Science, Master of Arts, and Ph.D. respectively. The object

o5 = (d5, a5, r5, c5, v5)
 = (KSU.CAND.DEGREE, char_string(2), byte, dynamic,
 atomic),

where

d5 = adm.progm •• KSU.CAND.DEGREE •• (2,0,8) .1980 . 2

in which the user name is an ordered sequence
 of names, KSU.CAND.DEGREE.

a5 = char_string(2),

which is the type of object's value to denote
 the classification of the target degree.
 As to its internal structure and logical
 relationship, since it is a simple object, these
 are not available,

r5 = coding of byte,

c5 = (l5, p5, r51, e5)
 = (dynamic, { Reg-FLG898, CS-Dept-FLG80F },
 (true, 3, true), { create, delete, change }),

v5 = "MS"

Object <6>: The notation for a birthdate could be varying as "October 22, 1957, "10/22/57", "10-22-57", or "102257". Let us take "102257" as six-digit integer to be an example of birthdate.

An object

```
o6 = (d6, a6, r6, c6, v6)
    = (BIRTHDATE, integer(6), Ω, static, atomic)
```

where

```
d6 = φ •• BIRTHDATE •• (122). 1980
a6 = integer(6)
r6 : Same as r2.
c6 = (l6, p6, r61, e6)
    = (static, φ, r61, φ)
    where r61 = (a6, s6, g6) = (true, 1, false)
```

By r61, it indicates that this object is available, with 1 copy, but without copiability.

```
v6 = 102257
```

Object <7>: In a normal case, an address is divided into three parts: street with number, city of state with zip-code, and country. Each of them could be denoted as a character string with necessary length. Assume that now we have a list of addresses of students all living in the U.S.A.. Thus, the value of the nationality part would be static as "U.S.A.". Apparently, the object LOCAL_ADDR, we need, must be a structure type to contain three simple objects of char-string type. Assume a type record is suitable to use for an address. The way to create a structure record for use and its occurrence of LOCAL_ADDR will be explained in Section 3.4. The structure-type object LOCAL_ADDR is

```
o7 = (d7, a7, r7, c7, v7)
    = (LOCAL_ADDR, record, Ω, dynamic, structure)
```

where

```
d7 = φ •• LOCAL_ADDR •• (7, 8, 9) . 1980 . 5
a7 = record
```

in which three elements of character string are needed in an

instance of record, named LOCAL_ADDR,

<1> however, the type of the whole object LOCAL_ADDR is a record.

<2> For a multi-object object as LOCAL_ADDR, its internal structure is three elements for the occurrence, and each of them is a simple object with a type character string.

<3> Its external relationship is that all occurrences of LOCAL_ADDRS have one element, country, mutual with an identical value, "U.S.A.".

r1 : Same as r2.

c1 = (dynamic, ϕ , (true, 3, true), { create,
delete, change })

```

v1 =  +-----+
      | "114 Bluemont Ave.      " |
      | "Manhattan, KS 66502    " |
      | "U.S.A."                |
      +-----+

```

3.4 Examples of Structure

To make a structure available to be used, there are two steps which should be done. The first, a prototype for a structure type must be created; and the next is to create an occurrence of the structure. Take the record and the array as examples of structure type.

Structure: Record

By definitions stated previously, a structure of record and its occurrence are created successively. By Definition 6, a structure is a three tuple (o, p, c) , and

```

record : STRUCTURE (oR, pR, cR)
    oR = { oi : i = 1 to N }
    pR =  $\phi$ 
    cR = { create, delete, change }
    where oi = (di, ai, ri, ci, vi)
           i = 1 to N, as defined below,

```

and, the material needed to construct a record occurrence include seven objects as follows:

```

o1 = (NAME, char-string(25), byte, dynamic, atomic)
o2 = (ALT_NAME, set,  $\Omega$ , dynamic, structure)
o3 = (S_S_No, integer(9),  $\Omega$ , dynamic, atomic)
o4 = (SEX, boolean,  $\Omega$ , static, atomic)
o5 = (KSU.CAND.DEGREE, char-string(2), byte, dynamic, atomic)
o6 = (BIRTHDATE, integer(6),  $\Omega$ , static, atomic)
o7 = (LOCAL_ADDR, record,  $\Omega$ , dynamic, structure)

```

And, by Definition 7, a structure of record is created by defining a prototype tuple (nR, mR, rR, oR, pR) , thus, a structure of record represented by a five-tuple is

```
(record, f1 = o1, f1: a1, { DELETE, ADD, CHANGE },  $\phi$ )
      f2 = o2, f2: a2,
      .
      .
      .
      fN = oN, fN: aN,
```

According to the material needed to invoke an instance of a record, an occurrence of a record, called student record, is built up by Definition 7-A. By the definition, an occurrence of a structure record (nR, mR, rR, oR, pR) is a tuple (d, n, m, o, p) , and the student record would be

```
(STUDENT, record, f1 = o1, { DELETE, ADD },  $\phi$ )
      f2 = o2,
      f3 = o3,
      f4 = o4,
      f5 = o5,
      f6 = o6,
      f7 = o7,
```

For illustrative purposes, the occurrence of record, student record, denoted by the BNF notation is

```
STUDENT : STRUCTURE record (o1,o2,o3,o4,o5,o6,o7)
          USES ( add, delete)
          PARTITIONS (  $\phi$  )
```

And, let any of occurrence the STUDENT record is denoted as ori for the example to use.

For the earlier occurrence of a record of $o7$, LOCAL_ADDR, let us create its occurrence. An occurrence of record, LOCAL_ADDR, is

```
(LOCAL_ADDR, record, f1 = oS, { change },  $\phi$ )
      f2 = oC,
      f3 = oN,
```

where

```
oS = (STREET, char-string(25), byte, dynamic, atomic)
```

```
oC = (CITY, char-string(25), byte, dynamic, atomic)
```

```
oN = (COUNTRY, char-string(6), byte, static, atomic)
```

In the BNF, it is denoted as

```
LOCAL_ADDR : STRUCTURE record ( char-string() )
            USES (change)
            PARTITIONS (  $\phi$  )
```

Structure: Array

It is the same way as the creation of student record to create a new structure array and its occurrence, array of students. In common programming languages, the declaration of array with element type student record, denoted as oR_i , is `ARRAY [1.. MAXSIZE] OF student`. In this model, the structure array can be defined by Definition 7 as follows:

```
(array, array[1] = o1, structure, {DELETE, CREATE, CHANGE},  $\phi$ )
      array[2] = o2,
      array[3] = o3,
      .
      .
      .
      array[AVAIL] = oAVAIL,
```

Assume that there are five student records to be put into an array, and denote them as oR_1 , oR_2 , oR_3 , oR_4 , and oR_5 . By definition 7-A, an occurrence (d, n, m, o, p) of array (nA, mA, rA, oA, pA) , named as `ST_LIST`, is invoked as follows:

```

{ST_LIST, array, array[1] = oR1, { ADD, CHANGE },  $\phi$  )
      array[2] = oR2,
      array[3] = oR3,
      array[4] = oR4,
      array[5] = oR5,

```

An alternative denotation in the BNF is

```

ST_LIST : STRUCTURE array ( STUDENT )
        USES ( add, change )
        PARTITIONS (  $\phi$  )

```

3.5-Control of Computation: Definition and Example

A data object is what drives the computation. The execution is basically "driven" by the existence of "material" objects on which actions are to be affected. The effect is to produce "resultant" objects which may be used by further action. The action is "detailed" by requests for actions, and such requests "construct" the action. Conditions of "stimulation" and "termination" are to control requests.

Requests

A request is a statement indicating an action to be performed. This statement specifies that certain material objects are to apply the action to produce other resultant objects.

Definition 3.5-1 An action is an object whose value consists of requests for more detailed actions—possibly having values in the hardware of the computational environment.

Definition 3.5-2 A simple action is denoted by a three-tuple (m, a, r) , where

m is the material required, expressed

as a list of objects,
 a is the designator of the action,
 r is the result produced by the action,
 expressed as a list of objects.

Definition 3.5-3 A simple request for an action to be performed on material producing results is denoted by a three-tuple (m, a, r), where

m is the material to be used in the process, expressed as a list of objects,
 a is the designator of the action,
 r is the result of the process, expressed as a list of objects.

In the BNF form, the simple request is

```
<simple request> ::= <a> (<r>; <m>)
<a> ::= <name>
<r> ::= (<name> (, <name>)* ) *
<m> ::= (<name> (, <name>)* ) *.
```

To illustrate the use of a request, let us take an action, designated with diffvar, to calculate the result y with the second-order linear differential equation

$$(d^2y/dx^2) - (a + (1/b)) (dy/dx) + (a/b) y = -(ac/b)$$

A function $y = f(x) = -c + c1 * \exp(ax) + c2 * \exp(x/b)$ with given arbitrary real numbers c1 and c2 can give us the y value. A request for the action, diffvar, is diffvar (y; x, a, b, c, d, c1, c2) with the material list (x, a, b, c, d, c1, c2) and the resultant list (y).

Ordering

Within the model, the ordering of execution among requests is governed by the principle of data drive. A request is eligible for execution once with the availability of the material. A request can be initiated even though not all materials are available. When its material is available, a request can start execution.

Definition 3.5-4 Data drive is the principle for the ordering of execution among simple request.

- A request is eligible for execution if
- (1) the action is available,
 - (2) each requisite data object in the material is complete and available, and
 - (3) the user is authorized to use each object involved.

Take five requests as examples:

multiply (sum1; e1, c)

exp (e1; ax)

divide (bx; b, x)

multiply (ax; a, x)

and assume the objects in requests are available and complete, including a, b, c, x, exp, multiply, and divide, and the user has authorization to use them. By the principle of data drive, the ordering of execution is that the multiply to produce ax and the divide to produce bx are immediately eligible for execution. Next, the exp gets executed after the multiply produces the resultant object, ax. Once both e1 and c are available, the multiply to produce sum1, then, can be executed.

The ordering in the example, action diffvar, can be represented in graphical form with lucidity. In Figure 3.5-1, the circular

nodes represent material objects or resultant ones of the requests. The rectangular nodes represent the actions to be performed. There are two kinds of arcs: one from an action node to a resultant object, the other from a material object to an action node. Therefore, through a request, an action is linked with its material and requests.

A problem solver is allowed to specify the ordering of a set of requests by the use of the data drive principle and a choice of object designators. This process is called sequencing.

Definition 3.5-5 The sequencing of two requests can be determined by the problem solver through the appropriate selection of material and result object names. Given two requests $(m1, a1, r1)$ and $(m2, a2, r2)$, if $m1 \wedge r2 = \phi$ and $m2 \wedge r1 \neq \phi$.

For example, two requests,

TEST (A, B, C; X, Y, Z)

CHANGE (P, Q ; S,T)

do not have an explicit execution order. If it is desired to execute TEST first, an object can be created to produce

$$\begin{array}{ccc} m & \wedge & r \\ \text{TEST} & & \text{CHANGE} \end{array} \neq \phi .$$

Choose an arbitrary object, called TEMP. Two requests with the purpose are shown below:

TEST (A, B, C; TEMP, X, Y, Z)

CHANGE (P, Q, TEMP; S, T).

The request CHANGE cannot start execution until TEST produces TEMP. The purpose of sequencing is accomplished.

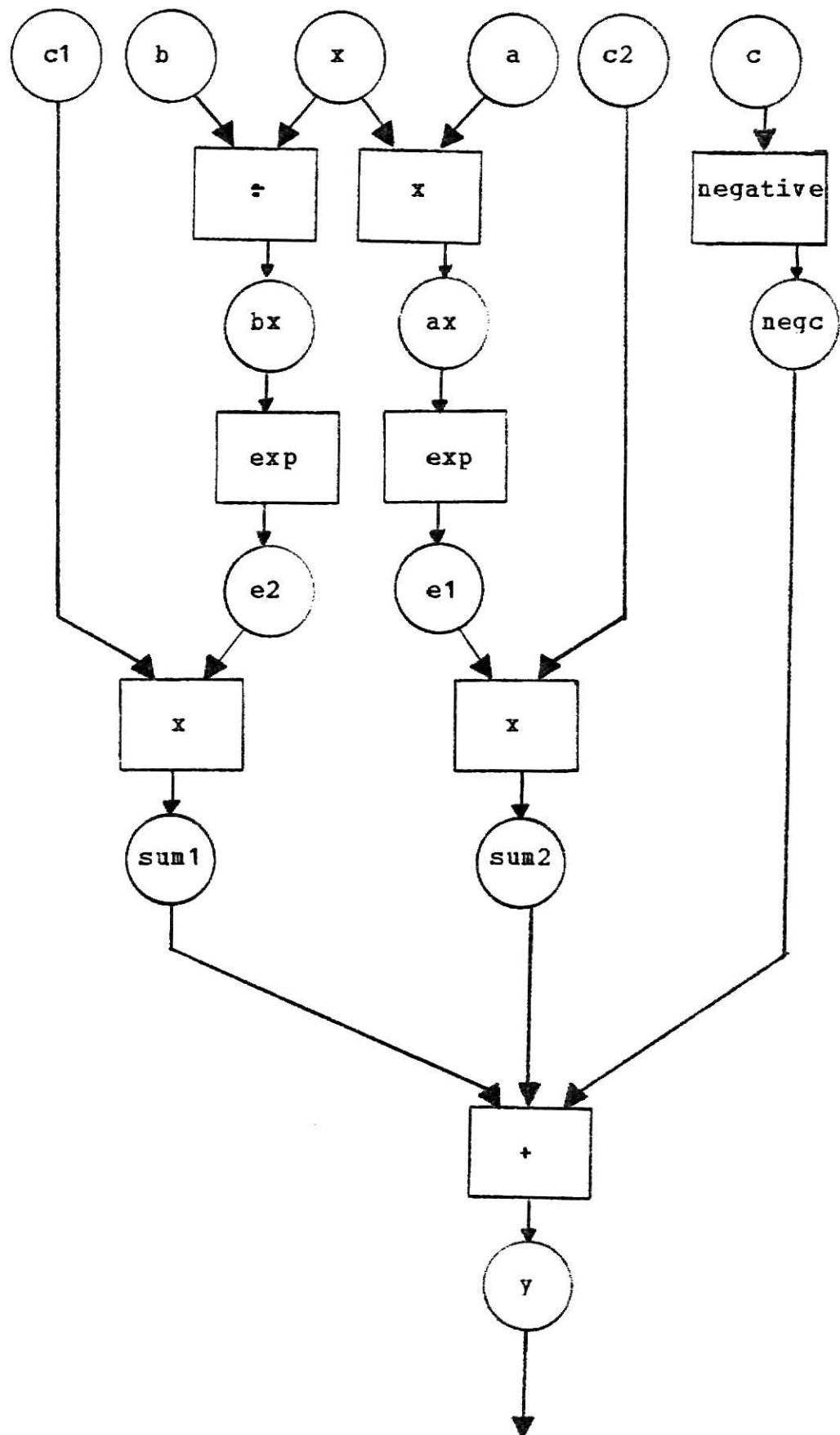


Figure 3.5-1 Ordering of An Action, diffvar.

The order of appearances of requests with a common local context has no influence upon the order of execution.

Detailing

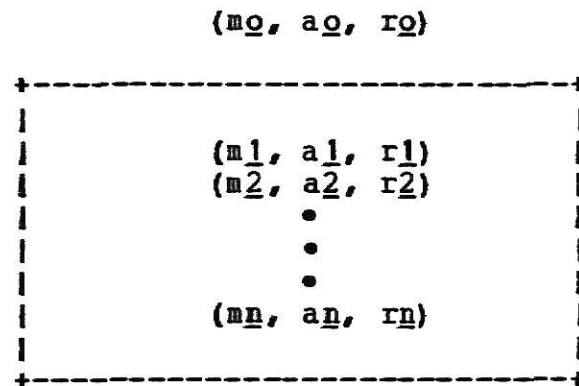
There is a way to define an action with material available to produce results, called detailing. This process expresses an action in detail as more computational particulars of a set of requests. This set of requests providing the computational particulars of an action is called a request set.

If a simple action ($\underline{m_0}$, $\underline{a_0}$, $\underline{r_0}$) is detailed by a set of n requests ($\underline{m_i}$, $\underline{a_i}$, $\underline{r_i}$), $i = 1, 2, 3, \dots, n$, then

$$\underline{m_0} = \bigcup_{i=1}^n \underline{m_i}$$

$$\underline{r_0} = \bigcup_{i=1}^n \underline{r_i}$$

Thus, the detail of a simple action ($\underline{m_0}$, $\underline{a_0}$, $\underline{r_0}$) is denoted as follows:



The request set, $(\underline{m_i}, \underline{a_i}, \underline{r_i})$ for an action $(\underline{m_0}, \underline{a_0}, \underline{r_0})$ is at one level lower in the hierarchy of requests than the request $(\underline{m_0}, \underline{a_0}, \underline{r_0})$.

Construction is the reverse of the detailing. It combines a set of requests into an action to represent the set. Thus,

$$\underline{m_0} = \bigcup_{i=1}^n \underline{m_i} - \bigcup_{i=1}^n \underline{r_i}.$$

In the hierarchy of requests, now, the constructed action is said to be one level higher.

The construction of an action, diffvar, is pictured in Figure 3.5-2. Its execution order was given in Figure 3.5-1.

diffvar (y; x, a, b, c, d, c1, c2)	
multiply (ax; a, x)	(ax \leftarrow a \cdot x)
divide (bx; x, b)	(bx \leftarrow x/b)
negative (negc; c)	(negc \leftarrow -c)
exp (e1; ax)	(e1 \leftarrow e ^{ax})
exp (e2; bx)	(e2 \leftarrow e ^{x/b})
multiply (sum1; e1, c1)	(sum1 \leftarrow c1 \cdot e ^{ax})
multiply (sum2; e2, c2)	(sum2 \leftarrow c2 \cdot e ^{x/b})
add (y; sum1, sum2, negc)	(y \leftarrow -c + c1 \cdot e ^{ax} + c2 \cdot e ^{x/b})

Figure 3.5-2 A Construction of Diffvar from Requests for Other Actions.

Conditions

The execution of an action is dependent on four conditions, which may be null for some of them. Two of them govern the eligibility for execution, called stimulations, appended to a request and an action respectively. The other two are terminations appended to a request and an action to control their termination of a request. A condition appended to a request is an external condition, and that to an action is an internal condition.

Definition 3.5-6 A condition is a boolean expression which may involve objects, literals, and logical and/or relational operators.

Definition 3.5-7 An action is denoted by a five-tuple (s_i, m, a, r, t_i) where m , a , and r are as defined in Definition 3.5-2 and

s_i is an internal stimulation, if
 $\text{null } s_i = \text{"true"}$, and

t_i is an internal termination, if
 $\text{null } t_i = \text{"false"}$.

Definition 3.5-8 A request is denoted by a five-tuple (s_e, m, a, r, t_e) where m , a , and r are previously defined in Definition 3.5-3, and

s_e is an external stimulation, if
 $\text{null } s_e = \text{"true"}$, and

t_e is an external termination, if
 $\text{null } t_e = \text{"false"}$.

Definition 3.5-9 The effect of a request is a five-tuple (s, m, a, r, t) where m , a , and r are previously defined, and

$s = s_e \wedge s_i$,

and

$t = t_e \vee t_i$.

The effect of a request (s, m, a, r, t) is eligible for execution when s is "true" and t is "false", based on the data drive principle. A request is terminated when $(t_e \vee t_i)$ is "true".

A user in a request may wish to override the internal stimulation and termination of his external conditions. Thus, the

conditions could be reduced from s to s_e and from t to t_e .

In the BNF form, the conditions are denoted by:

```

<stimulation> :: = <condition> ;
<termination> :: = ; <condition>
<condition> :: = [ <boolean expression> ]
<boolean expression> :: = "any well-formed
                           boolean expression".

```

An example of the use of stimulation is given. In Figure 3.5-3, an action, DIVIDE, has been designated with a denominator x . An internal stimulation for DIVIDE is " $x \neq 0$ and x is a real number". If the user wishes the request to be eligible for execution only with an integer denominator and after 7 pm, then the external stimulation is " $x \in \text{integer} \wedge \text{time} > 19$ ".

```

action : [  $x \neq 0 \wedge x \in \text{real}$  ] : divide (z; x, y)
request : [  $x \in \text{integer} \wedge \text{time} > 19$  ] : divide (z; x, y)
effect : [  $x \neq 0 \wedge x \in \text{real} \wedge x \in \text{integer} \wedge$ 
           $\text{time} > 19$  ] : divide (z; x, y)

```

Figure 3.5-3 Examples of Stimulation.

The next example is one of external termination. The action ADD and SUBTRACT have termination conditions, $\overline{(\text{sum})\delta}$ and $\overline{(\text{difference})\delta}$, to cause their requests to terminate when a value of 'sum' and that of 'difference' are available.

ADD(sum; a, b) : [$\overline{(\text{sum})\delta}$]

SUBTRACT(difference; a, b) : [$\overline{(\text{difference})\delta}$]

In this chapter, simple objects and multi-object structures have been used to represent information involved in examples of students' data. However, there is no methodology for creating structure by a form within the abstract model for aggregating objects into a structure yet. In the next chapter, a form will be developed by extracting the concept of cluster in Liskov's Abstraction Data Type.

CHAPTER 4

PROBLEMS AND EVALUATION OF THE UNGER MODEL

This chapter defines the problems with the Unger model and evaluates its strengths and deficiencies.

4.1 Strengths of the Unger model

The model puts its strengths in data abstraction, control construct abstraction, concurrency, and the structured development of problem solutions. The abstractions of data objects and of control constructs are employed to present to the user. The model attempts to permit use of abstractions without being concerned with how they are implemented. Structured programming enhances the reliability and understandability of programs. To deal with the complexity of problem solving, structured programming, abstraction, and concurrency can assist the user to accomplish it.

In this section, the strengths of the model are stated in four categories: (1) data abstraction, (2) control and structured programming, (3) concurrency and comparison with conventional programming languages, and (4) others.

4.1.1 Data Abstraction

The abstraction of data provides the user with a capability of defining data structures. Data are abstracted in a form whose

properties are known to the user but whose implementation details are unknown. To encounter more complex cases of problem solving, a more expressive structure to abstract data is provided in the model.

1. The quality of a specification model is largely based on the specification unit being specified [11]. If a specification unit is too small, e.g., a person's name, then it cannot respond as a useful concept. What is desired is a specification which is big enough to correspond a thing in the reality to an abstraction useful in problem solving.

All information involved in the solution of problems is represented in the model through the use of objects. An object in the model could be an algorithm, input data, or intermediate and final results. Specifically, it can be a constant, a variable, a collection, an action (i.e., a process), or a comprehensive structure. While a specification can work higher to an entity, such as a file, much of the extraneous detail can be eliminated.

2. Structures, formed from one or more objects, are used in the model to represent information. It represents information in one encompassing structure, and encourages structured programming development. Through "objects" and "structure" of objects, properties of data useful in a concurrent programming environment are provided. See also Section 4.1.3.

By the process of aggregation, a structure is created by combining one or more objects. As in Liskov's data abstraction, the components of the objects composing the new structure are not

available to the user, once the aggregation is defined. This property not only supports structured programming in which relevant detail is emphasized and irrelevant detail suppressed, it also increases the portability of implementation of the model. With development of structures, the modelling of information is clearer.

3. Through the designator of an object, the naming convention, the unique sequence and environmental identification is provided. The chronological identities and sequence position specified by the user indicate the time and order of object creation. The model provides an instance for sequencing incarnations, and offers a context for structured system identification.

4. Through the use of the longevity feature, the model protects certain kinds of data from change and access, such as fixed and static objects. For an object with dynamic longevity, the model also provides an instance to permit the system to record its history by keeping incarnations. Once a need for retrieving a specified incarnation of a dynamic object occurs, a mechanism is available to trace it.

5. The model provides a mechanism to record the number of identical copies of an object in the corporality component. Local storage of data objects is encouraged in the model, but, if not copiable, the objects can offer a very restricted form of global storage.

4.1.2 Control and Structured Programming

The purpose of structured programming is to enhance reliability and understandability. A discipline that a problem is solved by means of a process of successive decomposition is imposed in structured programming. The problem-solver is concerned with proving that his solution correctly solved the problem. What concerns him/her is the way his/her solution makes use of the abstractions, but not any details of how those abstractions may be realized (i.e., implemented in the next lower level of the hierarchy). When he is satisfied with the correctness of his solution, he turns his attention to the abstractions it uses [9]. Thus, structured programming is called as a problem solving technique based on abstraction. In the model, structured programming developments have provided mechanisms for the abstraction of data objects as well as of control constructs and of action constructs. Structured programming technique and data abstraction concept, thus, can assist the users to deal with the complexity of problem solving.

1. The model's action, detailing and aggregation (see Detailing in Section 3.5), supports a structured and systematic development of problem solutions. Detailing is a process involving the replacement of an action by a set of actions expressing more detail. Construction is a reverse process of detailing (i.e., combining a set of requests into an action to represent the request set). In the hierarchy of requests, the actions detailed in a request set for an action are said to be at one level lower than the request (of the action). Conversely, the

action constructed from its request set is one level higher within the hierarchy.

2. Modularization of action as a control construct is supported by the structured programming. At an arbitrary level, an action is detailed into actions which are known. At each level of the detailing process, the sequencing of the actions is basically data driven and further controlled by explicitly stated conditions.

3. "Structures" in the model, which are formed from one or more objects, are obviously more complex than conventional representation models. It is used to represent information in one encompassing structure, and also encourages structured programming development. In addition, the model can represent much computational information within the "structures".

Any statement consists of nouns and verbs. In the model, the nouns are names of objects and of structures; the verbs are the requests which describe an action. Let actions be termed as "dynamic" data and structures as "static" data. The critical advantage in the model is that static information is not described in the dynamic (action) environment, but it is to be transferred into the static (structure) environment. Such transference results in two advantages:

- (1) the dynamic environment of actions, which the user must understand, is simplified, and
- (2) the static environment of structures is expanded so as to assimilate more programming requirements, but still with ease of understandability.

4. Even though all material of an action is not available, the action can still be initiated in a controlled and non-destructive way. Thus, in the model, the requirement that data exist before execution is relaxed. By utilizing the concept of partialing, any specified computation is allowed to proceed until the lack of at least one material object causes it to be suspended temporarily. As the requisite material becomes available, the execution would be resumed from the point of suspension. This indicates the property of concurrent process in the model, and to be discussed in the next category.

Partialing is a construct allowing for partial computation on incomplete material. It allows an action to become eligible for activation before all of its material exists. For example, this construct would be helpful in a production environment where subcomponents of the result can be computed from some of the material. The material not necessary for eligibility consideration for the action must be indicated in the model with an underscore. Take an action, TOTAL, specifying partialing as an example,

```
total (check_no, amount; pay, idcard, timecard, hrrate)
```

allows its requests to start execution with requisite material: "pay", "idcard", and "hrrate", before "timecard" is available (indicated by the underlining). Give another example of a request which prohibits partialing as follows:

```
total (check_no, amount; pay, idcard, timecard, hrrate)
```

5. The execution control of an action is dependent upon conditions of stimulations and terminations. An advantage of

conditions defined in the model is the flexibility in the expression of boolean conditions. As previously defined in Section 3.5, a condition could be "any well-formed boolean expression" which may involve objects, literals, and logical and/or relational operators.

6. The model provides a communication mechanism in the form of the action. Through the use of parameters, communication between the invoked and invoking environment is accomplished to avoid the undesirable side effects resulting from the scope of name rules in conventional languages.

An example of a piece written in PASCAL is given to illustrate the problem occurring from the scope of the binding. Consider invocation

square(t)

in the context of

```

var t : integer;
function sqr (x : integer) : integer;
begin  sqr := ... end;
  .
  .
  .
var t : integer;
begin
  t := sqr(t);
  t := t
end

```

which is unsatisfactory for two reasons:

- (1) There is a conflict between the occurrence of 't' in the actual parameter and local identifier 't' of the procedure definition body into which the actual parameter is being substituted.
- (2) There is a conflict between the occurrence of 'sqr' in the procedure definition body and local identifier 'sqr' of the context into which the body is being

substituted [17].

By the way of communication in the Unger model, let us see how such a problem can be avoided. For the parameters, the material and the results, used during communication, the material of requests are normally copied into local incarnations of the material, and the results are used to create local incarnations. Most communication of values occurs during stimulation and termination [18]. If the objects used as actual parameters are not available, then two situations might occur. First, if such an unavailable object is used as material, it cannot be changed until the request is terminated. Second, an object is being used as a resultant object, the request cannot be initiated until all references to that incarnation are completed [1].

The input (material) and output (result) correspond to actual parameter and local parameter during the invocation in the example written in PASCAL. Since there is a unique name for each object in the model, this problem will not occur. The example of "square" could be expressed in the model as an action `square(newt ; t)` with unique names for 'square', 'newt', and 't'.

4.1.3 Concurrency

A language which allows the expression of concurrent problem solutions enhances a user's ability in problem-solving [18]. Under the influence of conventional programming languages, many programmers do not distinguish sequential computation from

concurrent computation. A model possessing intrinsic concurrency can provide an environment for users' thoughts to enhance efficiency of the production.

1. The model defined and developed is intrinsically concurrent, and, thus, it can enhance the creation of programs which utilize available hardware concurrency. Also, it may allow the production of programs with increased efficiency.

The feature of intrinsic concurrency is regulated by the control structure with data-drive principle. In the model, (1) the allowance of execution suspension, caused by incomplete material, and (2) the initiation of an action without the requirement of complete material permits more than one action, whether eligible or not, to be initiated concurrently guided by the conditions with data-drive control. This is same idea to process synchronization as the monitor. Based on a mechanism known as condition, a process can suspend ("delayed") itself and wait until the condition is resumed ("signaled") with execution eligibility by some other process. For a sleeping process, it will continue execution from the point it suspended itself once becoming eligible [17]. Intrinsically, the feature of concurrency exists in the Unger model.

2. Although a conditioned, data-drive, intrinsically concurrent model replaces the underlying sequencing mode for conventional languages, sequential computation can be specified in the model. A problem-solver can use sequencing, previously defined in Section 3.5, by linearizing objects to control an action sequentially.

3. In order to deal with complex system involving concurrency, a model which supports a concurrent thought process is encouraged. Since many problems encountered have solutions involving concurrency, to allow expression of these solutions in an intrinsically concurrent model could be more efficient and clearer.

4. Conventional languages that current programmers use do not have the recognition of concurrency inherent in a problem solution. Therefore, programmers do not always recognize such a property. The model provides a syntactic and semantic basis for concurrency to be expressed, therefore the recognition can be greatly encouraged.

4.1.4 Others

1. The major importance of the model lies in its simplicity. The model is uniform accross a span of uses, so the user just need learn one set of syntactic and semantic rules. The ease to express common computations is what the user desirably expects.

2. The model is described by a helpful notation. To effectively develop and carefully evaluate the properties of any model of computation, a precise notational system is required. In the model, a mathematical notation employing such basic concepts as sets, tuples, mappings, and an extension of Backus-Naur Form was found most useful to be adopted [18].

The model does provide strengths lacking in the common constructs of conventional model. Let us list the strengths of

the model as follows:

- Concurrent computations are intrinsically represented.
- One encompassing data structure is used.
- Conditional computation is supported.
- Common computation is expressed in a natural form.
- The structured programming development is encouraged.

4.2 Problems with the Unger Model

In this section, several questions invoked during the study of the Unger model are defined. On each question, different viewpoints and ideas relative to the problem are proposed, but the discussion of these will be detailed in Chapter 5. Not all questions stated in the following are directly carved by the problems unsolved in the model. However, a couple of them are subjects existing in the literature of data base still in dispute, such as Problem 1 and 4.

Problem 1: What is "data" in a data model?

The dominant importance of data in the information system world is reflected in the attention it receives particularly in database. Over the past decade, a topic of research activity in programming language has been to explore the issues related to data abstraction (i.e., abstract data types). However, it is a fact that the notions about or meaning of data are still not clearly understood. In the database world, much of the work has concentrated on it.

A number of viewpoints have been presented in the literature of

database. First, Mealy gives a new view at data in his paper, "Another Look at Data" [12], and he views "data maps" as what we mean data in a data model. Second, Unger does define what is the "data" existing in the model. In her model, data are represented by objects and structures, and an existence function is used to determine the existence of data. Third, Kent states that "data in a system is what can be extracted rather than what is physically stored" in "Data and Reality" [6]. The last view about data is from Sundgren's Theory of Data Base [16]. He defines data as "the arrangements intentionally made by a person through a medium". The concept of "existence" is also mentioned by these authors. Among these viewpoints, there is a similar point.

Problem 2: There is no methodology for creating structures in the Unger model.

In the Unger model, the body of a structure and the concept of aggregation has been completed, but a "form" to aggregate composing objects into a structure is not defined. In order to sketch out a "form" to accomplish it, the ideas in a couple of helpful papers would be referred.

Beside keeping a portion of a structure from the Unger model, I will extract additional viewpoints from the Liskov's "Abstract Data Type" [9] and the algebraic specification. In Liskov's paper, a construct is actually provided to create a structure type, and it is called "operation cluster". There are four parts in a cluster. Not all four parts are encouraged to be adopted, but some good points from the algebraic specification replace the undesired points in the Liskov's cluster. How the actual "form"

is to be created and the reason to adopt a number of points from these papers will be detailed in next chapter.

Problem 3: Should an object which is a structure be allowed to have a corporality?

An object is complete only if its components (d,a,r,c,v) exist in the model, and the longevity of the corporality is the focus on which we are going to work. Questions are raised: (1) does a structure have longevity, or (2) do only the atomic elements have longevity? If a structure does, a principle or a rule is needed to make the whole body of a structure exist in harmony with its elements. The advantages and disadvantages to have a structure or/and its elements with longevity would be discussed too.

Problem 4: How to avoid the inconsistency on the same piece of data represented in different systems?

As data exist in more than one system or model or data base for sharing uses, it is expected that they are stored in a way with same value. The inconsistency of the data in two or many systems is to be avoided. For any two systems, we wish an object in one system to have a unique object in the other. In addition, we may want to make sure that anything happening to the object in the first system also does in the second. To proceed with this question, some ideas from Mealy's work are used. How do we deal with inconsistency once it occurs? A rule is given for solving the problem, but it may not be an idealistic solution. We expect as the Chinese proverb that "To throw a brick today and to get a gem in return tomorrow" from the readers to proceed in the future

work.

Problem 5: There is no means to indicate as primitive object.
An object, based on a primitive, is denoted by
a Greek omega, Ω .

The Greek letter, Ω , stands for a primitive at the machine level with coding and compacting in a specific computer system. The separation of consideration from the logical aspect from the physical aspect has been emphasized in the data abstraction and in data independence of data base literature. The representation of data, including the format, coding, location, and access methods, is not encouraged to be what the user pays attention.

In this chapter, the strengths of the Unger model are stated, and five problems are defined. For these five questions, not all of them are raised due to a deficiency of the model. The insights into the solutions to some problems will be given in Chapter 5.

CHAPTER 5

INSIGHTS INTO SOME SOLUTIONS TO THE PROBLEMS

Five problems were defined in Chapter 4. In this chapter, for each problem, one or more points of different views are presented. Even if some of the problems cannot be solved in this paper, it is expected that the contents covered in the chapter will help the readers with future work. Several rules are proposed to attempt to solve some of the problems.

5.1 Problem 1: What is "data" in a data model?

A data model describes not only the organization of data, but also the operations to be made upon the data. We expect such a model to establish certain correspondences between constructs inside the data model and things in the real world. Ideally, it would be a one-to-one correspondence.

What, then, is data which is modelled? To begin with the discussion on this question, let us refer the viewpoints presented in the literature of data base area. One is from Mealy's "Another Look at Data" [12] which was taken as a prologue in this paper. The other three are from Unger's paper [18], from Kent's "Data and Reality" [6], and from Sundgren's Theory of Data Bases [16].

Mealy called data as "fragments of a theory of the real world". According to the nature of the theory, data are supposed to record a set of facts about some set of entities, either real or

abstract [12]. With choices of data maps, entities, and values, data maps are regarded as data in Mealy's view.

The constructs inside a data model are not necessarily an atomic value mapped from an argument of an entity. That is why the data map in Mealy's paper admits structural data maps, which have the entity set as a subset of the value set. There are a set of entities, E , a set of values, V , and a data-map set, D , whose members are maps of the form:

$$\theta : E \rightarrow V$$

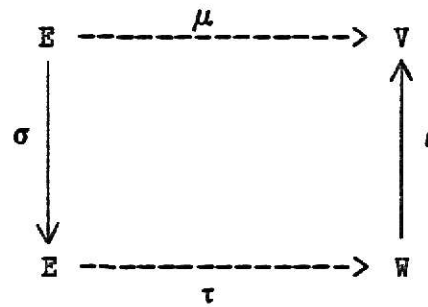
which are restricted to be functions (or single-valued maps). The structural data map is defined as of the form:

$$\sigma : E_1 \rightarrow E_2.$$

The value set, E_2 , might be considered to be constructed from other sets. It is implicitly assumed that E_2 is a subset of V to admit structural data maps. The set of elements of V exclusive of elements of E_2 is

$$W = V - E_2.$$

Thus, all data maps can be defined in terms of a non-structural map τ applied to a structural map σ . In the following diagram, let μ be an arbitrary data map and ι be the identity map of W into V (to point merely out that W is a subset of V). A fact shown in the diagram is that, to start with an element of E , the same value can be obtained by applying the map μ as by applying σ , τ , and ι in order.



Why are data maps instead of their elements regarded as data? In Mealy's viewpoints, neither values nor entities can be redefined or changed during data processing. For example, the value set cannot be redefined, if it is the integers. What actually happens, while data are processed, is actions on data maps. Suppose that $\mu(e) = v$. The ordered pair (e, v) is a member of the set μ . To redefine the value of the data item (e, v) is to redefine μ by removing that pair from the data map set μ , and then adding a new pair (e, v') .

Toward the question of what exists in a system, Mealy discards dispute of "existence with perceiving". In the literature of database, some have claimed that the existence of things is independent of whether anyone perceived them properly or not; some claimed that there is no existence without being perceived. Mealy thinks that a data base never records all of the facts about a group of entities; a fact may be recorded with complete or less accuracy. Data do not necessarily represent facts with utter accuracy. The subject of existence of data in a model is covered in Unger's and Kent's too.

Kent said that data in a model is that which can be extracted, rather than that which is physically stored. Some information is actually described to the model, e.g., the names of the students

in a department. However, some have to be deduced through computation or from the other actually existing data, e.g., the total of students in the department. Some examples of deduced data are given as follows:

- (1) Manually, the number of all students can be counted simply asking for the names.
- (2) Declarations for the information system define a count field as a part of the data needed by your application. The field is never stored. When an application retrieves a department record, the system counts the number of students and inserts the count into the record deduced for the application.
- (3) If there is an interface with a query processor, one can ask it how many students are in the department. The query processor extracts the list of names, and counts the total number for the answer.

Definitely, the first example has a security problem. Security is not considered at this point.

As stated by Kent, a model is said to be a set of named functions. Each function is capable of returning a certain value when invoked with a certain argument. After an update is done to modify the function, it will subsequently return a different value for the previous argument. The implementation of the functions is transparent to the user, and it might involve simple access to stored data, complex traversals of data structures (e.g., pointers), or computations. Thus, the data content of the data

model is defined by this set of functions, rather than in terms of physically stored data.

Unger models information in "data object" or "structure" of objects. The value of an object represents the informational unit for which the entire object was created. It might be an atomic value like a number, a complex of values obtained through the use of spatial coordinates, a collection of objects (structure) like a file, or the detail of an action, compatible with the attribute.

In terms of single-valued mappings, five components (d,a,r,c,v) of an object are expressed. An existence function having a boolean value is used to determine whether a component has been bound. An object is complete in the model after all components are bound with the true value of the existence function for all components. In most instances, the binding can be transparent to the user since it can be automatically supplied by the system.

Sundgren defines "data" as the "arrangement made intentionally by a person". The complete definition is extracted from his work [16] as follows:

Definition 4-1. If a person intentionally arranges one piece of reality to represent another, we shall call the former arrangement data, we shall say that the arranged piece of reality is a medium, which is used for storing the data.

In most cases, data represent primarily a person's knowledge about reality rather than the medium for storing the data. On the

basis of the person's perceived knowledge, the data registration is such as instant automatic cameras, max/min thermometers, recorders, and etc. So, the property of being data need not depend on the cooperation of a human mind in the registration process, but it is decided by the arrangement intentionally made for the registration. The stored data are called the data contents of the accessible memory. If stored data exist without accesses or processes upon it, then, the data content would not be of much use.

Although different authors have their own words to define what data are, they do try to convey a mutual point, explicitly or implicitly. Data are what can be extracted from a system rather than what are physically stored. Thus, the dimension of data is expanded to include the intermediate or final result through a computation, a procedure, an operation, an algorithm, a data traversal, a pointer, or a data value obtained from the spatial coordinate address. As to Mealy's view, even he puts emphasis on "data maps" instead of attributes and values, however, it still conveys the same idea. The value mapped from the attribute might not be actually stored but be able to be reached through a certain kind of function (i.e., mapping). As to the Unger model, the idea is not explicitly stated, but the encompassing data structure of "objects", "structures", and "actions" implicitly supports the idea. A data object might be an atomic value, an algorithm, a comprehensive value obtained through a spatial coordinate, a structure, or action(s). As to Sundgren's definition of data, it emphasizes the different categories of data encountered in the real world, such as tape, photograph, slide, or temperature.

However, Sundgren does state that data is not of much use if there are no processes or accesses upon it. For a piece of data, if it is just stored in a system as a piece of junk in an old, old grandma's basement without being extracted for use, then, it is even not necessary to be stored as data. This viewpoint is a little different from the Kent's idea. With the statement by Kent, the conclusion on "what data is" is made as "data is with what can be extracted rather than with what is physically stored in the system".

5.2 Problem 2: There is no methodology for creating structures.

In the Unger model, there is no overt methodology for creating structures, so a form for aggregating data objects into a structure is created.

In Liskov's paper, the data abstraction comprises a group of related functions or operations that act upon a particular class of objects with the constraint that the behaviour of the objects can be observed only by applications of these operations. A new linguistic construct, the operation cluster, was introduced to provide programming language support for abstract data types. The cluster contains programming code which implements each of the characterizing operations of a data type (i.e., structure in the Unger model), called operation definition.

The cluster contains four parts: cluster interface, object representation, object creation, and operations. See Figure 2.2 in Chapter 2 for the example of a cluster. The cluster interface

defines the name of the cluster, the parameters required to create an instance of the cluster, and a list of the operations defining the type, e.g.,

```
array: cluster (element_type : type)
      is delete, add, change
```

Comparing with structure prototype in the Unger model, the contents covered in both methodologies are mutual. Something lacked in the Unger model is a "form" to fulfill the aggregating activity. Reviewing the detail of the remaining three parts of the cluster, it is helpful to render some of them into the model.

Whereas, the algebraic specification with more expressive power on the operations' effects by axioms is desirable to be referred rather than the code-oriented operation definitions. As stated in detail in Chapter 2, this approach is strongly recommended for several reasons. In the DECLARE portion, the operations allowed to manipulate a data type are single-valued mappings without side effects. It uses the axioms within the FOR-ALL-END block to describe the semantics of the operations. With the recursion, these axioms could be interpreted easier than as they look. See several examples from Figure 2.3 to 2.6 in Chapter 2 for illustration of this technique.

To illustrate the preference toward the algebraic specification over the coding-style operation definition in Liskov's, two examples of the stack, in Figure 2.2 and 2.3 by different methodologies, are used again. Consider the operations PUSH and POP which are single-valued mapping,

```
PUSH(stack, item) ---> stack
```

POP(stack) ---> stack

define the output value, stack object, in terms of the input values, an integer and/or a stack object. The object representation in the cluster is

```
rep (type_param : type) = (tp : integer;
                             stk : array[1..] of
                             type_param;)
```

Written in PASCAL, a typical stack structure might be

```
type stack = record
    top : integer;
    data : array[1..100] of integer
end
```

and then the meaning of

```
t := PUSH(s, i)
```

could be stated in Liskov's operation definition as follows:

```
push: OPERATION (s : REP, v: s.e_type);

    s.tp := s.tp + 1;
    s.stk[s.tp] := v;
    RETURN;

END
```

A critic is that the operation definition does not describe the concept of stacklike behaviour, but specifies much of the extraneous detail. For example, the concept that POP returns the

value most recently pushed on the stack can only be inferred from this detail. This detail of the operation definition is undesirable for two reasons [11]. First, the inventor of the concept must get involved in the detail which is really implementation information, rather than stating the concept directly. Second, the definitions of PUSH and POP are not independent. A change in the definition of one is almost certain to lead to a change in that of the other. In addition to being related through the structure for stack object, the definitions of PUSH and POP are expected to be related in the interpretation of the structure. Instead of pointing to the first available slot, the selector "top" points to the topmost piece of data in the stack.

It is not necessary to describe the individual operation separately. Instead, the effects of the operations can be described in terms of one another, as axioms describe the semantics of operations in the algebraic specification. The effect of POP might be defined in terms of PUSH by

$$\text{POP}(\text{PUSH}(s, i)) = s$$

which states that POP returns the item most recently pushed.

The prospective form would adopt the Liskov's object interface, and object representation, and also the axioms of operations in algebraic specification is preferred to replace the operation definition and object creation to describe the semantics of the operations.

By the BNF notation, a form to aggregate objects into a structure is given as follows, called NEW FORM:


```

< struct_name >   :  structure { <name> :   type }
                    is  <op_des>*
                    rep  { <name> : type } = { <type_definition>+ }
                    operation
                        <operation>*
                    effect <title>?
                        <effect>*

end.

<op_des>   :: = <name>  _
<type_definition>  :: = <name>  : <type>  ;
<type>       :: = boolean ! integer ! real ! <struct_type>
<operation>  :: = <op_name> { <inner_op> } --> <value_op>
<value_op>   :: = <boolean> ! <struct_name> ! <name> ! {undefined}
<inner_op>   :: = <struct_name> ! <null> ! <name>
<effect>     :: = <op_name> { <param> } = <value>
<param>      :: = <effect> ! <name> <names>
<names>      :: =  _ <names>+
<value>      :: = <boolean> ! <struct_name> ! <name> ! undefined
<boolean>    :: = true ! false
<title>      :: = { for_all <assign>+ }
<assign>     :: = <name> <--> <name>  _
<symbol_string>? :: = "<symbol_string> can occur 0 or 1 times."

```

Figure 5.1 NEW FORM for Structure's Aggregation.

Let us create an instance of a structure based on the "form"

above. Previously, an example of a stack had been taken in many approaches, and it is taken as an example for illustration in Figure 5.2.

```

Stack : structure ( material_type : type )
      is CREATE, ISNEWSTACK, TOP, PUSH, POP

rep ( material : type ) = (max : integer;
                           field : array[ 1..max]
                           of material;)

operation
  CREATE () ---> stack
  ISNEWSTACK(stack) ---> boolean
  TOP(stack) ---> material U {undefined}
  PUSH(stack) ---> stack
  POP(stack) ---> stack

effect (for all s <-- stack, m <-- material)
  ISNEWSTACK(CREATE) = true
  ISNEWSTACK(PUSH(s,m)) = false
  POP(CREATE) = CREATE
  POP(PUSH(s,m)) = s
  TOP(CREATE) = undefined
  TOP(PUSH(s,m)) = m

```

Figure 5.2. An Example by Using NEW FORM
for Aggregating A Structure.

The operation definition in a cluster is not used in NEW FORM.

Whereas, if it is persued for any specific reasons, it is possible to append it to the effect part as implementation portion for the structure type. If there is any hidden operation in the operation part, an asterik ('*') is appended prior to the operation name to denote the difference from the regular operations.

5.3-Problem 3: Should an object which is a structure be allowed to have a longevity?

As stated in Definition 5 in Chapter 3, a structure defined in the Unger model is $o = (d, a, r, c, v)$ composing of n objects $o_i = (d_i, a_i, r_i, c_i, v_i)$, $1 \leq i \leq n$. A structure does have information of corporality included in the model, but the relationship with its object components o_i has not been discussed. Before each object component o_i is invoked to join the others into a structure, it itself must be complete with existence of five components $(d_i, a_i, r_i, c_i, v_i)$. Once the aggregation is done, the five components of each object which composes the new structure are not available any more, based on the concept of abstraction. We expect any characteristic of these component objects is not to be violated or destroyed or conflicted after it is used in a new structure. Thus, let us take a number of situations in which the relation between a structure and its objects might have.

The longevity information of a structure has not been discussed overtly in the Unger model. As to the longevity of each object in a structure, since an object must have existed before defining a structure, the object's longevity could be considered as either effective or in-effective under certain conditions, within the

structure. If the longevity of an object within the structure becomes in-effective under certain conditions, its longevity will be resumed once it is released out of the structure as its own individual. What "in-effective under certain conditions" means will be defined in Case 3.

Three possible situations occurring between a structure and its objects are set up in order to find out (1) whether a structure has an overt longevity, and (2) if it does, with what approximately workable principle or rule, the father-son relationship can be in harmony.

case		structure		object
1		no		effective
2		yes		effective
3		yes		in-effective (under conditions)

Case 1: The Structure is intentionally designed to satisfy more complex data, and it supports the development of structured programming. The purpose of structured programming is to enhance reliability and understandability. If a structure in which more than one object exists does not have longevity specified, then there is no protection on the structure as a whole to allow change. Thus, this case is not desirable.

Case 2: If a structure has a longevity information available and its objects have their longevity information effective, then a rule is needed to allow these two levels to coexist without any loss or violation or conflict. Before a rule suitable for this case is stated, let us review four categories of longevity which will be used in the following.

Let the rank of longevity be ascending in parallel with the order of ability to allow change. For a structure, the longevity could be one of three categories except fixed. An object with fixed longevity exists prior to the existence of the context in which it is used, such as alpha-numeric alphabets. A structure would not have a case of fixed kind for its creation within the system, because a structure is created by a user while utilizing the system. So, the category of longevity for a structure would be static, dynamic, or fluid. The rank order of the three longevities is static, dynamic, and fluid in an ascending sequence.

Consider a rule to be called RULE 1 for Case 2. Assume a structure called FILE containing objects of type RECORD, R_i . While observing RECORDs as a structure in the hierarchy, the

structure RECORD has element(s), E_i , as its components with their own type(s).

RULE 1. The longevity of a structure must not be less (more or equivalently) restrictive than any composing objects. (i.e., a structure has lower or equal rank to its components.)

Three examples with different longevities are depicted to view the possible longevity set of its components under RULE 1 are stated in Figure 5.3.

If Case 2 is chosen to solve the problem, then it is expected that the longevity of a structure would not violate the longevity of any component object. The longevity of a structure level must be more restrictive than its immediate lower level, and so on down to the innermost level. Take an example for clearer illustration. Suppose there is a structure composing objects o1, o2, and o3. If o1 is dynamic, then the structure cannot be fluid, but with the choice of static or dynamic. As we know about the dynamic object, an object which is dynamic has incarnation record generated as any change of its value occurs. If the structure is allowed to have less restrictive longevity than o1, the characteristic of o1 will certainly be violated. Just as the same philosophy as the construct of pyramid or hierarchy, the toppest level is the most restrictive with the smallest range.

In Figure 5.3, a nested structure with FILE structure, RECORD structure, and ELEMENTS is given. And three categories of

longevity on ELEMENT assigned one at a time lead to different possible sets of longevity for each RECORD (as a component of FILE) and for the FILE.

- (1) If an ELEMENT, E_i , is static, then its structure R_i has only one choice of longevity based on RULE 1, that is static. Take any single RECORD, R_i , as a composing object next. Since it must be static, ELEMENT E_i is static too, without alternative choice.
- (2) If ELEMENT is dynamic, then R_i could be (a) dynamic or (b) static. In the case of (a) dynamic, FILE may be dynamic or static. In case (b) static, FILE must be static.
- (3) If ELEMENT is fluid, then R_i could be (a) fluid, (b) dynamic, or (c) static according to RULE 1. When R_i is (a) fluid, FILE can be fluid, dynamic, or static. When R_i is (b) dynamic, FILE can only be either dynamic or static by Rule 1. If R_i is static, then FILE has to be static.

<u>Object</u>	<u>Longevity</u>		
ELEMENT <u>E_i</u>	= static	= dynamic	= fluid
RECORD <u>R_i</u>	∈ {static}	∈ {dynamic V static}	∈ {fluid V dynamic V static}
FILE	∈ {static}	∈ {dynamic, static} V {static}	∈ { fluid, dynamic, static} V {dynamic, static} V {static}

Figure 5.3. Sets of Longevity of A Structure Example
Based on RULE 1.

After viewing the possible longevity for a hierarchy of a structure based on RULE 1 in Figure 5.3, let us take an example to see how the rule works on a nested structure FILE-RECORD-ELEMENT in Figure 5.4. Since the simplest level was created earliest, assume that there are E1, E2, E3, E4, and E5 to be invoked to aggregate two occurrences of a structure, R1 and R2. Suppose E1 is dynamic, and E2 is fluid. Thus, R1 composing E1 and E2 must be dynamic or static. Suppose E3 is static, and E4 and E5 are both dynamic. Then, R2 has to be static. Expanding to next higher level, we have a structure FILE composing R1 and R2. Since R1 is dynamic and R2 static, the FILE has exactly one choice, that is static.

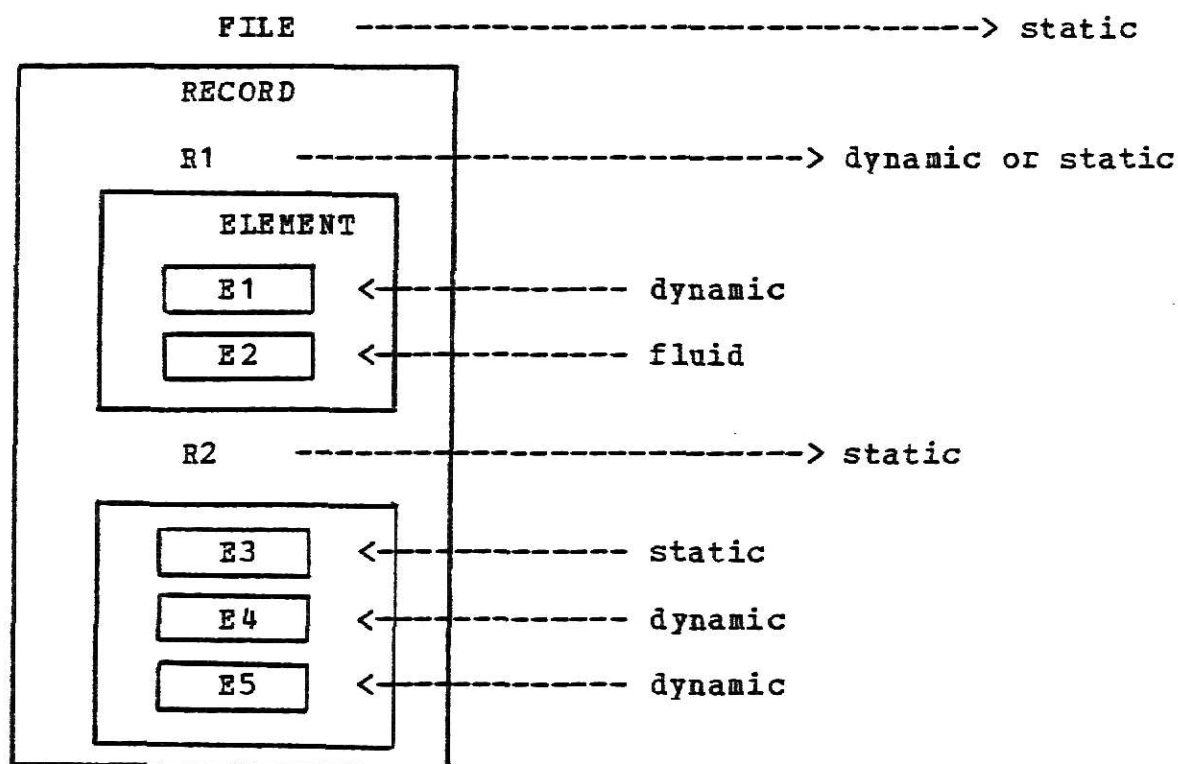


Figure 5.4. An Example of A Nested Structure on Case 2 by RULE 1.

Take an example in the real world which follows RULE 1. In Section 3.7, the seventh object called LOCAL_ADDR is a structure with three string-typed components. The first component is "street with number" with dynamic longevity. The second is "city of state with zip-code" and has dynamic longevity. The third one is "country" with fixed longevity and value "U.S.A.". If by RULE 1, the structure LOCAL_ADDR has to be static, since the third component is static. However, in normal cases, LOCAL_ADDR would be expected to be dynamic while being used. RULE 1 results in limitation of occurring to some cases illustrated as the example.

The advantage to have the structure with a longevity value is the future need of the structure to be used as a component of another structure at upper level. Then, the current structure

would be used as a component. Since any object to compose a structure must be complete prior to aggregation, it is not eligible if the current structure object does not have longevity available. So, a corporality for a structure is necessary.

The disadvantage is that RULE 1 would finally limit the outmost level in the structure hierarchy to be nearly always the lowest rank. The only instance that the outmost level, e.g., a structure, does not have longevity static is when none of the components of a structure has a static longevity in. Since the inner-most components are defined the earliest, the longevity of a structure cannot affect its components to follow the rule any more, as it is created for needs. Consider an example of a record R2 composing three elements E3, E4, and E5. Suppose the longevity for R2 is dynamic, and E3 is dynamic, E4 is static, and E5 is fluid. According to RULE 1, this example is not allowable, because E4 has lower rank than R2's rank. If we do follow RULE 1 to organize the structure hierarchy, then it is expected that almost all structures would be forced to be static (i.e., the lowest rank). It would turn out that most occurrences of structures are limited and forced to be static, composing of all static components down to the inner-most level, for information like historical event with eternally fixed binding.

Case_3: Consider this case that a structure has a determined longevity and the longecity of its component(s) would become ineffective under certain conditions. It is important not to consider components simply be in-effective but "with certain condition(s)". If the restrictive control of change upon an

object becomes in-effective after the object is called into a structure, the original properties of the object might be destroyed by any change. If as a component object is released from the calling structure, the object, which became in-effective and converted within its calling structure, can resume with its pre-calling identity of longevity to exist as what it was originally. Therefore, the originality of the object could be remained.

In Case 1 and Case 2, component objects are effective within a structure, to consider possibly more flexible alternative on longevity, the longevity of any one of these objects is allowed to become in-effective only with certain conditions. To prevent from the violation on its originality, a mechanism for tracing the originality will be included in NEW RULE. Thus, the consideration of losing the essence of an object can be released. What "in-effective with certain conditions" means is that any component object would temporarily release its longevity binding for the purpose of cooperating with a calling structure, but it is very important to allow the temporary binding-release under certain conditions. What these conditions are will be presented in NEW RULE later. The importance of these conditions is (1) to avoid the unconditional over-release of the longevity of a component object, and only the objects which are needed to become in-effective can have such a release, (2) to lead the in-effective objects to be converted in a fashion by which the pre-calling status of the objects can be traced back by a tracing mechanism. The details about how conditions, conversion, and tracing mechanism work between a structure and its components will be

stated in NEW RULE.

A new rule is needed for Case 3 to establish a structure hierarchy. Let it be named as NEW RULE.

NEW RULE: When an object is summoned (or called) to be a component for a new structure, a new incarnation of sequence number would be initialized to be zero, and the previous sequence number is stored.

Within the structure, each component follows the rules that,

- (1) if its longevity is less restrictive than or equal to its structure's, convert its longevity to the structure's,
- (2) if its longevity is more restrictive than the structure's, convert it to dynamic whatever it is, and
- (3) a special case is to convert the object to dynamic for storing incarnations, if both the object and its structure are fluid.

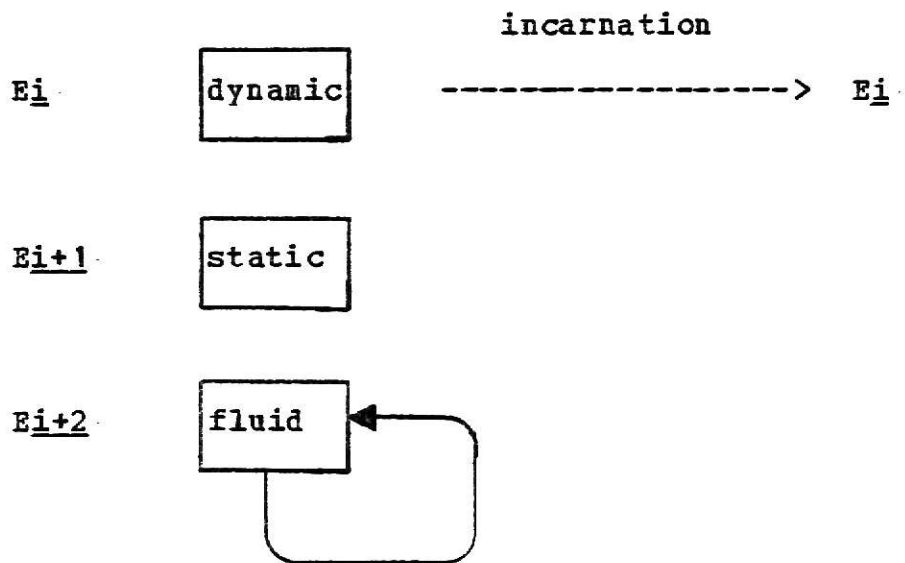
Once the object is returned out of its structure to its own existence, its pre-called originality would be traced

back without any change by the following rules,

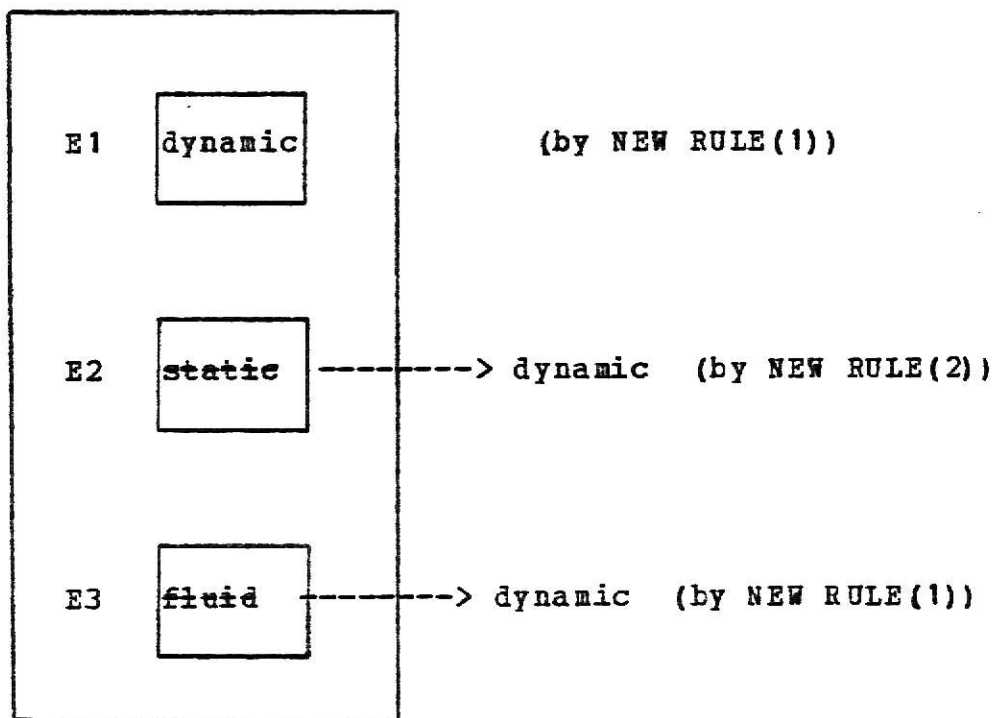
- (a) if its structure is static, then
the object would be returned with the
current incarnation d.0,
- (b) if its structure is dynamic, then
the object would be traced by the previous
incarnation as d.-n, where n is its sequence
number, and the current incarnation within
the structure is thrown away, and
- (c) if its structure is fluid, then the
object would be returned with the
incarnation d.-n, where n is the sequence
number of the object.

Note: By the rule, a component object which is static would be allowed to change its value within a structure. However, it will still trace its pre-calling identity by the mechanism, incarnation record, provided in the rule. Thus, the static object will not lose its identity even if there is any value change on it after being called into a structure.

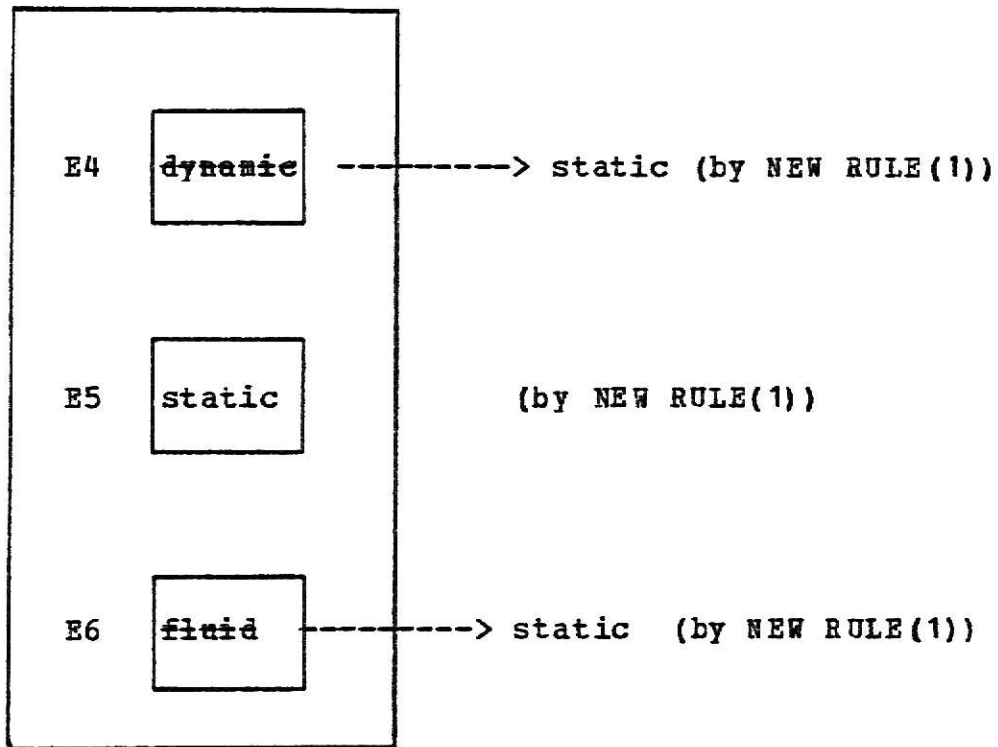
An example following NEW RULE is given in Figure 5.5, as follows,



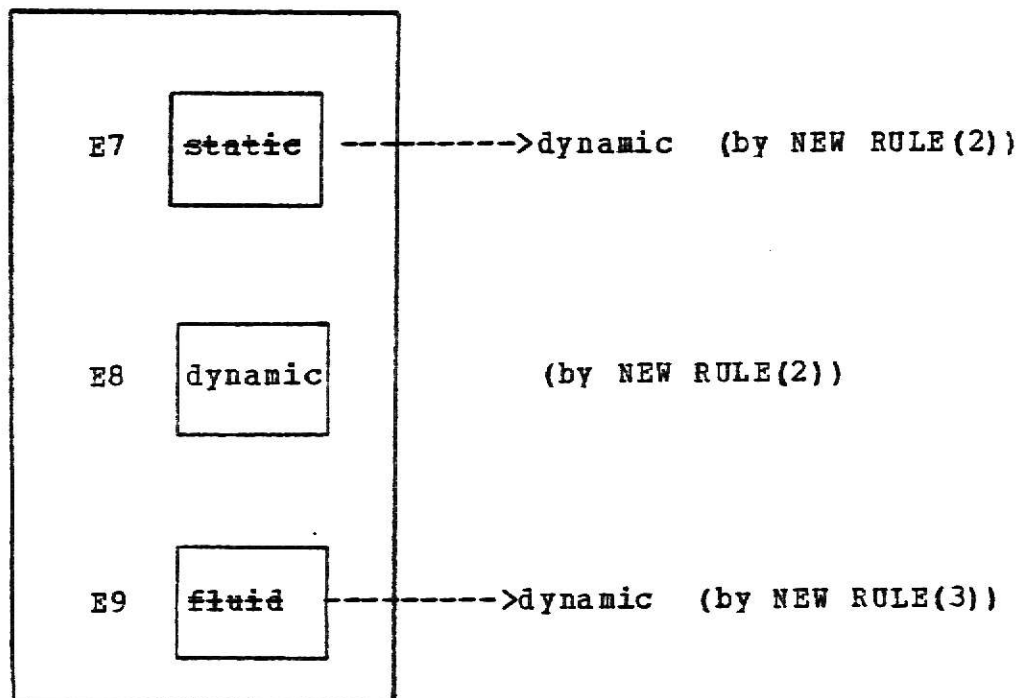
R1 <--- dynamic



R2 <--- static



R3 <--- fluid



FILE <--- fluid

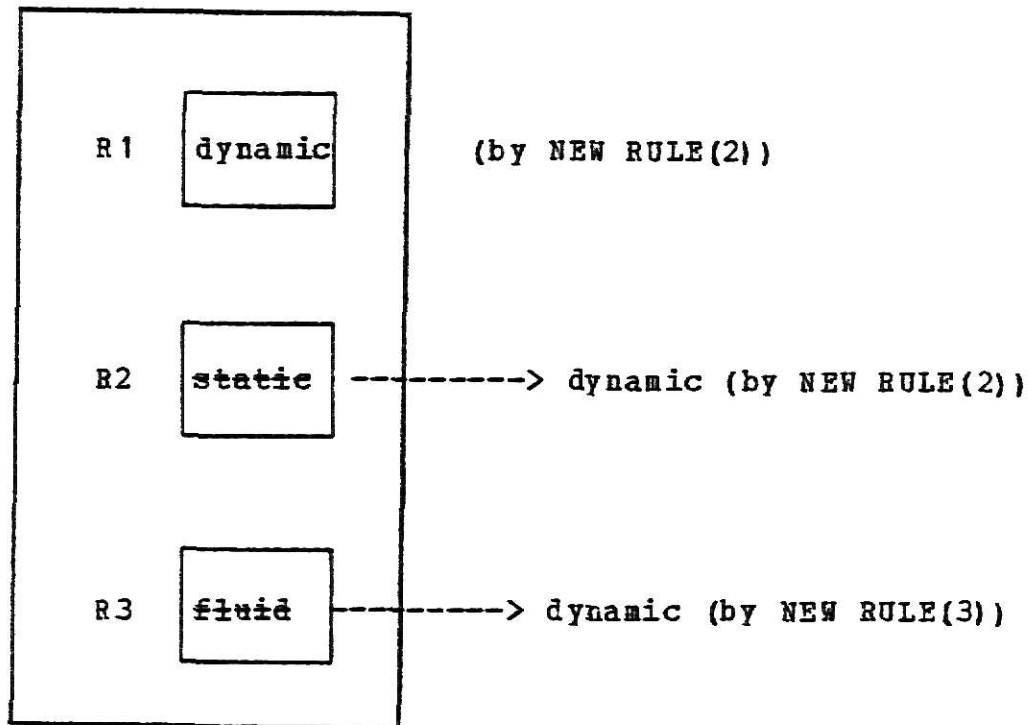


Figure 5.5. An Example of A Structure on Case 3 by Using NEW RULE.

With the three cases and rules proposed above, the problem to make structure reliably coexist with its objects in harmony is supplied with a possible solution in NEW RULE.

5.4 Problem 4: How to avoid the inconsistency on the same piece of data represented in different systems?

To establish consistency of data in two systems, S1 and S2, we wish for an object in S1 to have a unique object in S2, and vice versa. It would be a one-to-one onto mapping between two systems. Besides, if we can insure that anything happening in one system

also does in the other, the consistency of the same object in different systems would be accomplished.

Consider a general system composing of a four-tuple of objects, values, data maps, and procedures [12].

Let

$$S1 = (E1, V1, D1, P1)$$

and

$$S2 = (E2, V2, D2, P2).$$

Further, let π be any procedure in any procedure in $P1$, mapping $D1$ into a new set of data maps $\overline{D1}$, and let ρ be a system map from $S1$ into $S2$:

$$\rho : (E1, V1, D1, P1) \dashrightarrow (E2, V2, D2, P2)$$

$$\begin{array}{ccc} D1 & \xrightarrow{\rho} & D2 \\ \pi_1 \downarrow & & \downarrow \pi_2 \\ \overline{D1} & \xrightarrow{\rho} & \overline{D2} \end{array}$$

Representations are equivalent only when system maps are one-to-one onto and commute with the procedures. A one-to-one onto map pairs each element of $D1$ with a unique element of $D2$, and vice versa. If (1) ρ is one-to-one onto and (2) $\rho\pi_1 = \rho\pi_2$, then an object could consistently exist in two systems. Suppose ρ is one-to-one onto, and the inconsistency occurs if the second condition

fails (i.e., $\pi_1 \neq \pi_2$). Since, in practice, the condition that ρ is not one-to-one onto could frequently be prevented [12], the question left is how to find π_2 to best present π_1 .

An example of real number is given in the following, which illustrates why ρ may fail to be one-to-one onto. Usually, the reason it occurs is that the sets V_2 and V_1 have different sizes. In the case of machine representation of the real numbers, only rational numbers may be represented with complete accuracy on a machine, and the common floating-scale representations can represent only a finite number of rationals at that. Since in our scope of problem-solving the situation that the one-to-one onto mapping fails does not occur frequently, the most important condition is the second condition, $\pi_1 \neq \pi_2$.

Consider how to conquer the problem if the object was not represented consistently in two systems. Assume the paycheck information of Mr. So-and-So had been recorded in two systems S_1 and S_2 . A certain need is requested so as to have two values of the same object in two systems, SALARY, be considered simultaneously. We found that the SALARY figures in S_1 and in S_2 are inconsistent. S_1 has \$2,500 of type integer, but S_2 contains \$2,500.00 of type real. The concern is to make them consistent. It is supposed to have

$$\mu \left(\begin{array}{c} \text{SALARY} \\ \text{So-and-So in } S_1 \end{array} \right) = \mu \left(\begin{array}{c} \text{SALARY} \\ \text{So-and-So in } S_2 \end{array} \right)$$

However, the results in two systems are

SALARY		μ	value
So-and-So			
in S1			2,500
in S2			2,500.00

Taking the longevity and chronological identification into consideration, let the rule to solve this problem be called as RULE 4.

RULE 4: In system S_i and S_j , where $i \neq j$, an object

o_i from S_i and an object o_j from S_j record the same piece of information. If the value $v_i \neq v_j$, then convert one to be equal to the other according to the rules,

- (1) if longevity $l_i = \text{static}$ and $l_j \neq \text{static}$, then v_j is converted to be equal to v_i , and vice versa.
- (2) if both o_i and o_j are static, then the chronological identification t_i and t_j are next alternation to consider. If t_i is farther than t_j (e.g., 04-08-80 is farther than 04-15-80), then convert v_i to v_j ,

and vice versa, and

- (3) if neither l_i nor l_j is static, then
follow the same rule in (2).

Considerations such as (1) whether the conversion would violate the authorization of change, (2) whether the object after conversion is a correct solution expected from the original users and/or the current users, and (3) any other neglected questions are not considered.

5.5 Problem 5: Define in the objects the fact that the object is based on a primitive, denoted as Ω .

The Greek omega, Ω , which appeared in a couple of examples of objects in Section 3.3, stands for a primitive at the machine level with the coding and compacting in a specific computer system. Since the representation of coding, format, location, and access method might vary over time and/or from an environment to another environment, it allows the users to ignore this level of representation and to concentrate on other details more immediately relevant.

In the conventional data environment, user (both designer and programmer) have to know answers to the following questions before manipulating the data:

What is its format?

What is it located?

How is it accessed?

It turns out to be a problem that, as the needs of the enterprise change, the format of the data might change. As a result, a user must spend an increasing percentage of his time in maintenance and updating. The users should be oriented toward the information content of the data, rather than being concerned with details of representation and location [1].

The concept of abstract data types encourages the separation of the abstraction of data object from its implementation. The user of an abstract data type only concerns himself with the behavioral semantics of the type: what meaningful operations can be applied to objects of the type. Irrelevant detail of the internal representation and structure of the type is suppressed.

During the design phase, the consideration of physical aspect is suggested to be neglected by the user. Usually, a data base will be put into execution long after the design phase. Therefore, it is not efficient to consider the machine level at this phase. It should also be mentioned that the principle of abstraction has several obvious similarities to the concept of data independence in data bases [4]. Both provide the separation of logical view from the implementation. If any change occurs to the system, the specification will hopefully not be impacted by the change.

As to the definition of an object, should the representation component be neglected as a null in the model or be denoted as a system-dependent primitive at machine level or simply be deleted from the model? If this component is omitted, would the model be

complete as before and simpler?

In this chapter, five problems are presented but not completely solved. The purpose of this chapter was to browse through some related viewpoints in the articles, and to encourage the future efforts. Some ideas and possible solutions are raised by the author. Within several cases and a number of rules designed for the problems, some defects might be criticized.

CHAPTER 6

SUMMARY

It becomes clear that it is impossible to find an optimum model of data capable of satisfying all the needs of all users of all kinds of information systems [2]. In the paper, a number of existing data models were introduced. It has been found that there is no such thing as the "simplest" and "best" data model. Using several criteria, an attempt to evaluate the four models will be made in this chapter. After studying these models, we will discover that the concept of data abstraction and the structured-programming concepts are frequently being contents of the models. It is apparent that these concepts have dominated the research of data for a long time. In the last of the chapter, the current status and future work on data abstraction which dominantly influences data modelling will be stated.

6.1 Evaluation of the Models Based on Six Criteria

A modelling technique to abstract data must satisfy a number of desired properties if it is to be useful. There are six criteria presented below to permit us to evaluate briefly the models being covered in the paper. Consider the list of requirements for a data modelling as follows:

1. Easy to comprehend.

A model (or a specification technique) needs to be easy to read and to understand for all those people who work with the notation being used with a minimal difficulty. Properties of specifications which determine understandability are size of a unit and lucidity.

2. Easy to construct.

Ease of construction means, first, the distance of understandability between concepts in the data model and concepts in the mental model of the users is minimal [13]. The second facet means the difficulty of constructing a specification.

3. Formal.

A model should be written in a notation which is mathematically sound. It must consist of a formally definable set of allowable expressions. Thus, a thing (real or abstract) in the real world can be assigned a "meaning" by a semantic mapping" [15]. There are two important aspects to formalization: first, a model must fully define the syntax and semantics of the structure or contents written in the model. Second, the notation must provide the development of methodologies for proving the correctness of use of data abstraction.

4. Minimality.

A model only describes the relevant details and nothing more. The details which are of interest must be described precisely

and require as little extraneous information as possible. For example, a model must include what operations a data unit (an object or a structure) should perform, but little or nothing about how the operation is performed.

5. Wide range of applicability.

Each specification methodology can describe a class of associated concepts in a straightforward fashion. As to the concepts outside of the class, they may be defined but possibly with difficulty. Thus, a methodology is more useful if the class of concepts is larger [11].

6. Correctness.

If a model does not provide correctness, then its other properties have no meaning since we cannot rely on it. The proof can be carried out as the modules are developed, rather than waiting for the entire model to be created.

By means of these criteria stated above, let us review and evaluate four models introduced previously in the paper. The evaluation on the models is made as the order they were introduced: (1) Mealy's theoretical model, (2) Liskov's data model, (3) the Unger model, and (4) algebraic specification.

6.1.1 Evaluation on the Mealy's Model

The theoretical model proposed by Mealy is a system of sets of entities, values, data maps, and procedure maps. Since structural data map is introduced into the model, any data map which assigns

a value to an attribute of the entity can be defined as a composition of a structural data map followed by a non-structural map. The construction of any complex entity can be easily accomplished through structural maps by using pointers. The model adopts the notion of set and of mapping, and it is very mathematically sound. The correctness of proving, therefore, can be easily achieved by such a mathematical formalization to assure the mapping to be one-to-one onto. Implicitly, the operations of an entity are defined through mapping. The model only includes entities, values, data maps, and procedure maps. As to minimality, it provides a minimal specification of data. On the other hand, such a minimality hurts the criteria of comprehensibility and the constructibility to the reality. Categories of type of data, its organization, and representation are described in data description, which is outside of the body of the system (or model), and these categories are defined by procedure maps. That is, data description is a specification of the maps and the characteristics of the entity and value sets. This model, which separates information about the entity and value sets apart from the body of the system, does not offer readers with ease of understanding. The structural data map can define the data so complex as a result obtained through a computation, a series of processes, or an algorithm, to make the range of applicability be as wide as it covers.

6.1.2 Evaluation on the Liskov's Model

In Liskov's model, the distinction between abstractions and

implementations permits a module to be decomposed, and, thus, comprehension can be easier. A module might be a multi-procedure module such as a single data base, i.e., a single procedure comprising a number of invoking operations of data types. Imposing upon the data abstraction concept and structured programming technique, a reader can easily understand what a module does rather than be concerned with how a module performs in detail. What a reader sees in a module is the module name with any required operations or procedure name(s) at the outmost level of the construction hierarchy. Therefore, one module at a time is studied to determine that it implements its abstraction. The study requires understanding the behaviour of the abstractions it uses, but it is not necessary to understand the modules implementing those abstractions. These modules can be studied separately. One more advantage provided for the ease of reading is the use of compound name of operation call, e.g., `stack$push(s,t)`. In the compound name, the type-name prefix enhances the understandability. The model allows the use of different data types, and the prefix helps clear the possible ambiguity. Besides, the operation call is clearly distinguished from procedure call. The modularity makes a model be easier to understand, to construct, and to prove correct. This model is very programming-oriented, thus the decomposition of the proof is essential instead of mathematical proof. It permits the proof to perform on decomposing modules. Based upon the data abstraction concept, the entire group of procedures as a module permits all information about how these procedures are grouped and interact to be hidden from other modules. The hiding leads to simpler

modelling. The minimality could be enhanced by this to a degree. Although the model is not in mathematical notation, it is still not difficult to be formalized by the discipline of modularity. Without mathematical deduction or proof, the correctness can still be assured through modularity. With the hierarchy and modularity built higher and higher, the range of applicability could be wide. There is a critic made on the operation definition code on the operation cluster. Since a data specification should not be viewed as describing the eventual implemented program but merely as a means for understanding what the operation is to do [3], the code part for defining operations of a data type is not recommended.

6.1.3 Evaluation on the Unger Model

Among the four models presented in the paper, the Unger model provides the highest degree of comprehensibility for its naturalness. Five components of an object cover most information about a piece of data. Thus, the constructibility to the reality is comparatively good. Some interesting properties uncovered in many other techniques are collected into the model, such as spatial and time consideration, concurrency, corporality, copiability, and etc. As to the constructibility to establish the model, it is ranked high for the modularity feature. By using a structure, detailing and construction of action, and control, a module, if it is decomposable, can be decomposed into several lower levels as a hierarchy. With the data abstraction concept, the reader only has to know the closest level which is most

immediately relevant. For the same reason stated above, the correctness can, therefore, be proved and assured much easier. However, comparing with some techniques as algebraic specification or Mealy's theoretical model, the minimality of the Unger model is poorer. It is a trade-off between minimality and comprehensibility. The model is abundant with the descriptive power on data, and the scope of data to be defined is broad, such as simple value, algorithm, value obtained through spatial coordinate, action, or a collection of values. One major property provided by the model is the concurrency, which increases its range of applicability, since many problems encountered in the real world have solutions involving concurrency. With respect to formality, the model is mathematically sound. It is described by a mathematical notation employing sets, tuples, mappings, and an extension of the Backus-Naur form. Together with the properties of modularity and control conditions, the correctness of the model can be proved with ease.

6.1.4 Evaluation on the Algebraic Specification

Algebraic specification provides a group of applicable operations and their functionality. The first part of operation declaration with one-to-one mapping provides the specification technique with ease of comprehension and construction. Whereas, it somewhat limits the range of applicability by the algebraic approach. For example, an uncountable domain, such as the real numbers, cannot be defined using the algebraic specification. It might appear that the comprehensibility may suffer for the

difficulty in determining equivalence of two arbitrary expressions yielding equal results. However, an algorithm provided to deduce a canonical form can solve it. The canonical form comprises a minimal constructor set of operations to support an abstraction implementation, and the form results from simplification of the expressions obtained by formally applying the operation symbol to the expressions. Also, the recursion structure makes the comprehension easier than as it looks. Since a canonical form can be derived mechanically for the specification, the constructibility to a model can be enhanced by such a development of construction methodology. But as to the constructibility to the reality, it does not include many other interesting properties of data, such as corporality. Algebraic specification is minimal, because it appears that hidden operations are not needed in the model. Only using a few lines, a fairly complex object has been completely defined. On the other hand, a model, if it is good with respect to the criterion of minimality, might be difficult to construct and to comprehend, unless it is given in terms of a natural representation for the defined objects. Fortunately, this specification provides a canonical form as its natural representation for the defined objects to release from such a trade-off.

As to the notation, the algebraic approach can be easily formalized by borrowing from existing mathematics. The expressive power of semantics by its notation had been explained in the Problem 2 in Chapter 5. Not only the syntax of operations are declared, but their semantics of effects and behaviour are described in the axioms too. With respect to the criterion of

correctness, the specification proves the correctness by implicitly showing (1) that the defining axioms hold and (2) that the mapping in the declaration is one-to-one. A promising way to establish the correctness is by means of mathematical proof [10]. Since algebraic specification is formalized by existing mathematics, it is the best model with respect to formality for the ease of correctness-assurance by a mathematical proof.

6.2 The Role of Data Abstraction Played in Data Modelling

Over the past decade, a dominant theme in the research activity of methodologies and languages has been to explore the issues related to data abstraction. By the concept of data abstraction, a piece of thing, such as a program, can be organized into modules in the way that implementation detail was localized as much as possible. Much of the work in abstraction research activity is identified with the concept of abstract data types. The strategy of the work focuses on how to form modules consisting of a data structure and its associated operations, and apparently the focus of attention is set on data rather than on control. The abstract data type effectively treats these modules in the same way as available types such as integer to be treated. Such a module includes the properties of a new group of data objects, and their associated operations which will be performed on the objects of the new type by giving the effects these operations have on the values of the objects. The concept of an abstract data type was first implemented as a class in the language Simula 67 (by Dahl et al). Other versions of this basic concept have seen included in

many recently proposed languages; for example, there are clusters in the language CLU (by Liskov and Zilles), forms in the language Alphard (by Wulf), classes in Concurrent Pascal (by Brinch Hansen), and opaque types in PASQUAL (by Tennent). There are important differences in generality between these [8], but a detailed discussion of these is beyond the scope of this paper.

The research work of abstraction has provided formal notations such as abstract models, input-output specification, and algebraic axioms for describing the effects that operations perform on the values. Other properties such as time and space requirements, memory access patterns, reliability, and synchronization have not been much addressed by the data type research. Besides formally deriving the correctness of a model from the mathematical function, certain properties of the computation that are important must be preserved.

6.3 Future Work

There is a trend toward coexistence of different data models within one system [5]. In the present research of data modelling in data base management, there is no optimum model satisfying all kinds of needs. Since the range of applicability is different for the different models, it is expected that using a mechanism combining different techniques when describing a large unit would be a profitable approach [11].

The principle of data abstraction has a significant influence in the area of data study, and it will continue to do so. A model

with no problem is not obtained in the paper; however, it is expected that the coexistence concept can be developed to produce a generalized mechanism by which a desired data model can be generated with the semantics preserved. This area involving data abstraction appears to be a very promising one for future study, and the efforts in extending and formalizing existing models and in proposing a new methodology are of the most importance.

BIBLIOGRAPHY

1. Atre, S., Data Base: Structured Techniques for Design, Performance, and Management, Wiley-Interscience Publication, 1980, Page 16.
2. Falkenberg, E. D., "Data Models: the Next Five Years", in INFOTECH State of the Art Report Data Base Technology Volume 2: Invited Papers, Infotech International, 1978, Page 53-68.
3. Guttag, J. V., and others, "The Design of Data Type Specification", in Current Trends in Programming Methodology, Vol. IV, Data Structuring, Yeh, R. D. (edt.), Prentice-Hall, Inc., 1978, Page 60-79.
4. Hammer, M., "Data Abstractions for Data Bases", in Proceedings of Conference on Data: Abstraction, Definition and Structure, ACM SIGPLAN Notices, Vol. 8, No. 2, 1976, Page 58-59.
5. Inmon, W. H. and Friedman, L. J., Design Review Methodology for A Data Base Environment, Prentice-Hall, Inc., 1982.
6. Kent, W., Data and Reality, North-Holland Publishing Company, 1978.
7. Larson, J. A., Database Management System Anatomy, Lexington Books, 1982.
8. Linden, T. A., "The Use of Abstract Data Types to Simplify Program Modifications", in Proceedings of Conference on Data: Abstraction, Definition and Structure, ACM SIGPLAN Notices, Vol. 8., No. 2, 1976, Page 12-23.
9. Liskov, B. and Zilles, S., "Programming with Abstract Data Type", SIGPLAN IX, No. 4, April 1974, Page 50-59.
10. Liskov, B., and others, "Abstraction Mechanisms in CLU", in Communications of the ACM, Vol. 20, No. 8, August, 1977, Page 564-576.
11. Liskov, B. and Zilles, S., "An Introduction to Formal Specifications of Data Abstractions", in Current Trends in Programming Methodology, Vol. 1., Software Specification and Design, Prentice-Hall, Inc., 1977, Page 1-32.
12. Mealy, G. H., "Another Look at Data", in AFIPS Conference Proceeding, 1967 Fall Joint Computer Conference, 1967, Page 525-534.
13. Nijssen, G. M., "The Next Five Years in Data Base Technology", in INFOTECH State of the Art Report Data Base Technology Volume 2: Invited Papers, Infotech International, 1978, Page 213-256.

14. Shaw, M., "The Impact of Abstraction Concerns on Modern Programming Languages", Computer Science Department, Carnegie Mellon University, Pittsburgh, Pa., April, 1980.
15. Steel, T. B., "The Current ANSI/SPARC Proposals", in INFOTECH State of the Art Report Data Base Technology Volume 2: Invited Papers, Infotech International, 1978, Page 345-356.
16. Sundgren, B., Theory of Data Bases, RETROCELLI Information Systems Series, 1975.
17. Tennent, R. D., Principles of Programming Languages, Prentice-Hall International, Inc., 1981.
18. Unger, E. A., "A Natural Model For Concurrent Computation", Kansas State University Technical Report 78-35, Dissertation, 1978.

A STUDY OF DATA

by

HSAO-YING JENNIFER TIAO

B. A., Tamkang University, 1980
Taiwan, Republic of China

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1983

ABSTRACT

The dominant importance of data in the information world reflects in the attention it receives in programming language and in database studies. It is a fact that the notions about or meaning of data are still not clearly understood. Languages investigations together with other research projects have provided formal notations, based on data abstraction, such as abstract models, input-output specifications, and algebraic axioms. A user is concerned not only with pure functional correctness but also with properties such as time and spatial requirements, complex aggregation behaviour, and concurrency.

Formal notations being included in the report are (1) the traditional File Model, (2) the Mealy's Theoretical Data Model, (3) the Lispov's "Abstract Data Type", (4) the Unger's "Natural Model for Concurrent Computation", and (5) the Algebraic Specification.

This report has (1) surveyed the five data modelling notations that have been concerned with the abstract description of information, (2) created a number of examples of data model in the Unger's Natural Model, (3) raised and partly solved five problems occurred in the Unger model and in the database world, and (4) evaluated the last four data models based on five criteria. For some of the problems proposed, a number of designed rules have been presented to

attempt to solve the problems. To illustrate whether the proposed rules would work, several examples for the corresponding cases have been given right after the rules.

The concept of coexistence of different data models within one system would be a profitable approach for the future work. It is a fact that there is no such an optimum model satisfying all kinds of needs. While attempting to solve some problems, the author combined a number of techniques from different methodologies to the problems. The area involving data abstraction appears to be a very promising one for future study, and the efforts in extending and formalizing existing models and in proposing a new coexistence mechanism is of the most importance.