THE DESIGN OF SEMANTIC DATABASE MODEL SDBM

by

LINCHI XIE

B.Sc. Shanghai Jiao Tong University

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE (BUSINESS ADMINISTRATION)

in

THE FACULTY OF GRADUATE STUDIES

Faculty of Commerce and Business Administration

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

February 1987

© Linchi Xie, 1987
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at The University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Faculty of Commerce and Business Administration

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date: February 1987
ABSTRACT

This thesis is mainly concerned with semantic data modelling related to database design. The domain of this research is restricted to general data modelling and the discussion is carried out at the conceptual level.

The thesis assesses a number of serious modelling shortcomings of the conventional data models and reviews several basic principles and mechanisms developed in current semantic data modelling research. Based on these findings, the thesis identifies the inadequacy of the conceptualization of data modelling and develops the two-view conceptualization of data modelling. The basic idea behind the two-view conceptualization is that the conceptual structure of the applications being modelled should be separated from its external data representation.

A new semantic database model, SDBM, is designed based on the conceptualization. The model makes a clear separation between the conceptual structure and its external data representation. It offers a data type mechanism to deal with the data representation, a window mechanism to model the conceptual structure, and a transaction mechanism to provide database operations. One of the major extensions of the current semantic data models is that with SDBM the specialization relationship is just a special case of constraints that can be specified among SDBM windows.

A formal syntax and informal semantics of SDBM are given in the thesis along with comparisons between SDBM and a closely-related semantic data model, Taxis.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ii</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>vi</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 OBJECTIVES</td>
<td>1</td>
</tr>
<tr>
<td>1.2 APPROACH</td>
<td>3</td>
</tr>
<tr>
<td>1.3 PREVIOUS WORK</td>
<td>5</td>
</tr>
<tr>
<td>1.4 THESIS OUTLINE</td>
<td>8</td>
</tr>
<tr>
<td>2. REVIEW OF DATA MODELLING</td>
<td>10</td>
</tr>
<tr>
<td>2.1 SEMANTIC MODELLING PROBLEMS OF THE RELATIONAL DATA MODEL</td>
<td>10</td>
</tr>
<tr>
<td>2.1.1 TUPLE ORIENTATION</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2 SEMANTIC OVERLOADING</td>
<td>12</td>
</tr>
<tr>
<td>2.1.3 ATOMIC DOMAINS</td>
<td>13</td>
</tr>
<tr>
<td>2.1.4 NORMALIZATION</td>
<td>15</td>
</tr>
<tr>
<td>2.1.5 RELIANCE ON USER-CONTROLLED IDENTIFIERS</td>
<td>16</td>
</tr>
<tr>
<td>2.1.6 GENERALIZATION/SPECIALIZATION</td>
<td>18</td>
</tr>
<tr>
<td>2.1.7 REQUIREMENT OF HOMOGENEITY</td>
<td>19</td>
</tr>
<tr>
<td>2.1.8 AGGREGATION</td>
<td>20</td>
</tr>
<tr>
<td>2.2 SEMANTIC EXTENSIONS TO THE RELATIONAL DATA MODEL</td>
<td>21</td>
</tr>
<tr>
<td>2.2.1 EXTRA CONSTRAINTS</td>
<td>22</td>
</tr>
<tr>
<td>2.2.2 SEMANTIC CLASSIFICATION OF RELATIONS</td>
<td>23</td>
</tr>
<tr>
<td>2.2.3 THE ENTITY-RELATIONSHIP APPROACH</td>
<td>23</td>
</tr>
<tr>
<td>2.2.4 CONCLUSIONS</td>
<td>24</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

I am deeply indebted to many people who have helped through my graduate study at the University of British Columbia. Here, I can only mentioned those who have been involved in my thesis work.

I wish to express my sincere thanks to my supervisor Prof. Robert C. Goldstein for his guidance and encouragement. I truely appreciate his time and efforts in supervising my study. I have learnt so much from his database course which has greatly stimulated my interest in database research.

I also wish to thank Prof. Yair Wand for his insightful comments on the thesis, which have helped to make the thesis more complete. Many thanks to my third reader Mr. Albert Fowler.

I would like to thank Ms. Grace Wong who have made my stay in Canada much easier. I can always resort to her for help. Thanks also to Ms. Mackie Chase who took pains to correct many typos and English language errors in my thesis draft, which makes this final version more intelligible.

Deep thanks to my friend Kefeng Xu who typed part of this thesis. We share a lot of serious thoughts as well as many not-so-serious jokes.

My study at University of British Columbia is financially supported by the Canadian International Development Agency.
1. INTRODUCTION

1.1 OBJECTIVES

It has been recognized that the tools for database design should be oriented more to the application-minded user than to the computer expert. The application-minded user is interested in the modelling of real world applications which are described in terms of entities, their attributes, their relationships, and their changes. Therefore, data models as conceptual modelling tools for database design should be based on real world entities as directly as possible.

As a result of this recognition, during the past few years many researchers have devoted a considerable amount of effort to so-called semantic data modelling. Semantic data modelling has the goal of capturing more of the meaning of the 'real' world being modelled than the conventional data models can. A large number of semantic data models have been proposed. A few examples are the Data Semantic data model (Abrial [1974]), the Entity-Relationship model (Chen [1976]), Semantic Hierarchy Model (Smith & Smith [1976]), the RMT data model (Codd [1979]), the TAXIS data model (Mylopoulos et al [1980]), the DAPLEX data model (Shipman [1982]), and the Semantic Data Model (Hammer & McLeod [1981]). From their work, a set of concepts, mechanisms, and methodologies for developing database-based information systems at the conceptual level have been established.

At the same time, many important concepts and methodologies have been developed in programming languages. Consider for example the data typing concept, abstract data types, data and procedural abstractions, control structures, and exception handling. These concepts have found extensive use in the database design. Software engineering concepts have also been applied to databases in the development of data modelling tools and
techniques (Brodie et al [1984]). The result is that many recently developed semantic data models are not just data models. Rather, they are database programming languages which are the integration of data models and programming languages.

In this thesis, through designing SDBM (Semantic DataBase Model), we will investigate alternatives for conceptualizing semantic data modelling and alternatives for incorporating programming language constructs and semantic data modelling concepts. SDBM is designed to offer a clearer conceptual view of database designs and to offer greater flexibility in certain aspects than other relevant data models. The most important aspect of this investigation is the conceptualization of data modelling in terms of two different views. On one side, we consider the 'real' world\(^1\) to be modelled as a collection of objects. These objects have properties and are interrelated in a certain structure. On the other side, objects, relationships, and properties have to be represented in the computer. On this side, the focus is on how data is represented in the computer. This is the external view of objects, relationships, and properties. The SDBM type concept is designed for external data representation. For data modelling to be complete, appropriate correspondence between these two sides must be provided. The SDBM window concept is introduced to provide such correspondence. For the user, an SDBM window allows him to see part of the 'real' world. For the designer, SDBM windows also specify the conceptual structure among objects, relationships, and properties. It is our conceptualization that differentiates our approach from the previous work.

We will also investigate the separation between the type hierarchy and the window hierarchy in databases. However, most of all, we are interested in providing a data model that has greater modelling ability than existing data models. We will demonstrate that our conceptualization of data modelling is sound through the design of the semantic database model, SDBM.

\(^1\)The 'real' world refers to the part of the real world that we are interested in.
1.2 APPROACH

This thesis starts with the question 'What are the data modelling problems of the conventional data models?'. In order to limit the research to a manageable range, we focus our attention on the relational data modelling methodology which is currently widely used and is expected to be used even more in the future. For the same reason of manageability, the domain of this research is restricted to general data modelling. By that, we do not intend to make our model capable of modelling every aspect of the 'real' world. For example, we exclude modelling of data from special purpose applications such as CAD/CAM and office procedures. What we are interested in is data that can be regularly formatted and data that is managed in many current databases. Examples which fall into our modelling realm include student registration data, personnel data, payroll data, and inventory data.

Within this perspective, semantic data modelling problems of the relational data model are identified through analyzing the data modelling implications of some of its presumptions and inherent constraints and through analyzing its inability to provide the basic modelling mechanisms that are necessary at the conceptual level. These problems set up the major target of this research.

To overcome these identified problems, a semantic database model SDBM (Semantic DataBase Model) is designed, primarily for the purpose of database design. The underlying mechanisms of this model come from two major disciplines, semantic data models and programming languages.

In this research, semantic data modelling is viewed as a set of well-established concepts including object orientation, classification, aggregation, generalization, and the concept of transaction, etc. They are the basic tools for organizing and manipulating objects, and are essential to almost all the semantic data models developed recently.
SDBM is designed to be a management facility for objects which are, conceptually, the most elementary units that can be operated on. Two other conceptual constructs are relationships and properties. Objects, relationships, and properties together form the conceptual world which is the conceptual counterpart of the 'real' world.

This conceptual world may be viewed from different angles; different views may be related to each other; and possibly different viewers (users) may have different requirements for information details. To cope with these varieties, we will develop a window concept. We will actually specify the conceptual structure through this window concept.

The computer data representation aspect of SDBM objects is handled by integrating into the system a data type facility. The system provides some built-in data types and some construction mechanisms to create complex data types. This facility enables certain automatic semantic integrity enforcements.

Through analysis of the relational data model, it is clear that the record data structure is a major cause of its data modelling problems. To solve these problems, we may abandon the record structure and look for some completely different structures, or we may modify the record structure and the way it is used. The latter alternative is taken in this research, because our further investigation reveals that the trouble with the record structure is only part of the story. The real underlying cause of relational data modelling problems is that objects are confusingly mixed up with their computer data representations. Namely, the problems mainly lie in the the way the record data structure is being used. The SDBM record structure is, therefore, designed. This kind of record permits complex component data types. Certain important inter-component constraints can be specified as part of the type definition.

The advantages of static type-checking are recognized by designing SDBM to be typed. In SDBM, every denotable value has a type. Every expression can be statically type
checked. A generalization/specialization mechanism is also provided in the SDBM type facility. Type hierarchy rules exist to check the correctness of specification. Aggregation mechanism is used in data type construction and window construction.

In addition to static type-checking, SDBM also enforces semantic integrity constraints through window checking. Window checking is primarily used for referential constraints, relationship constraints, and uniqueness constraints.

The behavioral aspect of the database is modelled by SDBM transactions which are considered to be the most basic operation units that can be used by the end user to manipulate a database. The SDBM transaction is constructed from statements which are the smallest conceptually meaningful operations. Several sequence control mechanisms are borrowed from programming languages to enhance the expressiveness of SDBM transactions.

In summary, the approach taken in this research is to first identify the semantic data modelling problems of the relational data model and the basic mechanisms used in current semantic data modelling research. Then the two-view conceptualization of data modelling is formed. Based on this conceptualization, several fundamental data modelling concepts and mechanisms are re-interpreted and some necessary modelling mechanisms are added. Finally, a semantic database model, SDBM, is designed through the integration of these concepts and mechanisms.

1.3 PREVIOUS WORK

A considerable amount of prior research work in the semantic data modelling area has influenced the design of SDBM. Usually, such research work results in new models or modelling methodologies. This section identifies previous work related to this thesis and argues the relative advantages of SDBM by pointing out shortcomings of some closely related work.
In his landmark paper, 'Data Semantics', Abrial investigated the modelling of data semantics in a mathematical model based on entities and binary relationships (Abrial [1974]). The model is rather abstract and is viewed as a conceptual tool for understanding data semantics. Hammer, Schmidt, and Weber identified the applicability of data type concepts in databases (Hammer [1976]), (Schmidt [1978]), (Weber[1976]). Brodie is one of the first researchers to develop a semantic data model which applies the data type concept to the specification and verification of databases (Brodie [1978]). His model focuses on the logical properties of data expressed in denotational specifications.

McLeod and Hammer designed a semantic data model, SDM, for database design and documentation purposes (McLeod & Hammer [1981]). The model is not computer executable. Subsequently, they developed a database programming language DIAL which is based on SDM (Hammer & Berkowitz [1980]). The data type concept is supported by DIAL. However, the data type system is not integrated together with features from data modelling. The data type system only provides a very limited number of data types and type construction mechanisms. Moreover, the concept of data type and the concept of class\(^2\) are not separated.

The two semantic data models that are most directly related to the design of SDBM are TAXIS and Galileo. TAXIS is a database programming language which provides a framework for information system design and is based on the concept of class which is applied uniformly to all the aspects of data modelling (Mylopoulos et al [1980]). Therefore, we have classes such as metaclasses, data classes, and transaction classes. The fundamental modelling mechanism of TAXIS is inheritance. TAXIS is not a typed database programming language. Well-known techniques for the data type checking in programming languages cannot be directly applied. TAXIS provides very simple data structures. Only the tuple data structure is available. In addition, in TAXIS everything is an object and therefore everything

\(^2\)The concept of class will be explained in Section 3.3.
is grouped into classes. Although such a uniform approach to data modelling makes the model simple because of scarcity of basic modelling constructs, it is not always desirable and may be excessive in certain circumstances. One of the problems with such an approach is conceptual overloading of the class concept. For example, in a TAXIS program the sole purpose of some classes is to define another class. The former classes are conceptually different from the latter and apparently do not directly interest the database designer in determining the conceptual structure of the 'real' world. Moreover, TAXIS maintains no clear separation of type and class concept. Each data type has a unique class. Finally, in TAXIS the type associated with a class is restricted to a tuple type. That is, objects always have values of the tuple type, and they can be related to each other only through properties which are single-valued functions.

Galileo is designed by Ablano (Ablano [1985]). It is designed into a strongly typed database programming language. It utilizes the benefits of static type checking. A generalization mechanism is built into the type system of Galileo. The type hierarchy in Galileo specifies the intensional aspect of its class hierarchy.

With Galileo, it is not possible to construct two extents on the same data type, and therefore, different classes must have different data types. The type hierarchy of Galileo except for abstract types is specified implicitly and established through automatic enforcement of Galileo type hierarchy rules. Such implicitness is believed to be detrimental to the development of reliable database information systems. In addition, the referential constraints between objects of different classes have to be specified manually along with their processing procedures in Galileo. Finally, it is worth pointing out that Galileo is not a data model. Rather, it is a database programming language. It is very complex when compared with many other semantic data models.

---

That is, two sets of associated instances. A set of associated instances are usually managed in terms of a class in most semantic data models.
From our point of view, the most serious problem with these semantic data models is their underlying conceptualization of data modelling. They have not made a clear distinction between the conceptual structure (objects, relationships, and properties) and their external data representations. We will elaborate this point later on.

1.4 THESIS OUTLINE

As usual, this first chapter has given the reasons for originating the research, presented the research approach to be taken, and discussed the related previous work.

The next chapter provides a review of conventional data modelling. It focuses on identifying the semantic modelling problems associated with the relational data model and problems with some of its direct extensions. These problems set up the goal for SDBM. Chapter 3 summarizes several of the most important features of current semantic data modelling. Specific semantic data models are not discussed. These features provide some basic modelling mechanisms for the design of SDBM.

Chapter 4 gives a description of our conceptualization of data modelling. It explains the basic concepts such as objects, properties, relationships, data types, roles, and windows. The chapter also discusses the specialization mechanism based on the conceptualization and proposes some extensions to this mechanism.

In Chapter 5, an overview of SDBM is given. The overview is divided into three major parts, the SDBM type system, the SDBM window system, and the SDBM transaction system. Syntax is given informally by using prototypes. The semantics of each syntactic construct are also given informally both through examples and through explanation in plain English. A formal syntax of SDBM, written in context-free grammar, is provided in APPENDIX A.
Chapter 6 gives an example of using SDBM to design a hypothetical database. The example is highly simplified in order to emphasize the underlying principles and basic structures of the model. In APPENDIX B, we compare the database model, SDBM, and the data model, TAXIS.

Finally, Chapter 7 concludes the thesis by summarizing the contributions of this research.
2. REVIEW OF DATA MODELLING

2.1 SEMANTIC MODELLING PROBLEMS OF THE RELATIONAL DATA MODEL

There are serious semantic problems associated with conventional data models in general. For obvious reasons, the hierarchical data model and the network data model are basically oriented towards the computer instead of the human user. At the conceptual level, their modelling ability is limited. In this paper, we will not consider these two data models but focus attention on the relational data model only.

2.1.1 TUPLE ORIENTATION

The relational data model has the relation as its modelling primitive. Basically, a relation is a flat file with tuples as its rows. Although tuples are excellent for data processing purposes (primarily because of the current architecture of the computer), sometimes they are not directly related to the primitives in the human problem domain. With tuples, both the database designer and the database user may have to effect a mental transformation from human problem domain primitives, such as objects, to their computer representation, tuples. The transformation is complicated by the complex relationships between relations (or tuples) and conceptual primitives. As will be seen in the following discussion, relationships of this type are not one-to-one mappings.

For example, in a relational database, a 'real' world object may be represented in any of the following ways:

1. As part of a tuple.
2. As a single tuple.

---

*One difference between a relation and a flat file is that rows have no order in a relation while in a flat file they do. However, this difference is not significant in our discussion.*
3. As a collection of tuples within a relation.

4. As a join of tuples from two or even more relations.

This suggests that a tuple may or may not represent an object. To illustrate the implication of this fact, let us take a look at a relational database consisting of several relations each with tuples describing certain aspects of employee objects. For example, a relation may describe the employee as a supervisor and another relation may describe him as a research team member. However, the database management system based on the relational data model is not capable of obtaining and keeping the knowledge that they are just different aspects of the same employee. Therefore, it is impossible for it to carry responsibilities such as that for the cross relation consistency of the stored data.

The relational representation of properties of an object and relationships between objects suffers from the same problems.

1. A property or a relationship may be represented as part of a tuple that describes an object. That is, a property or a relationship is expressed in terms of coexistence in a tuple. For example, an one-to-many relationship is almost always represented by embedding the relationship into the tuple on the 'many' side.

2. A property or a relationship may be represented in separate tuples. For example, if COLOR is a multi-valued property of object PART, a separate relation has to be created. In fact, for any many-to-many relationship, a separate tuple must be used if normalization is desired. The worst implication of representing a relationship with several separate tuples may be that the database user might not know the existence of such a relationship. Or even if he knows the existence of a relationship, he may have difficulty in traversing relevant tuples because the relational database simply does not explicitly keep the information on this kind of relationship.
Such a non-one-to-one relationship between the relational data modelling primitives and the human conceptual primitives may also cause some difficulties for the relational database user because he is unable to directly manipulate objects and relationships between objects. The primitives he can directly manipulate are much less meaningful tuples and relationships among tuples. The desired result may be obtained as the side effect of this kind of manipulation. For example, the unit of creation and destruction is the tuple in the relational database. If an object is modelled by two tuples, then it can only be created as the result of inserting two tuples.

Notice that certain conclusions of the above discussion may be subject to disagreement depending on the interpretation of the relational representation of objects, relationships, and properties.

Also notice that because the concept of an object is not well defined (In fact, this is a concept that can't be rigorously defined), we may be able to avoid the problems identified above by adopting the view that a tuple always corresponds to an object and anything that can't be represented by a tuple is not considered an object. However, we believe that designers as well as users do have certain tendencies in perceiving objects, properties, and relationships. We should not try to unnecessarily adapt human designers and users to constraints presented in the relational data model.

2.1.2 SEMANTIC OVERLOADING

The relational data model has only one construct, the relation (and consequently, the tuple). This fact is often quoted as evidence of the simplicity of the relational data model. However, such simplicity does not always lead to a good result. Viewing it from a different angle, we will see that such simplicity can cause difficulty in interpreting a relation. This is sometimes called 'semantic overloading' (McLeod [1978]). That is, a single construct is used for many purposes:
1. To record the existence of objects.
2. To describe properties of an object.
3. To model relationships between objects.

A user may have difficulty in determining which of the above purposes a given relation is used for.

2.1.3 ATOMIC DOMAINS

By definition, the relational data model specifies that each entry of a relation can only have atomic domains. And to ensure that a relational database behaves properly, the relations are usually required to be, at least, in the third normal form. 'Atomic domain' means that the domain must be non-decomposable and non-hierarchical. Several problems are associated with this restriction.

1. Multi-valued properties are difficult to accommodate. Unfortunately, multi-valued properties are phenomena that are confronted frequently in the real world. For instance, a part may have more than one color. In order to remain in the third normal form, the relational data model either introduces an extra relation or adds more attributes to a relation.

Conceptually, the first approach is not always desirable. For example, the part relation
\[ P(\text{NAME, SIZE, SUPPLIER, COLOR}) \]

has to be broken into two relations
\[ P1(\text{NAME, SIZE, SUPPLIER}) \] and \[ P2(\text{NAME, COLOR}) \]

just because of the multi-valued nature of COLOR attribute. Conceptually, we would prefer the single-table representation. This approach also introduces additional problems associated with the referential integrity (e.g. the integrity constraint between the NAME attribute of the relation P1 and the NAME attribute of the relation P2).
The second approach has only very limited applications. First of all, we have to know the maximum number of values a property can have. Secondly, relations created in this way could be very sparse (many null values may occur). Finally, this approach is not symmetrical and natural. Conceptually, we would prefer a single multi-valued attribute to several single-valued attributes.

2. Object domain (domain whose values directly model objects) is impossible to express. For two major reasons, we would like the domain values of an attribute to directly model a collection of objects:
   a. Under certain circumstances, it is close to the way we think about certain relationships. For example, in the relation \text{STUDENT}(\text{STDNO,NAME,DEPT,YEAR}), we could treat the domain of attribute \text{DEPT} in two different ways. First, we could think of \text{DEPT} as a collection of character strings. This is, in fact, the view taken by the relational data model. Alternatively, \text{DEPT} could be viewed as a collection of department objects. If we are interested in modelling departments as objects, the latter approach is more natural.
   b. It helps the computer to precisely specify and enforce semantic integrity constraints. For example, the problem of referential integrity can be solved automatically, if object domains are allowed. The relational data model simply does not have such a capability.

3. The relational data model uses very few types of symbolic strings (e.g. integers, reals, and characters) to represent the varieties of real world properties. These few data types are static in that they have predefined operations which can not be changed. In addition, no abstraction mechanism is provided to structure domains. Therefore, desired behavioural semantics can not be attached to domains and subsequently properties. The result is that two conceptually different properties may have to have the same behavioural semantics. For
example, if the part SIZE and the employee AGE are both defined on the domain of two-digit integers, such illegal operations as AGE greater than SIZE is perfectly legitimate in the relational data model.

2.1.4 NORMALIZATION

It has been realized that despite the apparent simplicity of the relational data model, particular care has to be taken to prevent a relational database from behaving abnormally. A relational database design theory called normalization was introduced for this very purpose.

In a sense, relational normalization increases the ability to capture certain meanings of data (e.g. functional dependencies). However, conceptually it is irrelevant or even undesirable. We will elaborate this point below.

The relational normalization theory dictates that anomalies, including insertion anomalies, deletion anomalies, and modification anomalies, be eliminated through decomposition or synthesis. There are two basic principles underlying the relational normalization, the principle of equivalence and the principle of minimal redundancy. Our argument is that all these concepts are basically data-processing oriented.

1. Decomposition and synthesis are recommended because the relational data model is handicapped by its limited data structure. An application-minded user does not care what data structure the computer is using. Neither does he care whether it decomposes or synthesizes its structure. In addition, there are semantic problems associated with the relational normalization theory itself. For example, normalized relations are usually less meaningful at the conceptual level, since a relation which contains certain information of a 'real' world object is often not in normalized form and the process of normalizing such a relation tends to split it into several relations which do not correspond to the object being modelled.
2. Equivalence is also basically oriented towards the computer. The basic criterion of equivalence is the lossless join property under the universal relation assumption (Aho et al. [1977], Rissanen [1977], Beeri et al. [1980], Maier et al. [1979], Maier [1983] and Ullman [1983]). To the human user, two tables are usually not equivalent to one table.

3. Minimal redundancy is desirable at the data-processing level, but may not be so at the conceptual level. Conceptual data redundancy is often desired, because for example the same person may be viewed from different angles or at the different levels of detail. Furthermore, even for data processing purposes the relational normalization can only eliminate some kinds of data redundencies. It may in fact introduce new data redundancy.

Obviously, there is considerable lower level data processing involved in relational normalization. The key point is that the system should be designed in terms of semantically meaningful constructs so that analysis and design on the database designer side as well as interpretation and manipulation on the user side can be facilitated. The data structure manipulation for the purpose of efficient processing and internal data integrity should be left to the computer and completely hidden from the user.

2.1.5 RELIANCE ON USER-CONTROLLED IDENTIFIERS

In the relational data model, user-controlled names or symbolic references are used to convey a considerable amount of semantic information. For example, objects and relationships are all represented by symbolic names that are chosen at the user’s discretion. Whether the expected objective can be achieved depends on the database designer’s careful selection and the user’s consciousness. Kent has given an extensive discussion of this subject (Kent [1979]). See also (Date [1983]). The problems associated with reliance on user-controlled identifiers are summarized as follows.

1. User-controlled symbolic identifiers may be subject to changes. For example, a
supplier number may change for some reason. A person may not like his name any more and decide to change it. It is also quite possible to see a change of a journal name. When user-controlled symbolic identifiers are used as keys, such changes, though conceptually legitimate, are usually not allowed with the relational data model. This kind of change has to be accomplished through an insertion operation following a deletion operation. There are at least two problems with this approach. First, it is difficult for the user to find all the occurrences of symbolic identifiers for an object so that proper insertions and deletions may be done. Second, as the result of deletion and insertion, semantic ambiguities may be introduced. For example, there is no way for the system to tell if we have only changed the person's number or we have replaced him with a different person who has identical properties except his number. Such semantic ambiguities do matter when a certain report on, for example, personnel archives is to be generated.

2. Symbolic references could result in references to non-existent objects. This is usually called ‘dangling reference’ which is sometimes desirable and sometimes not. Consider for example the part-supplier case. For each part, if we only want to know who the supplier is, we obviously do not care whether the supplier information is still in the database or has been deleted from the database when, for instance, the supplier goes bankrupt. Under such circumstances, a ‘dangling reference’ does no harm.

Let us consider a different situation where the information on purchasing contracts is to be recorded. Obviously, a contract can exist only if the supplier still exists. In this case, a ‘dangling reference’ is highly undesirable.\(^5\)

\(^5\) Of course, we only consider the static state of the system to be modelled. Dynamic transitions are not within our consideration.
The problem is that the relational data model provides no alternatives for specifying such constraints.

3. The database designer can have difficulty in selecting a proper identifier for a relation or attribute because of the strong interaction between the identifiers and the perception of objects being modelled. Perception of objects can affect the choice of identifiers, and conversely identifiers can also affect the perception of objects. The latter can result in a quite different semantic interpretation of a relation or attribute.

4. Even if a choice of keys is made, it is often arbitrary on the conceptual level. This arbitrariness can create serious problems when, for example, a user view that uses an alternative key needs to be derived.

5. In reality, different identifiers may be used to represent the same object. For instance, Robert Jackson and Bob Jackson could refer to the same person. Different identifiers may even be from different domains, for example, part-name and part-#. Because the relational data model only depends on domains to control joinability, the above real world possibilities must be banned in the relational database.

2.1.6 GENERALIZATION/SPECIALIZATION

The relational data model is not an effective representation tool in some cases. This statement also applies to the modelling of the important conceptual mechanism of generalization/specialization. The problems involved can be categorized into the following two parts.

1. The relational data model can handle certain machine-controlled specialization by introducing extra relations. But this is both artificial and potentially

----------

6 Machine-controlled specialization refers to specialization that can be obtained through evaluating some expressions, for example, first order logic expressions, against a group of objects, usually in a relation.
problematic when data integrity is concerned. In fact, as the number of specializations increases, the approach can result in unbearable data redundancies. For this reason, the concept of relational view may be used. However, the view concept can only provide limited help. For example, it is impossible to model the additional specialties of a subset with the relational view.

2. It will be very difficult for specialization that can only be specified and controlled by the user to be modelled by the relational data model. User-controlled specialization is often necessary. For example, in the PART-SUPPLIER case, we may want to have some extra information about wholesalers. Probably only human beings can understand what is meant here by 'wholesaler'. Therefore, we have to manually classify it. But at the same time, we still want the computer to maintain the semantic integrity constraint that a wholesaler is a supplier.

2.1.7 REQUIREMENT OF HOMOGENEITY

Kent identifies two assumptions underlying the relational representation, namely horizontal and vertical homogeneity in data (Kent [1979]). Horizontal homogeneity refers to the situation where each tuple of a given relation contains the same fields; and vertical homogeneity refers to the situation where in a relation a given attribute allows only the same kind of information in each tuple. For a real world user, these constraints are undesirable, because in reality counter examples are abundant.

For instance, a patient relation may have an attribute PREGNANT-OR-NOT. For a male patient, we obviously do not want this attribute at all. On the other hand, it does not make much sense to have different relations for male and female patients. It is worth noticing that in this case, permitting null values might cause interpreting difficulty. It becomes more problematic when there is nonhomogeneity associated with an attribute.
which we might want to use as an identifier. For example, social security numbers may often be used as employee identifiers. But in a multi-national corporation, some employees might not have social security numbers (Kent [1979]).

Vertical nonhomogeneity is also a common phenomenon. For example, company cars can be assigned either to employees or to departments. Here employees and departments quite possibly have different formats of identifiers. The relational data model can handle this situation in a number of ways such as adding a new attribute, creating a new relation, creating new uniform identifiers, etc. However, none of them provides a good conceptual solution. For example, if we use two attributes, one for employees and another for departments, we not only introduce data integrity problems but also create a representation structure which bears little resemblance to the conceptual structure of what is being modelled. More awkward problems might arise when we try to assign cars to a new unit with an even different identifier format. For a more detailed discussion on this subject, see (Kent [1979]).

2.1.8 AGGREGATION

The relational data model is not a very effective tool when we want to model another important data abstraction process, aggregation. For example, when recording data flow diagrams obtained from information needs analysis, we obviously want to preserve the relationships between different levels of a data flow diagram. Here, a level(n) diagram can be viewed as an aggregation of several level(n+1) diagrams.

With the relational data model, a special relation

\[ R(\text{DIAGRAM},\text{SUBDIAGRAM}) \]

may be created. In each tuple of this relation, SUBDIAGRAM is an immediate subdiagram of DIAGRAM.
There are at least four difficulties associated with this approach. First of all, the modelling process is not direct and explicit. All the relationships between different levels are hidden in data. Secondly, specialities that may be associated with different levels are difficult to accommodate. In our example, no such difficulty arises because diagrams at all levels are assumed to have homogeneous attributes. However, it is common that each level of aggregation has its own unique attributes. Thirdly, certain semantic integrity constraints cannot be maintained by the relational data model. For example, if A is a subdiagram of B and B is a subdiagram of C, then A is a subdiagram of C. This constraint can be easily violated. Finally, in certain cases information retrieval will be inconvenient. For example, it is awkward to express the query, 'find all the subdiagrams of diagram A'.

To conclude the discussion of semantic modelling problems of the relational data model, it should be mentioned that the above list is not exhaustive in any sense. The discussion could be expanded by further elaborating the listed points. For example, a further discussion can be made about problems associated with composite primary keys. This paper has neither enough space to accommodate them all nor the intention to do so. Moreover, it should also be mentioned that the above criticisms of the relational data model may not be so sound when looked at from some different angles. Some of our criteria such as simplicity and naturalness are generally subjective and may subject to different interpretations by different people.

2.2 SEMANTIC EXTENSIONS TO THE RELATIONAL DATA MODEL

Ever since Codd published his article 'A Relational Model of Data for Large Shared Data Banks' (Codd [1970]), the relational data model has been subjected to the substantial scrutiny of database researchers. There are not only considerable criticisms as presented in the previous sections but also many extensions which aim at enhancing the semantic
modelling power of this data model. In this section, I will briefly summarize this work by commenting on how these extensions achieve this goal and what the problems associated with the approaches are. This section will only cover the direct extensions of the relational data model. By ‘direct extension’, we mean the basic construct of the extended data model are still relations and tuples. The next chapter reviews semantic data modelling in more detail.

### 2.2.1 EXTRA CONSTRAINTS

To some extent, the relational data model has the above problems because the model is too general. In order to use such a general model to capture the great variety of the real world, the database designer has to sometimes make very subtle and difficult choices among many possible alternatives. Similarly, the database user usually faces difficulties in correctly interpreting the intension of the database. Therefore, a natural way of solving the semantic integrity constraint problems of the relational data model appears to be to add some kinds of additional semantic integrity constraint mechanisms to it. In fact, much work has been done in this direction (Stonebraker et al [1974], Eswaran [1975], Astrahan [1976], Hammer & McLeod [1975], Hammer & McLeod [1976]).

In this type of approach, integrity constraints are specified by a predicate assertion facility. Assertions can be stated in the database schema, and enforced automatically by the DBMS. For example, in System R (Astrahan et al. [1976]) there is an integrity constraint subsystem which uses the query language SQL to express constraints. The system can handle both static restrictions, e.g. attribute SEX can only has two values ‘F’ and ‘M’, and dynamic restrictions, e.g. the specification of a trigger.

Although this approach proves to be very powerful and flexible in dealing with some semantic integrity problems, it can not solve many other problems of the relational data model such as those associated with the tuple representation. Furthermore, it is a
piece-meal approach. Constraints and the data model are separated. This may lead to unnecessarily increased conceptual difficulty in using the model. A better approach would be to let more constraints be directly expressed in the data model.

2.2.2 SEMANTIC CLASSIFICATION OF RELATIONS

The second approach is characterized by its attempt to classify relations so that some semantics can be clearly captured. Schmid & Swenson [1975] and Wiederhold [1977] investigated the classification of relations into the following categories:

1. Those that describe the existence of objects (autonomous objects).
2. Those that model multi-valued characteristics of objects.
3. Those that capture relationships among objects.

This approach explains various ways in which relations can be associated with semantic primitives. However, as pointed out by McLeod [1978], it does not really provide any guideline as to how relations should be used, nor does it extend the power of the relational data model in capturing more semantic information. Moreover, it does not eliminate any previously identified problems at all.

2.2.3 THE ENTITY-RELATIONSHIP APPROACH

Another important extension of the relational data model is the work done by Peter Chen [1976]. His model is called the entity-relationship model. The original ER model has been traditionally viewed as a direct extension of the relational data model. However, it would be more helpful to view it from two different angles, 1) its data structure and 2) its conceptual view. As far as the data structure is concerned, there is nothing significant about the model. Except for some minor differences, it is very close to the proposal of Schmid & Swenson [1975]. However, the significant part of the ER model is its underlying conceptual view which is profound and far-reaching. It is this conceptual view that has stimulated considerable interest in the entity-relationship approach. To some extent, many
recently proposed semantic data models can be directly or indirectly linked to the ER model. It is also very interesting to notice that the ER model, originally proposed as a database tool, has gained wide acceptance in system analysis and design (Chen [1981]).

The major problems associated with the ER model are identified as follows:

1. *Table Representation.* The structure of the ER model is the same as the relational data model, namely a two-dimensional flat table. Therefore, problems identified for the tuple representation still exist.

2. *Rigid Distinction between Objects and Relationships.* The ER model explicitly defines relationships and prohibits them from participating in further relationships. This distinction is neither necessary nor convenient.

3. *Lack of Richness.* The ER model in its original form is a good database design aid, but not rich and flexible enough. For example, it does not allow attributes to be viewed as relationships. It does not allow entity sets to be attribute domains, either. In fact, the value of entity attributes can only be taken from so-called value sets which contains atomic symbolic strings.

2.2.4 *CONCLUSIONS*

In this section, we have briefly reviewed some direct extensions of the relational data model. In conclusion, although this earlier research has contributed a great deal to semantic data modelling, it is handicapped to some extent by its close tie with the relational data model. None of them is sufficient to be used as a satisfactory conceptual modelling tool to capture the semantics of real world applications.
3. REVIEW OF SEMANTIC DATA MODELLING

This chapter is devoted to a discussion of semantic data modelling. The discussion does not focus on individual semantic data models, but rather on the basic concepts and principles that are common to many of these data models. In the following, these concepts and principles are discussed in turn.

3.1 BASIC GUIDELINE

The human conceptual primitives are objects, properties, and relationships between objects. Database design is a transformation from the conceptual representation of an application into structures particular to a data model. Using a database is also a transformation but in the reverse direction. One of the objectives of the data modelling activity is to reduce these mental transformation as much as possible.

3.2 OBJECT ORIENTATION

In semantic data modelling, 'object' is considered the most basic modelling construct. An object is any thing that interests us. In the data modelling literature, different terms have been used to mean the same thing. Abrial [1974], Berild & Nachmens [1977], Smith & Smith [1976], Shipman [1981], and Brodie [1984] use the term 'object'. Chen [1976], Kent [1979], and Codd [1979] use 'entity', while others choose the term 'token', for instance in (Mylopoulos et al. [1980]), and (Tsichritzis & Lochovsky [1982]). The term 'object' is used in this paper as the most basic modelling construct. The term 'entity' is used to mean the real world thing to be modelled.

Based on the observation that objects may be different in nature and in their implications for data processing, researchers generally classify objects into categories. Abrial
distinguishes between concrete and abstract objects (Abrial [1974]). A concrete object represents a physically existing thing, for example, a part, a supplier, or an employee, while an abstract object represents a concept such as date. Chen makes the distinction between a regular and a weak object (Chen [1976]). A regular object may exist independent of any other objects, whereas the existence of a weak object is dependent on the existence of some other object (or objects). The data processing implication is that the deletion of a regular object may cascade to cause deletion of all the related weak objects. In (Codd [1979]), Codd proposes to classify objects into characteristic, associative, and kernel objects.

To human beings, the existence of an object is manifested by its properties which are the characteristics of that object. If we define attributes as characteristics of a real world entity, properties are their conceptual modelling counterparts. One important principle is that there should be a one-to-one correspondence between entities of the real world and objects in the database.

We should emphasize that in this paper the term ‘object’ is used as a modelling construct. An object is an entity of the modelling world. The principle of one-to-one correspondence, which cannot be over-emphasied, is inherent in this definition of objects. This principle indicates that for an entity in the ‘real’ world, there should be correspondingly in the model one object representing that entity. Moreover, each object in the model must not represent more than one entity in the ‘real’ world. Finally, this principle is being discussed only at the conceptual level. Physically, certain redundancy may sometimes be desirable. For example, in order to achieve optimization of different access strategies, multiple copies of the same data may be required. Of course, such redundancy must be well controlled so that consistency and synchronization are ensured. The implementation of this one-to-one principle makes it possible to eliminate some very serious integrity problems such as insertion and deletion anomalies, and to some extent
the referential integrity problem, of the relational data model.

Another important concept of the object orientation is the object identity. While in a set-oriented data model such as the relational data model two objects (tuples) can only be distinguished by some intrinsic properties, object identity allows any object to be referenced directly as a unit. An important implication is that an object is not the same as its symbolic name. This avoids the problem of unchangeable unique identifiers and a collection of other serious data modelling problems. We will come back to this point later in Section 4.2.3. A further consequence of object identity is that relationships are asserted between objects instead of their symbolic identifiers. This contributes to the elimination of the referential integrity problem, for example, by specifying that an object cannot be deleted until it no longer participates in any relationships.

As a final comment on the object orientation, it should be clarified that although object orientation is thought of as a semantic data modelling principle, this does not mean that conventional data modelling does not deal with objects. The difference between conventional data modelling and semantic data modelling is that the latter manipulates objects directly in the models, while the former does it indirectly through constructs such as tuples. The data abstraction aspect of semantic data modelling is discussed in the following section.

3.3 DATA ABSTRACTION

Data abstraction is a process in which irrelevant details are suppressed and details relevant to the application being modelled are retained. Relevant details can be understood either with respect to the applications at hand or to a particular group of users. Clearly, different applications need different details. Different users also need to access different levels of detail. These needs have undoubtedly motivated the explicit application of data abstraction in databases. Furthermore, like any other abstraction process, data abstraction is
a very effective way of reducing the mental load on both database designers and users. As soon as we understand a group of phenomena, we abstract the relevant information from them and then represent such abstraction with a concept which is usually identified with a symbolic name. Thereafter, we only need to refer to this symbolic name instead of going back through all the detailed information. The basic mechanisms that are used in most semantic data models to provide data abstraction are: classification, aggregation, generalization, and association⁷.

3.3.1 CLASSIFICATION

As mentioned before, objects are used as basic constructs to model the real world. It is noticed that objects may share some commonalities, particularly some common properties. Based on these common properties, we can group objects into categories --- object classes, or classes for short. This grouping process (mechanism) is called classification. Precisely, classification is defined as a form of abstraction in a collection of objects, which is considered as a higher level object. By classification, the relevant common properties shared by objects are kept by the type of objects in a class while all the other properties of these objects are ignored. Classification establishes an instance-of relationship between a particular collection of the objects currently existing in the database and a class denoted by the name of the class.

The term 'class' has several synonyms such as 'entity set', 'entity class', 'entity type', 'object type', etc. Unfortunately, 'class', 'set', and 'type' do not usually mean the same thing. To avoid any confusion I will use 'class' through this paper to mean a collection of classified objects⁸. One of the distinctive features of classes is that they are persistent.

---

⁷ This is not to be confused with the association between two sets. Association as an abstraction mechanism will be explained in Section 3.3.4.

⁸ The concept of class defined here can be found in various semantic data models proposed recently. As will be seen in the next chapter, the class concept is less powerful than the SDBM window concept in terms of data modelling ability.
Namely, they exist even after a program invocation has terminated.

Some semantic data models also allow classification to be applied to classes. The result is a metaclass. A metaclass is an abstraction of a collection of object classes which share some common properties. Note that the properties that are to be abstracted belong to an object class rather than instances of a object class. An example of a metaclass is CHARACTER-STRING with object class ALPHBETIC-STRING and object class NUMERIC-STRING as its instances. Notice here that a specific character string is not an instance of CHARACTER-STRING as defined.

3.3.2 AGGREGATION

Sometimes a relationship among several objects may become so important that we would rather ignore these objects and refer only to the relationship. Clearly, this is an abstraction process. It is usually called aggregation. Formally, aggregation is defined to be a form of abstraction in which a relationship among objects is regarded as a higher-level object, with lower-level details suppressed.

Thus, with this abstraction, an object may be viewed in two different ways: first as a relationship between an identity and a set of lower-level objects (attributes) and second as an object in its own right (Date [1983]). Some refinements of aggregation are made in semantic data models. There are basically three forms of aggregation (Codd [1979]).

1. *Cartesian aggregation* which views an object as an aggregate of properties. For example, object STUDENT may be considered to be an aggregate of properties STUDENT-NO, NAME, SEX, and ADDRESS. Aggregation of this kind is originally addressed by (Smith & Smith [1977]).

2. *Part aggregation* which views an object as an aggregate of its parts. For example, a PC may be viewed as a part aggregation of a monitor, a keyboard, a CPU and a disk drive. This aggregation can be applied recursively to
generate a part aggregation hierarchy. Part aggregations are primarily useful to maintain relationships among objects, i.e. part hierarchies, whereas Cartesian aggregations focus on describing objects by their properties.

3. **Cover aggregation** which refers to the grouping together of a collection of possibly heterogenerous objects into a higher-level object, in accordance with some kind of membership criterion (Date [1983]). For example, a task force may be viewed as a cover aggregation of ships, planes, tanks and personnel. There are two important characteristics of cover aggregation. First of all, the structure of its members is not well defined. This is unlike Cartesian aggregation where the structure of its members is explicitly specified in the definition of aggregation. Secondly, the grouping may or may not be machine-understandable. For this reason, the Semantic Data Model calls it user-controllable grouping (Hammer & McLeod [1979]).

There is still another kind of aggregation, statistical aggregation, though it is somewhat different from the aggregation concept defined here. This concept is usually supported in most data models by providing some built-in functions, e.g. MAX, MIN, etc.

**3.3.3 GENERALIZATION/SPECIALIZATION**

Generalization is a form of abstraction in which a set of similar objects is also considered at a generic level. With the generalization mechanism, an object may also be viewed as a generic object. This abstraction establishes an is-a relationship. By generalization, many individual differences between objects are suppressed. In set theory, this is similar to generating a superset. For example, object class DOCTOR and NURSE can be generalized to a more generic object class STAFF, if we are not particularly interested in their differences.
Generalization can be repeated to specify a generalization hierarchy, also called subclass hierarchy when the generalization mechanism is applied to object classes. This hierarchy corresponds to one of the important ways we acquire and process our knowledge. Therefore it is not surprising that generalization should be one of the inherent properties of data modelling. However, the conventional data models fail to provide a rigorous and consistent mechanism for modelling this hierarchy. This is especially true of the relational data model which almost totally ignores the importance of generalization. Theories like normalization focus exclusively on Cartesian aggregation.

In database design, it is the concept of specialization, the reverse of generalization, that is widely used to 'generate' new object classes. Because a database application is often well-bounded, we therefore can work downwards, i.e. from generic concepts to more specific concepts. The term 'generalization' and 'specialization' are distinguished only to indicate the order in which relevant classes are defined. They have no different implications for the conceptual structure of the database. In other words, generalization and specialization are exactly symmetrical with respect to the conceptual structure. Several very important characteristics are inherent in specialization. There are at least the following four:

- The extension of the specialization is a subset of the extension of the generic class.
- The specialization inherits all the properties from the generic class. For example, if class PERSON has a property called height, then class CHILD will have property height too, if class CHILD is specified to be a specialization of class PERSON.
- Values of inherited properties must be specializations of the generic property values. For instance, if values of PERSON's height are specified to be less than 10, then values of CHILD's height must also be less than 10.
- The specialization may have additional properties. For instance, class CHILD has a GUARDIAN property which class PERSON may not have.
3.3.4 ASSOCIATION

This is a concept introduced by M. Brodie\textsuperscript{9}. Association is a form of abstraction in which a collection of member objects is considered as a higher level set object. By association, the properties of a set object are explicitly differentiated from the properties of its member objects.

For example, each member of object class STUDENT has property SNAME, STD#, and AGE. At the set level, the set object STUDENTS may have property STUDENT-REPRESENTATIVE and AVERAGE-AGE which are related to its member properties but do not exist at the member level. Here we say that, in this respect, STUDENTS is an association of student objects. Another example is that set object PEOPLE is an association of object instances of class PERSON. Here, PEOPLE may have properties such as POPULATION which is related to the set of objects as a whole, whereas PERSON have properties directly related to each individual person objects.

In this section, I have briefly reviewed four basic concepts of data abstraction, namely classification, aggregation, generalization and association. These concepts provide useful or even necessary constructs for a powerful data modelling tool --- semantic hierarchy.

3.4 SEMANTIC INTEGRITY CONSTRAINTS

Another important aspect of data models is what and how semantic integrity constraints are modelled. Data models differ dramatically in the provision of means for representing semantic integrity constraints. This section reviews major semantic integrity constraint issues in semantic data modelling.

\textsuperscript{9} He claims that sets are fundamental modelling concepts. Association is one of three primitives of his set concept (Brodie [1983]).
A database can be viewed as a repository of data values and a database schema can be viewed as a collection of constraints that restrict values that may exist and changes that may happen in the database. Therefore, to some extent, a data model is nothing but a methodology for specifying constraints. The following discussion will be pursued from this viewpoint. As will be seen, this view clarifies the difference between conventional data modelling and semantic data modelling.

In conventional data modelling, database schema and integrity constraints are separate concepts. Usually, a database schema is more concerned with accurately reflecting the real world situation, while integrity constraints restrict the possible database states that can be generated from a given schema to those that meet the constraints. The former is more concerned with organizing modelling constructs into meaningful structures. In this paper, the term 'semantic integrity constraint' is used to refer to any database constraint that ensures accuracy, correctness, or validity of data in the database.

With a specific data model, different semantic integrity constraints may have different relationships with the data model. Some are the inseparable part of a data model. They are called inherent constraints of that data model. For example, that tuples in the relational data model must be unique within a relation is an inherent constraint. Constraints that must be specified using some kind of constraint mechanisms are called explicit constraints. For example, constraints that are specified with certain first order logic assertions are explicit constraints. The third kind of constraints are implicit constraints. An implicit constraint is one that can be derived from either inherent and/or explicit constraints.

Viewing a data model as a methodology for organizing semantic integrity constraints, we see that one major difference between data models lies in what and how semantic integrity constraints have been built into data models. The relational data model has very

\[1^{10}\] In some literature, this kind of constraints are called semantic integrity constraints.
few inherent constraints. Most semantic integrity constraints have to be externally specified. Often, many kinds of semantic integrity constraints cannot possibly be specified.

To overcome the modelling shortcomings of the relational data model, researchers have proposed many external semantic modelling facilities. We have seen examples of this kind in INGRES and System R.

With most semantic data models, however, more data semantics are captured by incorporating in the semantic data models a greater variety of inherent constraints. Of course, there is a trade-off in doing so. On one side, we are increasing our ability to model certain data semantics. On the other side, we are narrowing the modelling realm.

Several types of semantic integrity constraints are of special interest to semantic data modelling. They are domain constraints, cardinality constraints, uniqueness constraints, referential constraints, and inheritance constraints.

A domain constraint refers to the restriction on values that a property can have. In the literature, the term ‘type constraint’ is also used. Almost all the data models prefixed with the adjective ‘semantic’ provide facilities to enforce these kinds of constraints. The basic mechanism employed by these models is to provide additional levels of data abstraction between properties and primitive types such as integer and character. In this way, underlying domains can be associated with distinct semantics.

In database literature, a cardinality constraint usually refers to a restriction on the cardinality of relationships between classes. Cardinality is the number of objects in an class that can be related to an object in another object class and vice versa. This type of constraint is often extensively studied in semantic data models that directly deal with binary relationships, for example, in the Binary Semantic Data model (Abrial [1974]). It is easy to show that uniqueness constraints are a special case of cardinality constraints.
Referential constraints refer to the restriction on 'dangling references'. This restriction requires that if an existing object has referenced other objects, then all referenced objects must exist. 'Dangling references' may be caused either by insertion of a new object or by removal of an existing object. Upon insertion, all the referenced objects must be already in the database, and upon removal, the object must not being referenced by any object. Otherwise, insertion and deletion should not be allowed to take place. This constraint is also called the existence constraint (Codd [1979]) or dependency constraint (Chen [1976]). In various extensions to the relational data model, some referential integrity rules have been developed to handle this type of constraints (Codd [1979], Date [1981]).

An inheritance constraint refers to inheritance properties of data abstraction. For example, each subclass inherits all the properties of its superclass. This kind of constraint is automatically enforced in most recently developed semantic data models, such as TAXIS and Galileo. In these models, the constraint is specified whenever the database structure is specified.

The approach taken by semantic data models in modelling constraints is considerably different from the conventional data models. Most semantic data models present modelling primitives, structures, and constraints within an integrated framework, while the conventional data models usually approach them separately.

3.5 APPLICATION OF DATA TYPES

In recent years, more and more attention has been paid to the application of data abstraction capabilities developed in programming languages to the database management system. Systems such as TAXIS (Mylopoulos et al [1980]), Galileo (Ablano [1985]), RIGEL (Rowe [1979]), and INGRES (Stonebraker et al [1976]) all provide some applications of data types. This section gives a brief review of such applications.

\[1 \text{When we say 'A is referenced by B', we mean that to maintain integrity of a database, A must exist if B exists in the database.}\]
3.5.1 USE OF THE DATA TYPE CONCEPT

The concept of the data type was originally developed in programming languages for the purpose of improving data reliability, program readability, program verification, and processing efficiency. The concept has been considered fundamental in the development of programming languages, basically because it is a very effective abstraction mechanism. Recently, a considerable amount of research has demonstrated that with some extensions this concept, especially the concept of abstract data type, can apply directly to the database context. The primary uses of the data type concept in the database context are:

- for the specification of a database. The concern of database structures is centered around data and their abstraction. Abstraction mechanisms such as generalization, aggregation and encapsulation\(^1\)\(^2\) can be provided by data types. At the same time, data types also specify that certain operations are to be provided for data objects. One of the many advantages of applying data types to database specifications is that certain semantic integrity constraints, e.g. domain constraints, can be defined uniformly. For example, a hierarchy of data abstraction can be established with desirable semantics being associated with each participating data type. Another distinctive advantage of applying data type concept to the database design is that the behavioral semantics, in terms of specific operations, embedded within a data type, can be integrated with the structural semantics of a database design. From this point of view, abstract data types are those data types with behavioral semantics which can be specified by the designer.

- for the verification of a database specification. To prove that a database specification includes the desired structure and has no inconsistencies has not been an easy task. The use of the data type concept makes it possible to apply verification techniques developed in programming languages. Such techniques include theoretical models like

---
\(^1\) By encapsulation abstraction, we mean that the user of the abstraction does not need to know the hidden information in order to use the abstraction and he/she is not allowed to directly use or manipulate the hidden information.
operational semantics, denotational semantics, and axiomatic semantics. Two examples of such application are the work done by Brodie [1978] and the work by Wong [1981].

for the validation of a database. The data type concept provides a means to relate constraints and data objects. Typechecking techniques associated with data types have proven to be one of the best mechanisms for ensuring database correctness.

3.5.2 DATA TYPE STRUCTURES

Data type structures provided by different semantic data models differ significantly from each other. Some offer only very primitive types such as integers, character strings, booleans, and enumerations. A few examples are Taxis, Dial, and Gem. Others such as Rigel, Adaplex, and Galileo provide a richer set of data types. All these models support the notion of class which, as mentioned before, is a special data type associated with a persistent extent, namely an associated collection of objects.

In a recently published paper (Buneman & Atkinson [1986]), Buneman and Atkinson have argued that type and class may be separated to provide more general data models. They also pointed out that many recently developed data models have not maintained a clear separation of this kind. Separation of types and classes indicates that a data type may have multiple extents. This is often desireable in many situations. However, such separation of types and classes is usually not supported by semantic data models. This is a serious flaw in many semantic data models. They do not support the fact that a role can have several types and a type can be used to model more than one role. That is to say there could be a many-to-many relationship between roles and types. It will become more clear in the discussion of our conceptualization of data modelling.

---

13 The concept of role is explained in Section 4.2.3.
4. THE CONCEPTUALIZATION OF DATA MODELLING

In this chapter, we explain in detail our conceptualization of data modelling. This conceptualization will be applied to the design of a semantic database model, SDBM, which will be specified in chapter 5. In section 4.1 of this chapter, we give a statement of the underlying philosophy of our approach. The statement is presented without any explanation, because the rest of this chapter is, in fact, the elaboration of this statement.

4.1 THE UNDERLYING PHILOSOPHY

The 'real' world exists in the database.

4.2 THE TWO-VIEW CONCEPTUALIZATION

In the previous two chapters, a review of the data modelling practice was provided. We discussed both conventional data modelling (relational data modelling in particular) and semantic data modelling. The ability of conventional data models to model the 'real' world was analysed. In the analysis, we assessed data modelling shortcomings of the relational constructs and modelling mechanisms. A natural question to ask therefore is "With respect to these shortcomings, is there a more fundamental cause or is it just because some particular constructs or mechanisms are inadequate?" Tracing these shortcomings to their roots, we find that most of the shortcomings stem from the inadequate conceptualization of data modelling, which is embedded in these models and their corresponding modelling methodology. In other words, it is not some of their particular building constructs but the fundamental principles that are to blame.

Before further explaining this assertion, let us describe what our two-view conceptualization means.
4.2.1 A GENERAL DESCRIPTION

Conceptually, there are actually two different aspects involved in data modelling. On one side, we want the database to resemble the 'real' world as closely as possible. The 'real' world is considered to consist of a set of objects, their properties, and the relationships among these objects. These fundamental concepts and their differences from the identically-named concepts in previously developed data models including both conventional data models and semantic data models will be discussed in detail below. The guiding principle is that these fundamental concepts must be directly modelled in the database. On the other side, we should realize that we are constrained by the current technology of the computer. We have to represent these concepts in the computer. Unfortunately, we can not accomplish this without using various kinds of symbolic strings. In other words, symbolic strings are absolutely indispensable for the description of objects, properties, and relationships and for input and output between the user and the computer. However, no matter how they are used, the concepts of objects, properties and relationships and their representation with symbolic strings must be clearly separated. The separation must not depend on the consciousness of the designer and the user. Rather, it must be built into the data model as an inherent part of the data model so that many constraints which are natural in the 'real' world can be as naturally modelled.

To further clarify our conceptualization, we will discuss the relevant concepts in detail and present our interpretation of several existing semantic modelling mechanisms.
4.2.3 SDBM OBJECTS

The concept of an SDBM object is basically the same as it is defined in other semantic data models. Naturally, the fundamental principle is that there should be a one-to-one correspondence between entities in the 'real' world and objects in the database. The rest of this section is structured in such a way that several aspects of the SDBM object concept will be discussed. They include the existence of objects, the naming of objects, and the type of objects. Because it is difficult to present the discussion on these aspects sequentially, the discussion may be better understood after the reading of the entire section.

To discuss the type of an SDBM object, let us first describe the concept of role. The term role has been used in data models with various different meanings. Codd used it to describe the way in which a relation attribute relates to a domain (Codd [1970]). Chen used the term to describe the function that an entity plays in a relationship (Chen [1976]).

The role concept in SDBM is similar to that used by Bachman and Daya [1977]. It has a similar meaning when it is used in the theatrical context. A role is an abstraction of certain properties of a set of objects. There is a many-to-many relationship between roles and objects. That is, a role can be played by various objects and a specific object may play many roles at the same time. All the roles played by an object characterize that object. In SDBM, objects are not directly related to each other. Rather, they are interrelated through their roles. For example, consider that an object plays a student role and another object plays a course role. The two objects can be interrelated through these two roles, e.g. the student takes the course.

Now, let us go back to the consideration of the object type. The concept of object type has been used in semantic data models primarily for the purpose of reducing
complexity and increasing understandability. Unlike any other data models, SDBM objects have no type. This conceptualization is supported by the following considerations:

- We believe that this is a more natural way to look at objects. In most existing data models, objects and roles have been mixed up. For example, we could have an Engineer object type, an Employee object type, a Manager object type, and a Shareholder object type. But in fact they are just different roles. An object may assume all these roles.

- The abstraction function and the constraint specification function provided by the object type concept can be obtained through using the role concept. We will demonstrate this contention in Chapter 5 when we present the design of SDBM.

- There is a restriction in other semantic data model that an object may belong to several object classes but these classes must be in a same class generalization hierarchy. Our conceptualization of objects pinpoints this restriction and highlights its undesirability in modelling data.

Now, we turn to discuss the existence of an SDBM object. The role played by an object has been conceptualized differently from the object itself. We should also clarify the relationship between the role of an object and the existence of that object. They are two different notions, although in SDBM the existence of an object is manifested by the roles played by that object. In a sense, the role concept does indirectly model the existence of SDBM objects.

From a data modelling point of view, without playing any role, an object simply does not exist. For example, an object may appear to be a person, a student, and an instructor. The person, the student, and the instructor are all roles. Obviously, in reality an entity still exists even without playing these roles. But in data modelling, for practical purpose, an object needs to play at least one role to exist, which in fact implies that we are still interested in the corresponding entity.
Another characteristic that ought to be emphasized is the uniqueness of SDBM object existence. We have limited our discussion exclusively to the conceptual level. In fact the entire paper is presented at this level. At this level, a specific object can have only one existence. Multiple existence is not allowed.

From this uniqueness constraint, we can infer that in SDBM two objects are never the same, even though they may play identical roles and their external data representations may be exactly the same. This conceptual constraint, when implemented, can lead to the elimination of certain data duplications in the database. This uniqueness constraint also leads us to clarify our conceptual views of differentiating two objects. We maintain that any two objects must be conceptually differentiatable. Syntactically, however two SDBM objects may not be differentiated, for sometimes we do not need to make syntactic differences between two objects. However, we know that they are different objects. The conceptual differentiation provides us with the potential to syntactically distinguish them in the future.

Physically, an SDBM object may be represented with an internal character string which is invisible to the database user and unique for each object in the entire database, or they may be realized through some other means. These internal identifiers or any other physical constructs should not be accessed by the user.

The existence of an SDBM object starts with an explicit instruction of object creation. The act of creation also assigns a role to the object at the same time. Without binding to a role, an object can not be created. Once an object is created, it can then be assigned to various permitted roles. A role played by an object can also be de-assigned when the object no longer plays that role. The existence of an object is terminated with an explicit destruction instruction.
Now, we discuss the naming of SDBM objects. We have identified in chapter 2 that many pathological implications in the relational data model are caused by confusion of these two concepts, objects and their naming. In SDBM, we maintain an explicit distinction between objects and symbolic strings. To make our point clear, we will discuss issues of SDBM object naming at three levels.

At the internal level, an SDBM object may be represented with an internal symbolic string. Notice that such symbolic strings are significantly different from external symbolic strings. Firstly, they are invisible to the database user and can not be accessed by the database user. Secondly, they are generated automatically by the computer and are unique for each object in the entire database. Thirdly, their structure, syntax, etc. are irrelevant at the other levels.

At the conceptual level, an SDBM object has no name. That is, we do not bind an object to a specific symbolic string throughout the life time of the object. By not doing so, we isolate an object from any possible relationships it may be participating in and from any roles it may be playing. We also separate an object from its properties.

At the external data representation level, an SDBM object can be identified with any number of legitimate symbolic strings. These symbolic strings are bound to various syntactic forms. For example, an object may be identified by a number, an alphabetic string, or a record. The structure of symbolic identifiers for an object is differentiated from the object itself. Notice that here multiple identifiers for an object do not necessarily refer to synonyms as often mentioned.

In the rest of this paper, we may loosely call an object playing the role of a student a student object. Because an object can play several roles at the same time, it may have several different 'names'. However, this is only for expository purposes.
4.2.3 SDBM RELATIONSHIPS

Another fundamental concept is the relationship between two SDBM objects. This section discusses this concept. The next section will discuss the concept of object properties which can be viewed as a special type of relationship. Like the SDBM object concept, the fundamental principle of the one-to-one correspondence applies to the SDBM relationship concept. This section focuses on modelling of different types of relationships, their existence, and their representations.

We first consider the issue of n-ary vs. binary relationships. The criteria for deciding which is more suitable as the basic modelling construct are modelling simplicity and modelling accuracy, even though they are highly subjective. On one side, it is argued that because the binary relationship is the most primitive concept it is the most appropriate for conceptual modelling (Braochi et al [1976]). On the other side, the n-ary relationship is favored because of its conciseness. For instance, the relational data model uses n-ary relationships as its basic construct.

The major disadvantage with the binary relationship approach is that sometimes we are forced to introduce arbitrary relationships which we are not interested in. Consider, for example, the relationship where a student takes a course taught by a professor. In order to model this relationship with the binary relationship, we may arbitrarily break it into two binary relationships. For example, one may be that a student takes a course, and the other may be the relationship between the first relationship and a professor.

In SDBM, we adopt the n-ary relationship but with the restriction that the n-ary relation is primitive. An n-ary relationship is said to be primitive if we are not interested in the smaller relationships that can be used to reconstruct the n-ary relationship. This approach is similar to the approach characterized by "irreducible relations" (Hall et al [1976]).
We believe that a data model should be able to model relationships with different characteristics differently. One important characteristic of relationships is whether they have properties of their own. A binary relationship without its own properties can be modelled in three different ways. For expository purposes, let us again use the example that students take courses. One way to model this relationship is to treat the relationship as if it were a property. That is, the student object will have the property COURSE which identifies courses taken by the student, and the course object will have the property STUDENT which identifies students enrolled in the course. Obviously, these 'properties' are multi-valued.

Relationships with different kinds of functional characteristics (e.g. one-to-one and many-to-many) can all be modelled this way.

A very important conceptual constraint is that for such a relationship to exist both the student object and the course object must exist. SDBM has provided the proper mechanisms to ensure that this constraint is observed.

Another way to model the relationship is to treat it as an SDBM object, and then use the first approach to establish the relationship between this object and the student as well as the relationship between this object and the course object.

Obviously, it does not make much sense to model the relationship in such a way if it does not have its own properties. It simply complicates data manipulation without any gain.

The third approach is similar to the second approach. But with this approach, no object will be created to represent the relationship. In fact, this approach only creates a window through which the relationship can be seen. This approach will be clarified after the window concept is discussed. It will be demonstrated that this approach and the

---

1 The SDBM concept of window will be explained in Section 4.2.6.
second approach also have different implications for database manipulation.

For binary relationships with their own properties and any n-ary (n>2) relationships, the second and third approach are used. When the second approach is used, we in fact represent the relationship with several binary relationships between each participating object and the relationship appearing as an object or a pseudo-object in the case of the third approach. However, we are not interested in these binary relationships themselves, because the relationship of interest has been modelled with SDBM objects. With respect to this approach, we can view all the relationships in SDBM as modelled with binary relationships.

As mentioned previously, the existence of the SDBM relationship depends on the existence of the related objects. This necessary condition must be checked by the computer. To establish a specific relationship, we must explicitly modify appropriate properties of participating objects. These properties are those which have been dedicated to representing the relationship. A relationship can be terminated through modifying appropriate properties, too.

It is important to be aware of the conceptual difference between the existence of an object and the existence of a relationship.

We believe that modelling the relationship through object properties is natural. Take the student object as an example. We would naturally think that he has a name, he has a student number, he has an age, he has degrees, and he takes courses. Although the computer must distinguish courses from the name, age, etc., the user can choose to ignore their differences. For the user, they are all properties of the student.
4.2.4 SDBM PROPERTIES

The SDBM property can be viewed as a special kind of relationship. It is defined to be the relationship between an SDBM object and a symbolic string or strings. The relationship discussed in the previous section is between two SDBM objects.

The important characteristics of a symbolic string participating in a property can be summarized as follows:

1. The existence of a symbolic string is considered to be eternal. Therefore, the conceptual constraint imposed on the relationship between two objects will always hold for properties as long as the object exists. In other words, the existence of a property depends only on the existence of the object.

2. A symbolic string is not an object.

3. Not any symbolic string should be allowed to exist. For a specific property, only certain symbolic patterns are permitted. They are specified with the SDBM type and window system.

So far, we have focused on the discussion of the conceptual structure and its basic constructs — objects, relationships, and properties. We now turn to addressing data representations of the conceptual structure. The next section discusses the concept of SDBM data type and the section following presents the concept of the SDBM window.

4.2.5. SDBM TYPE CONCEPT

As mentioned before, we have to resort to symbolic strings for data manipulation. The semantics of objects, relationships and properties all have to be conveyed through symbolic strings. The conveyance is bilateral either from the user to the computer or from the computer to the user. To ensure the accuracy of such conveyance, we need certain mechanisms to structure symbolic strings and to provide certain semantics for the
conceputal structure. SDBM data types are designed to be such a mechanism.

Let us define the concept of data type. In programming languages, a data type is defined by a symbolic name, a set of data objects, a set of operations that manipulate these data objects, and the possible value bindings. This definition requires that the specification of a data type must contain the following three elements:

1. a name for the type
2. a syntactic specification for symbolic strings of the type
3. a set of operations for manipulating data objects of the type. They are the only operations that can directly manipulate the syntactic representation of symbolic strings of the type.

In SDBM, we use the concept of data type as defined above. We should emphasize that the term data object does not necessarily refer to the physical computer. They may refer to any virtual computers.

The name of a type is an abstraction of the syntactic specification and the set of operations. The name dictates the symbolic patterns and the operations that may be allowed.

In SDBM, the name equivalence rule is adopted for the data type equivalence test. Therefore, distinctive semantics are associated with the SDBM data type name. For example, data type MILE and data type WEIGHT may both be defined on the same set of integers with the same syntactic specification. But they are not comparable in SDBM, because different semantics are associated with them. We will discuss more about SDBM data types in the next chapter when their syntax and semantics are described.

\^5 Do not be confused with SDBM objects. The term data object here refers to a run-time grouping of one or more pieces of data in a virtual computer. Alternatively, it can be viewed as a container for data values.
4.2.6 SDBM WINDOW CONCEPT

So far, we have constructed a conceptual world consisting of objects, relationships, and properties. We also introduced the SDBM data type concept which is used to structure symbolic strings for the purpose of external data representation and manipulation. We have not yet answered questions such as "How can we specify the conceptual structure among objects, relationships, and properties?" and "How can we integrate the conceptual structure and its external data representation?". The SDBM window concept is developed to answer these questions. This is probably the most important concept of the Semantic DataBase Model, SDBM.

More specifically, the following functions will be accomplished by the window concept.

1. To provide a window through which certain aspect of SDBM objects or relationships can be seen (accessed) and manipulated.
2. To describe and maintain the structure among objects, relationships, and properties.
3. To model the persistence of objects, relationships, and properties.

We will further clarify this concept below.

To begin with, it is impractical, even though possible, to manage objects individually because of the following three reasons. First, the number of objects in a database may be enormous. Second, there may be differences between every two objects. Third, each object may play a variety of roles. Thus, we need to treat them in groups. This is accomplished through using SDBM windows. That is, a window specifies certain properties which we are interested in and which are allowed to be accessed and manipulated. Through this window, only objects that have these properties can be accessed and manipulated. The specific syntax of symbolic strings permitted to describe properties as well as objects and relationships through SDBM windows is specified with the SDBM type system.
From the conventional semantic data modelling point of view, an SDBM window maybe appear to be the result of applying the classification mechanism to objects and is, therefore, the same as the class concept used in many existing semantic data models. Although similarities do exist between these two concepts, there are conceptual differences. At this point, we can identify at least three differences between the window concept and the class concept.

First, a window is not an object container as a class is. Second, in a class objects are grouped together because of the homogeneous properties shared by them. However, through a window, we may access and manipulate two objects with different properties. The only requirement is that properties of every object have the same abstraction. Take the example used in Section 2.1.7. With SDBM, both female and male patients can all be seen through the window PATIENT\textsuperscript{16}. We will be able to see female patients with property PREGNANT-OR-NOT and male patients without this property. Although variations are allowed among SDBM properties, the number of such variations should be relatively small compared with the total number of objects or relationships that can be seen from that window.

The third difference is that a class contains only objects while both objects and relationships may be seen through windows, though they may not be seen through the same window. This will be demonstrated in Section 5.3 where the syntax and semantics of the SDBM window are presented. From now on, we will call the abstraction of properties seen through a window the role of that window. Apparently, it satisfies the definition of the role concept given in Section 4.2.2.

As identified before, SDBM window's other function is to describe and maintain the

\textsuperscript{16}When we say that object A can be seen through a window, we mean that A can be accessed or manipulated through that window.
A conceptual structure is a structure consisting of objects, properties, and relationships and is the conceptualization of the 'real' world being modelled.

4.2.7 AN EXTENSION TO SPECIALIZATION MECHANISM

In Section 3.3.3, we discussed the specialization mechanism which is used as a major tool to enhance the semantic modelling ability in many recently developed semantic data models. TAXIS applies this mechanism uniformly to all the aspects of data modelling.

Clearly, different views of the same thing are supported by this mechanism. However, using the conceptualization presented in this paper, we can easily pinpoint the limitation of

17A conceptual structure is a structure consisting of objects, properties, and relationships and is the conceptualization of the 'real' world being modelled.
the specialization mechanism. A major limitation which is inherent to the mechanism is that all the different views generated with the mechanism must be on the same specialization hierarchy. The implication of this limitation is that two different views of the same thing differ only in the levels of information detail. The fact that an object may be viewed from completely different angles can not be effectively modelled by this mechanism. For example, in the real world an object can act as a student and an employee at the same time. Here the student is neither a specialization of the employee nor is the employee a specialization of the student. This important real world phenomena cannot be effectively modelled by many existing semantic data models. For example, to model such fact with TAXIS, a TAXIS relation StudentEmployee has to be created, which should be a specialization of both the STUDENT relation and the EMPLOYEE relation. When the number of such specializations increases, the database specification can become very complex.

It is apparent that we should extend the modelling ability provided by the specialization mechanism. To discuss what extensions should be introduced, we first investigate the relationship between windows and objects.

For two different windows W1 and W2, an object S can be related to them in three different ways:

Type 1. If S can be seen through W1, it will be seen through W2.
Type 2. If S can be seen through W1, it may be seen through W2.
Type 3. If S can be seen through W1, it will not be seen through W2.

From now on, we will call these relationships constraints. Notice that these constraints are defined in such a way that they are oriented towards windows instead of towards individual objects. In other words, they are constraints among windows.

For type 1 constraints, the SDBM subwindow hierarchy will be used to model it. This type of hierarchy can be described with the specialization mechanism. The SDBM subwindow concept provides a means to give an object a more detailed view by adding
to the object information which may not be seen through its superviews. The most important property of the SDBM subwindow hierarchy is that an SDBM window and its subwindows are not dealing with different objects. Rather, they are modelling the same group of objects with the restriction that objects seen from a subwindow form a subset of objects seen through its superwindows. More detailed discussion on the SDBM subwindow hierarchy will be given in Section 5.3.2.

Type 2 constraints are the most common among these three types of constraints. For example, a student object may also be an instructor. But, another student object may not be an instructor. Therefore, the constraint cannot be definitely specified at the window level. Rather, it varies with individual objects. In SDBM, Type 2 constraints are modelled from two aspects. At the window level, we specify the possibility for an object to play multiple roles. For a specific object, different roles are assumed with corresponding data manipulations.

An example of Type 3 constraints is that a university may have a policy stating that a full-time student can not be a university administrator. A Type 3 constraint is actually an exclusion constraint between two windows. Type 3 constraints can be enforced at the window level. The specific SDBM construct provided for this purpose will be described in Section 5.3.3.
5. OVERVIEW OF SDBM

In this chapter, we present an overview of the semantic database model, SDBM. Various considerations of the design of SDBM are also addressed. SDBM is designed to attack the modelling problems of the relational data model and to present a more effective way of modelling the real world than that available from the relational data model. Compared with other recently developed semantic data models, SDBM has several distinct and novel features. In section 1.3, we pointed out shortcomings of some relevant models. In the previous chapter, we outlined the conceptualization behind the design of SDBM. We will summarize the novelties of our approach in the last chapter.

SDBM supports many well-developed features of semantic data modelling, e.g., object orientation and data abstraction. The model also supports the type hierarchy and the subwindow hierarchy. However, the most important feature is that greater modelling power has been made possible by conceptualizing semantic data modelling activities in a different but more effective and natural way.

This research is intended to be a preliminary part of further study in database programming languages. At the moment, we do not attempt to design SDBM as a database programming language. Rather, it is intended to be a semantic database model. The difference between database programming languages and semantic data models is one of emphasis: a database programming language is a set of constructs for specifying the objects and procedures involved in an information system, whereas a semantic data model is a set of constructs for building a representation of the structure of the 'real' world, along with the necessary operations to manipulate that representation (King & McLeod [1985]).
In the following, we will discuss the SDBM type system, and then the SDBM window system. Every SDBM schema specification consists of these two parts. Finally, the SDBM transaction facility will be described. In this chapter, the specification of the SDBM, both the syntax and the semantic, are informal. In APPENDIX A, a formal syntax is provided.

5.1 SOME NOTATION CONVENTIONS

The following notation conventions are used in the prototypes of SDBM syntax in this chapter.

Upper case : indicates an SDBM keyword which is to be typed in verbatim by the user.

Braces { } : indicates an optional repetition. The symbol inside may be repeated zero or more times.

Brace Plus { }+ : indicates a repetition. The symbol inside may be repeated one or more times.

Brackets [ ] : indicates an optional alternative. The symbol inside may have zero or one occurrence.

Bracket Plus [ ]+ : indicates an alternative. The symbol inside must have one occurrence.

< > : indicates a nonterminal symbol.

Vertical bar | : indicates an alternative.

:= : indicates that the left side is to be replaced by the right side.

To increase readability, in the syntax, we always add some kinds of prefix or postfix to nonterminal symbols to indicate their usage. In the case of ambiguity, " " are used to delimit a terminal symbol.
5.2 SDBM TYPE SYSTEM

In this paper, the two-view conceptualization is used throughout the design of SDBM. To realize such a two-view conceptualization in the computer, we need a mechanism to support data representations and a mechanism to support conceptual structures. The former is the SDBM type system which is discussed in this section. The latter will be discussed in section 5.3.

5.2.1 ELEMENTARY DATA TYPES

The built-in elementary SDBM data types include NUMERIC, BOOLEAN, CHARACTER, and ENUMERATION. They are the elementary data types that will be used to construct other more complex data types. Each elementary data type has certain built-in operations that are permissible with them.

5.2.1.1 NUMERIC

A SDBM NUMERIC data type is specified in a way similar to COBOL picture clause. The prototype of SDBM NUMERIC data type declaration is

\[
\text{TYPE} \ ( \ <\text{NumericTypeID}> = \text{NUMERIC} \ \text{WITH} \\
\quad <\text{NumericRange}> \ \{ \ OR \ <\text{NumericRange}> \} \\
) \]

\[
<\text{NumericRange}> := <\text{NumericConstant}>..<\text{NumericConstant}> 
\]

The following are some examples.
Example 5-1.

TYPE ( Salary= NUMERIC WITH 10000..99999 )
TYPE ( Height= NUMERIC WITH 0..99.99 )
TYPE ( Temperature= NUMERIC WITH -99..99 )

In the above examples, each type declaration is preceded with a keyword 'TYPE' to distinguish it from window declarations and transaction declarations. 'Salary', 'Height', and 'Temperature' are numeric-type identifiers. 10000..99999 declares a five-digit positive integer data type whose values are between 10000 and 99999; 0..99.99 declares a positive real number with two digits before the decimal and two digits after; and -99..99 declares an integer data type which could assume either positive or negative value with two digits. Like any other type declaration, the declaration is enclosed in parentheses ( ).

The operations defined for NUMERIC data type are the assignment, the arithmetic operations, and the relational operations.

5.2.1.2BOOLEANS

A boolean value can be either of, 'TRUE' or 'FALSE'. The specification is straightforward. For instance

Example 5-2.

TYPE ( True= BOOLEAN WITH {TRUE} )
TYPE ( B= BOOLEAN WITH {TRUE,FALSE} )

The prototype of BOOLEAN data type declaration is

TYPE ( <BooleanTypeID>= BOOLEAN WITHN
   "{"TRUE|FALSE|TRUE,FALSE|FALSE,TRUE"}" )

There is no ordering defined for BOOLEAN data type. The operations on a BOOLEAN data type include the assignment and the logic operations (AND, OR, NOT).
5.2.1.3 CHARACTERS

A character data type consists of data objects that have a string of characters as their values. This data type specifies certain desired format of character strings. For example

Example 5-3.
TYPE ( Name= CHARACTER WITH A(10) )
TYPE ( Addressnum= CHARACTER WITH 9(4) )
TYPE ( Student#= CHARACTER WITH X(11) )

Notice that although 'Addressnum' is a four-digit number, which is indicated by 9(4), it is not a numeric number. That is, the arithmetic operations do not apply to it. 'A(10)' indicates an alphabetic string with a length of 10. X(11) represents an alphanumeric string with length of 11. The prototype of CHARACTER data type declaration is

```
TYPE (<CharacterTypeID>= CHARACTER WITH
   <CharacterPicture> {OR <CharacterPicture>})
   
   <CharacterPicture>:= X(<Integer>)|9(<Integer>)|A(<Integer>)|<CharacterList>
   <CharacterList>:= {[A|X|9|<QuateList>]+}
   <QuateList>:= "<Character>{<Character}>"
```

The above syntax indicates that a character data type can be defined to have several variants which are separated by the keyword OR. This is desirable for example when telephone numbers are to be recorded so that we do not have to write down the area code for each phone number. <Character> is defined in APPENDIX A.
5.2.1.4 ENUMERATIONS

An enumeration is a list of distinctive values. Usually, these values are in a certain order. To eliminate some potential ambiguities, the ordering will not be considered in SDBM. Some declaration examples of the ENUMERATION data type are

Example 5-4.

```
TYPE ( Sex = ENUMERATION WITH {Male,Female} )
TYPE ( A = ENUMERATION WITH {High,Medium,Low} )
```

The enumeration data type is specially useful when a data object can only take on a small number of values. The prototype of the enumeration data type declaration is

```
TYPE ( <EnumerationTypeID>= ENUMERATION WITH
       "{"<EnumerationList>"}
       )
```

The basic operations on the ENUMERATION data type are the assignment and the relational operations. The relational operations do not include less-than and greater-than operators because no ordering of listed values is provided in the current specification of the SDBM ENUMERATION data type.

5.2.2 STRUCTURED DATA TYPES

A structured data type is a data object which is an aggregate of other data objects. A structured data type can be constructed from elementary or any other previously declared data types. Currently, the only structured data type supported by SDBM is the SDBM record data type.

An SDBM record data type is similar to an ordinary record data type. However, improvement has been introduced in order to eliminate some problems associated with the ordinary record data type. Let us first use an example to demonstrate the concept.
Example 5-5.

TYPE ( Date= RECORD WITH
    Month:Monthtype,
    Day:Daytype,
    Year:Yeartype;

BEGIN
    Month LT 13 AND Month GT 0,
    IF Month EQ 1 OR 5 OR 7
        OR 8 OR 10 OR 12
    THEN Day LE 31
    ELSE IF Month EQ 2
        THEN Day LE 28 OR 29
        ELSE Day LE 30
    ENDIF
ENDIF
END

In above example, SDBM record type ‘Date’ defines a set of values that can be taken to represent a chronological date. This data type guarantees (almost) that any incorrect input will be rejected when it goes through type checking. Therefore integrity of the date description is protected. ‘Monthtype’, ‘Daytype’, and ‘Yeartype’ are data types that must be declared previously.

The prototype of the SDBM record data type declaration is

TYPE ( <RecordTypeID>= RECORD WITH
    [ [VARIANT|COMMON ]
    <ComponentDeclaration>{,
    [VARIANT|COMMON]
    <ComponentDeclaration>};

    [ BEGIN
    <ConstraintAssertion>{,


The SDBM record data type declaration can be divided into two parts, component part and constraint part. In the component part which is ended with ';', a record structure is declared which is composed of a number of named components that are usually heterogeneous. The components of this structure can be declared to have any previously declared data type.

The SDBM record structure can signify that certain components are multi-valued with keyword MULTIPLE. If a component is specified to be multi-valued, it can take more than one value of the same type. Because the role modelled by an SDBM window may not be syntactically homogeneous, certain components may not make any sense with some objects. Therefore, the SDBM record structure allows its components to be optional, which is specified with the keyword OPTIONAL. Another important feature of the SDBM record structure is that it can have several variants. This kind of record structure specifies a collection of records of different syntactic types. All of them have some components in common. Components appear under the keyword COMMON or components that do not follow either COMMON or VARIANT are, by default, common to all the records of the structure. But some of these records may have components that are different from others. Such record variants are specified by giving several component-id and Component-declaration pairs which are placed under the keyword VARIANT. For example, an employee payroll record structure may have two variants because permanent employees are paid monthly and
temporary employees are paid hourly.

Example 5-6.

TYPE ( Employee= RECORD WITH

  COMMON
    EmpID: Integer,
    Dept: DeptName,
  VARIANT
    EmpType: {Permanent}
    MonthlyRate: Rate1,
    StartDate: Date,
  VARIANT
    EmpType: {Temporary},
    HourlyRate: Rate2,
    OvertimeRate: Rate3
)

In the constraint part which is optional and is enclosed by BEGIN...END, constraints on components or among components are expressed. Constraints are all specified in first-order logic. Constraint assertions are separated by a comma ',' which is a shorthand for the logic operator AND.

To simplify specification, an IF...THEN...ELSE structure is introduced and Boolean expressions are allowed on the right side of a relational operation sign. ENDIF is used to eliminate any potential syntactic ambiguity associated with the IF...THEN...ELSE structure.

The constraints specified here apply only to the value of the data type. Namely, the SDBM type system only deals with domain constraints and, to certain degree, cardinality constraints. The constraints on objects are specified in the window declaration part.

There are two basic operations defined for the SDBM record data type. One is the component selection operation (or the dot operation) whose syntax is

\(<VariableID>.<ComponentID>\)
It selects the designated component from the record bound to the variable `<VariableID>`.

The other operation is the assignment which assign the value of a record to another record of identical data type. For example

\[
\text{Record1} = \text{Record2}
\]

assigns the value of Record2 to Record1. Here variable Record1 and variable Record2 have been declared to be of the same record data type.

### 5.2.3 Abstract Data Types

An abstract data type is a new data type defined by the database designer that includes:

1. one or more data type definitions.
2. a set of abstract operations on the defined data types. These operations are usually defined with several pieces of programs (procedures or functions).
3. Encapsulation of type definition and operations.

In SDBM, we provide a facility which will support the above ADT (Abstract Data Type) concept. The following is an example that illustrates the specification of an SDBM abstract data type 'Date1'.

**Example 5-7.**

\[
\text{ADT ( Date1 = TYPE Date LOCAL; ADT (Compare(d1:Date,d2:Date): IS ENUMERATION WITH \{L,S,E\}; ACTION: FUNCTION Compare TYPE (T IS NUMERIC WITH 19760000..20000000); VARIABLE x,y: T; BEGIN x= d1.Year*10000+d1.Month*100+d1.Day,}
\]

The declaration of an SDBM ADT can be recognized by the keyword ADT. In the above declaration, the type definition of the abstract data type is given after keyword TYPE. The logical structure (syntactic structure) of ADT 'Datel' is defined with the data type 'Date' which of course must be previously declared. The operations that are permitted to access the internal components of the data type are declared after the keyword ADTOP which is short for ADT OPerations. Parameters of the ADT operation are given along with their data types. Operations are defined to be either functions or procedures. In the former case, a single result is returned explicitly. In the latter case, no result or more than one result are returned explicitly.

In the above declaration, there is a keyword LOCAL for the data type 'Date'. This declaration excludes any other operations from accessing the internal structure of the data type 'Date'. Therefore, the encapsulation abstraction is achieved. Notice that such encapsulation can only be obtained through dedicated language or model design and implementation. Ordinary subprogram facilities such as those in PASCAL are not sufficient to prevent illegal accesses to the internal structure, even though these facilities can be used to produce data types with new operations.

The details of ADT operations are specified after the keyword ACTION. In this part, actions performed by one or more operations are defined.
Let us explain the ADT operation 'Compare'. This operation is used to compare two dates. If the first date is later than the second date, character 'L' is returned. If the dates are the same, 'S' is returned. If earlier, 'E' is returned. Therefore, we define the value returned to be of an enumeration data type. This enumeration type is defined to be a subtype of the built-in data type ENUMERATION\textsuperscript{19} We assume that all the components of the data type 'Date' have been declared to be numeric and of the same data type. Arithmetic operations are performed on them, and the results are compared. Based on these comparisons, appropriate characters are assigned to the ADT operation.

The prototype of the SDBM ADT declaration is as follows. For more detail, see APPENDIX A.

\begin{verbatim}
ADT (<ADTID>=
   TYPE <TypeID> [LOCAL];
   ADTOP [<ADTFunctiopnOP>|<ADTProcedureOP>]*{,
   [ <ADTFunctiopnOP>|<ADTProcedureOP>]*};
   ACTION:
   <ADTFunctionOrFunction>
)
\end{verbatim}

\begin{verbatim}
<ADTFunctiopnOP>:=
   <ADTFunctiopnID>(<FunctionParameters>):<TypeSpecification>
\end{verbatim}

\begin{verbatim}
<ADTProcedureOP>:=
   <ADTProcedureID>(<ProcedureParameters>)
\end{verbatim}

Though generally speaking the internal structure of an abstract data type should not be accessible from outside the ADT definition, sometimes we only want to add several special operations to a data structure. We do not want to hide the structure and we do not want to eliminate the ordinary operations defined for that data type in SDBM. In this case, keyword LOCAL must not be specified. This will make the structure of the ADT and

\textsuperscript{19} The concept of subtype will be discussed in next section.
the operations on the structure visible outside. That is, any operation applicable to the
data type <TypelD> will also be applicable to the ADT.

5.2.4 SUBTYPE HIERARCHIES

The concept of subtype hierarchy (or type hierarchy for short) has been supported
by several programming languages or data models such as Simula 67 (Birtwistle [1973]),
Smalltalk (Ingalls [1978]), Ada (Wegner [1980]), and Galileo. The type hierarchy concept of
Ada is a facility to give another name for a type whose set of values has been
constrained. The concept of type hierarchy supported by Simula 67, Smalltalk, and Galileo
are similar. The type hierarchy concept of Galileo, for example, can be defined as follows.
Data type t' is a specialization or a subtype of the data type t if a value of t' can be
used in any context where a value of the data type t is permitted.

The type hierarchy concept of SDBM is similar to that defined above. However, there
are both conceptual and structural differences between the type hierarchy of Galileo and
that of SDBM. Conceptually, the type hierarchy of Galileo is a mechanism used to deal
with the intensional aspect of its class hierarchy. Its extensional aspect is dealt with by the
derived classes of Galileo. With SDBM, the type hierarchy does not necessarily define the
inclusion hierarchy of windows. Structurally, Galileo adopts a mechanism that automatically
establishes type hierarchies among data types based on their structure. One flaw associated
with this approach is that two 'compatible' structures may have very different semantics.
Consider for instance the following two data type examples defined in Galileo.
Example 5-8.
TYPE ( Company := ( Name : STRING
AND Address : Address
AND Phone : PhoneNumber)
AND Person := ( Name : STRING
AND Address : Address
AND Phone : PhoneNumber
AND BirthDate : Date)
)

Obviously, the data type 'Person' should not be considered a subtype of the data type 'Company'. With SDBM, type hierarchies are established when the designer explicitly specifies it and the related data structures are compatible according to SDBM type hierarchy rules which will be explained later in this section.

Let us first look at the type hierarchy related to elementary data types. The prototype is:

TYPE ( <SubtypeID> IS <TypeID> [ WITH
   <ElementaryTypePicture> ]
)

<ElementaryTypePicture> := <NumericRange>|   
<BooleanPicture>|   
<CharacterPicture>|   
<EnumerationPicture>

Where <TypeID> must be a previously declared type or subtype ID. Four built-in elementary data types can be used here. They are NUMERIC, BOOLEAN, CHARACTER, and Enumeration. They cannot be used as subtypes. <ElementaryTypePicture> is in fact a constraint which specifies a subset of values of type <TypeID>; otherwise the subtype relationship can not be established.
Notice that an elementary subtype declaration does not introduce a new data type. A subtype and its supertype are the same data type in terms of type equivalence. To put it in a different way, every variable declared of the subtype will have the same set of permissible operations as that of the supertype. However, it takes its values from a subset of values permitted to the supertype. The following are several examples of elementary data type hierarchy declarations.

Example 5-9.
```plaintext
TYPE ( ManagerSalary IS Salary WITH 30000..99999 )
TYPE ( Open IS BOOLEAN )
TYPE ( GradStudent# IS Student# WITH
      "95-"XXXXXX
      )
TYPE ( Sex IS ENUMERATION WITH {M,F} )
```

We now clarify the SDBM record type hierarchy concept with several examples. To a certain extent, in SDBM the type hierarchy is a convenient way of creating data types. For example

Example 5-10.
```plaintext
TYPE ( Time IS Date WITH
      Hour : (+99,NUMERIC)
      )

TYPE ( Am IS Time WITH
      BEGIN
      Hour LE 12
      END
      )

TYPE ( SpringDate IS Date WITH
      Month: IS MonthType WITH 3..5
      )
```
In the first example, the data type 'Time' is defined to be a data type 'Date' with additional component Hour which is defined by the data type HourType which is assumed to be a subtype of NUMERIC. The data type 'Date' used above is defined in Section 5.2.2. In the second example, the data type 'Am' is defined to be the data type 'Time' with the additional constraint that Hour must be less than 12. Structurally, the declaration of 'Am' is equivalent to the following declaration

Example 5-11.

```
TYPE ( Am = RECORD WITH
    Month: MonthType,
    Day: DayType,
    Year: YearType,
    Hour: HourType;

BEGIN
    Hour LE 12,
    Month LE 12,
    IF Month EQ 1 OR 5 OR 7
      OR 8 OR 10 OR 12
    THEN Day LE 31
    ELSE IF Month EQ 2
      THEN Day LE 28 OR 29
      ELSE Day LE 30
      ENDIF
    ENDF
END
)
```

These two declarations are, however, not equivalent. In the latter case, no type hierarchy is established. That is, the data type 'Am' declared in the latter example is not a subtype of 'Time'. It is a different data type with the same structure. We can make the latter a subtype of Time by explicitly adding 'IS Time' to the declaration.
Example 5-12.

TYPE ( Am IS Time WITH
  Month: MonthType,
  Day: DayType,
  Year: YearType,
  Hour: HourType;

BEGIN
  Hour LE 12,
  Month LE 12,
  IF Month EQ 1 OR 5 OR 7
   OR 8 OR 10 OR 12
  THEN Day LE 31
  ELSE IF Month EQ 2
    THEN Day LE 28 OR 29
    ELSE Day LE 30
    ENDIF
  ENDIF
END
)

It is apparent that the declaration of Am in Example 5-10 has the advantage of economical representation. Namely, it occupies much less space because only the additional components need to be declared. In SDBM, the order of components in a record data type does not affect the type hierarchy. Therefore, the above declaration is more flexible in that it can have its components in any order.

In SDBM, a record subtype ST is obtained by adding information to a record type T. Information can be added in any of three ways. First, the information can be added so that the set of components of ST contains the set of components of T. Second, the information can be added by simply adding more constraint assertions to T. Third, the information can be added through redefining the data type of an existing record component so that the new data type is the subtype of the original one. This third
approach has been illustrated with the last example in Example 5-10. Any combination of
these two will also produce a subtype.

In summary, we have the following rules to determine the subtype hierarchies. These
are necessary conditions but not sufficient. They are used in SDBM to check the
correctness of the type hierarchy declaration.

1. For any type \( t \), \( <t, t> \)

2. For any numeric data type \( t_1 \) and \( t_2 \), \( <t_1, t_2> \) only if the numeric value set
   of type \( t_1 \) is a subset of the numeric value set of type \( t_2 \).

3. For any boolean data type \( t_1 \) and \( t_2 \), \( <t_1, t_2> \) only if the boolean value set
   of type \( t_1 \) is a subset of boolean value set of type \( t_2 \).

4. For any character data type \( t_1 \) and \( t_2 \), \( <t_1, t_2> \) only if the character value
   set of type \( t_1 \) is a subset of character value set of type \( t_2 \).

5. For any enumeration data type \( t_1 \) and \( t_2 \), \( <t_1, t_2> \) only if the enumeration
   value set of type \( t_1 \) is a subset of the enumeration value set of type \( t_2 \).

6. For any SDBM record data type \( t_1 \) and \( t_2 \), \( <t_1, t_2> \) only if at least one of
   the following conditions is satisfied.
   a. If \( c_2 \) is a component of type \( t_2 \), then \( c_2 \) must be a component of \( t_1 \)
      and the data type of \( c_1 \) must be a subtype of the data type of \( c_2 \).
   b. If \( c_2 \) is a constraint assertion of \( t_2 \), then \( c_2 \) must be a constraint
      assertion of \( t_1 \).

The significance of the type hierarchy concept can be summarized in three parts.
First, it facilitates type checking. The type hierarchy ensures that if type \( t_1 \) is a subtype of
type \( t_2 \), then a type \( t_1 \) value can be used in any context where a type \( t_2 \) value is
expected. The reverse, however, is not true. This property makes it easier to develop
complex database software on an incremental basis, because once a system is verified and

\(^2\) This notation is borrowed from (Albano [1985]). \( <a, b> \) means that type \( a \) is a
subtype of \( b \).
validated for data at a more general level, it is guaranteed for data of any subtype which may be introduced later on. Second, the type hierarchy concept increases the semantic modelling ability in that it provides a means to associate distinctive semantics with data types in terms of type equivalence. We have mentioned before that SDBM uses the name equivalence rule. With this rule, any two type definitions are always different. For example, in the following declaration:

```plaintext
TYPE ( MILE= NUMERIC WITH 0..1000 )
TYPE ( HEIGHT= NUMERIC WITH 0..100 )
```

Both MILE and HEIGHT are numeric data types. But they are not equivalent. Therefore, illegal operations like MILE+HEIGHT can be detected. Data types that are equivalent can be declared with type hierarchy mechanism. For example, the width and length of a rectangular can be declared to be equivalent as follows:

```plaintext
TYPE ( WIDTH IS NUMERIC WITH 10..50 )
TYPE ( LENGTH IS NUMERIC WITH 10..100 )
```

In fact, any two data types with a common root are equivalent. Third, the notion of the type hierarchy also enhances the specification of the database. It makes the specification concise and readable. Therefore, it may contribute to more reliable database software development. As a final comment, for SDBM records, the structural part and the constraint assertion part have different implications for the type checking of type hierarchies.

### 5.3 SDBM WINDOW SYSTEM

In Section 5.2, we described the SDBM type system. An SDBM data type specifies a set of symbolic values that can be used to describe certain roles played by an object (or a set of objects) and a set of operations for manipulating these values. The SDBM type system only deals with the external data representation aspect of the database. To accomplish the database functions, however, we also need mechanisms to describe objects, to describe properties and relationships among objects, to model persistence of objects,
and to open a channel for the database user to communicate with the database. The SDBM window system is such a mechanism. Section 4.2.6 has outlined the basic concept of the SDBM window concept. This section will further clarify the specific syntactic constructs that support this concept.

For this purpose, several examples will be used in this section. Of course, all the explanations are informal. Window 'Student' and window 'Course' are declared as follows. The relevant data types are also defined.

Example 5-13.
TYPE (StudentType= RECORD WITH

 StdNo: StudentNumber,
 StdName: StudentName,
 DeptName: DepartmentName,
 CrsNo: CourseNumber (MULTIPLE),
 Section: SectionNumber (MULTIPLE),
 Address: Address,
 PhoneNo: PhoneNumber
)

TYPE (CourseType= RECORD WITH

 CrsNo: CourseNumber,
 CrsTitle: CourseTitle,
 DeptName: DepartmentName,
 Credit: CourseCredit
)

TYPE (CourseSectionType= RECORD WITH

 CrsNo: CourseNumber,
 Section: SectionNumber,
 Instructor: InstructorName,
 StdNo: StudentNumber (MULTIPLE)
)

TYPE (DepartmentType= RECORD WITH


DeptName: DepartmentName,
Director: DirectorName,
Address: Address

WINDOW (Student= TYPE: StudentType;
    REFERENCE: Department(DeptName= DeptName);
    RELATIONSHIP: Take(CourseSection(
        CrsNo=CrsNo,
        Section=Section
    )
    )<--> IsTaken,
    BelongTo(Department(
        DeptName= DeptName
    )
    )<--> Have;
    KEY: StdNo
)

WINDOW (Course= TYPE: CourseType;
    REFERENCE: Department(DeptName=DeptName);
    RELATIONSHIP: IsOffered(Department(
        DeptName= DeptName
    )
    )<--> Offer,
    Have(CourseSection(CrsNo=CrsNo,
        )<--> BelongTo;
    KEY: CrsNo
)

WINDOW (CourseSection= TYPE: CourseSectionType;
    REFERENCE: Course (CrsNo= CrsNo);
    RELATIONSHIP: BelongTo(Course(CrsNo=CrsNo,
        )<--> Have,
    IsTaken(Student(StdNo=StdNo,
        )<--> Take;
Several points need to be elaborated. First of all, there are four attributes that may be defined in an SDBM window specification. They are TYPE, REFERENCE, RELATIONSHIP, and KEY. We explain these attributes subsequently.

The TYPE attribute defines what type of values can be used to describe an object seen through that window. Therefore, in the example, a Student object must have values of data type StudentType and a Course object must have values of data type CourseType. Different identifiers are required for different windows and their corresponding data types. This is to eliminate potential confusion that might occur in the variable declaration part of the SDBM transaction which will be discussed later on. The data type used in the SDBM window declaration must be previously specified in the type declaration part of the SDBM schema. In this paper, we call the component of an SDBM record data type used to describe the SDBM window and therefore to describe certain aspects of objects the property of the objects. TYPE, REFERENCE, RELATIONSHIP, and KEY are called attributes of the window.

Interrelationships that affect insertions or deletions of objects need be specified in the window declaration part. In other words, references must be declared explicitly. This attribute is necessary because in SDBM objects and symbolic strings are explicitly

\(^2\) When we say that a Student object must have values of data type StudentType, we mean that only the values of data type StudentType can be used to describe the object playing the role of a student.
differentiated. Such declaration is accomplished through the REFERENCE attribute. Identifiers of the referenced window are listed after the keyword REFERENCE. In order to successfully create or insert an object with reference constraints, the database designer must also specify how references should be established. For this purpose, certain property identifiers that can identify specific objects in a window may be given in the parentheses following the window identifier in the REFERENCE attribute. These identifiers must be components of the data type corresponding to that window so that data types are secured and value transmissions are provided. For example, in window Course we have the following reference:

**REFERENCE: Department (DeptName=DeptName)**

Where the identifier DeptName before the sign ‘=’ must identify a property of window Department and the identifier DeptName after the sign ‘=’ must be a property identifier of window Course. These identifiers are not necessarily keys. The REFERENCE attribute is optional, which indicates that not every object depends on some other object to exist. The specific implications of the REFERENCE attribute for database operations will be discussed later when we describe the SDBM transaction.

The RELATIONSHIP attribute is used to explicitly establish relationships between objects. Let us explain this attribute with an example taking from Example 5-13.

**RELATIONSHIP: Take(CourseSection(CrsNo= CrsNo, Section= Section)) <--> IsTaken;**

The symbolic string ‘Take’ is the identifier for a relationship between a Student object and a CourseSection object. CourseSection in the parentheses identifies the window through which a Student object is to establish a relationship with an object, i.e. a CourseSection object. In the inner parentheses, the symbolic string ‘CrsNo’ before the ‘=’ identifies a property of CourseSection objects. the symbolic string ‘CrsNo’ after the ‘=’
represents a property of Student objects. It is not necessary for these two identifiers to be identical as in this case. But they must have the same data type. ‘Section= Section’ in the same parentheses can be interpreted similarly. These property identifiers are used to establish a specific relationship between two objects. Whenever a Student object is created or inserted into the Student window or when its CrsNo property value or/and Section property value is modified, its new CrsNo property value and Section property value will be passed to the CourseSection window. Based on the values of CourseSection properties CrsNo and Section, appropriate CourseSection object will be retrieved. If such an object is found, a specific relationship will be established. Otherwise, no relationship will be established. More detailed implications for data manipulation will be discussed in Chapter 6.

Symbolic string ‘IsTaken’ after the sign ‘<-->’ identifies the same relationship as identified by ‘Take’ but views it from the reverse direction. Namely, a relationship between two objects, for example between a student and a course section, can be viewed from two different angles. On one side, he/she is ‘taking’ a course section. On the other side, a course section is ‘taken’ by a student. Through specifying ‘Take <--> IsTaken’, we explicitly define two different views of the same relationship.

Although there are two identifiers for a relationship, we do not need to establish both the ‘Take’ relationship and the ‘IsTaken’ relationship in order to establish the relationship between a student and a course section. In fact, once a ‘Take’ relationship is established, an ‘IsTaken’ relationship is automatically established, because they are actually the same relationship.

The specification in Example 5-13 permits the relationship between a Student object and a CourseSection object to be established from either side. However, the relationship between a student and a department can only be established through the Student window. We give this restriction in the specification only to demonstrate that the way of establishing a relationship is controlled by the database designer. The information
requirement may shows that in the real world the relationship that a student belongs to a
department is established when the student is registered instead of when the department
is created. Therefore, it is also natural to have the above restriction. The RELATIONSHIP
attribute is optional.

Let us discuss the existence difference between the REFERENCE attribute and the
RELATIONSHIP attribute. The REFERENCE attribute specifies the semantic integrity constraint
that must be satisfied by an object if it is to exist in the window. The RELATIONSHIP
attribute, however, specifies the semantic integrity constraint for the existence of a
relationship. It states that for a relationship to exist two participating objects must exist.
But the existence of a relationship will not affect the existence of objects.

The KEY attribute is used to impose the uniqueness constraint on the data
representations of objects seen through a window. Identifiers listed after the KEY attribute
must not conflict with the associated data type declaration. In other words, they must be
components of the data type and therefore properties of objects when seen through that
window. Apart from specifying the uniqueness constraint, keys in SDBM are solely for
external convenience. They are not used internally to identify objects. The KEY attribute is
also optional.

The prototype of the SDBM class declaration is

\[
\text{WINDOW} (<\text{WindowID}> [ \text{ IS } <\text{WindowID}> \text{ WITH } | = ]^{+} \\
\text{ TYPE: } <\text{TypeID}>[,<\text{TypeID}>]; \\
\text{ [REFERENCE: } <\text{References}>; ] \\
\text{ [RELATIONSHIP: } <\text{Relationships}>; ] \\
\text{ [KEY: } <\text{KeyIDs}> ] \\
\) \\
\]

\[
<\text{References}>:=<\text{WindowID}>(<\text{PropertyPairs}>){, \\
<\text{WindowID}>(<\text{PropertyPairs}>)} \\
\]
<PropertyPairs> ::= <PropertyID> = <PropertyID> {,
                       <PropertyID> = <PropertyID> }

<Relationships> ::= <Relationship> {,<Relationship>}

<Relationship> ::= <RelationshipID> (<WindowID> [(<PropertyPairs>)]
                               ) <--><RelationshipID>

<KeyIDs> ::= <PropertyID>{,<PropertyID>}

Notice that in the above SDBM window syntax specification, a window can be
associated with multiple data types. This reflects the real world phenomenena that two
collections of properties with different symbolic appearance may be abstracted to the same
conceptual role. Let us use a familiar example that an employer could be an individual, a
corporation, or a government organization. This fact can be modelled in two ways. On
one hand, if we are interested in individuals, corporations, and government organizations
and we can treat them as different roles and model them with different windows, we can
model the fact with an employer window which has a single data type. We then insert\textsuperscript{22}
corresponding individual objects, corporation objects, or government organization objects
into the employer window.

On the other hand, if we are not particularly interested in treating individuals,
corporations, and governments as objects, the fact can be modelled with an employer
window that has multiple data types. For example:

\textsuperscript{22} The 'insert' operation will be explained in Section 5.4.3.4.
Let us take a look at how the fact that a student is enrolled in a certain section of a course can be modelled in SDBM. For convenience, we call this fact enrollment. In example 5-13, we have in fact modelled enrollment. But there an enrollment does not have its own properties. So, they are modelled with the binary relationship through the properties of students and courses. In the following examples, enrollments with their own properties will be discussed.

It is apparent that an enrollment can be viewed from two different angles. First, it can be treated as an object and declared as follows.

**Example 5-14.**

```plaintext
TYPE ( EnrollmentType= RECORD WITH
    StuName: StudentName,
    StdNo: StudentNumber,
    CrsNo: CourseNumber,
    Section: SectionNumber,
    Grade: Grade
)
```

```plaintext
WINDOW ( Enrollment= TYPE: EnrollmentType;
    REFERENCE: Student(StdNo= StdNo),
                Course(CrsNo= CrsNo);
    KEY: StdNo,CrsNo,Section
)
```

The above declaration defines Enrollment objects to have properties StuName, StdNo, CrsNo, Section, and Grade. The existence of an Enrollment object is dependent on the existence of the corresponding Student object and the Course object. This approach is
similar to that of the relational data model.

Alternatively, we can view an enrollment as an aggregation of a student object and a course object. Instead of creating a new object for the enrollment, we can simply generate a new ‘window’ through which the aggregation of the student object and the course object is seen. Although we have discussed the SDBM window concept, we restricted ourselves to seeing only certain aspects of the same kind objects through the window. Here, we proceed to extend the SDBM window concept so that relationships among objects of different kinds can also be viewed through the window. Let us explain this concept through the following declaration.

Example 5-15.
WINDOW ( Enrollment= TYPE: StdNo OF Student,
                      CrsNo OF Course,
                      Section: SectionNumber,
                      Grade: Grade;
                      AGGREGATE: Student,
                      Course
                      )

The difference between this declaration and the declaration in Example 5-14 is that this SDBM window declaration does not create a new set of objects. In other words, there will be no enrollment objects. All we have from this declaration is a new view of some already existing objects. Keyword AGGREGATE indicates the windows from which objects will be used to construct the aggregation. Keyword TYPE defines the external view of the aggregation. In this case, properties StdNo, CrsNo, Section, and Grade are given. Notice that the window identifier following the keyword OF indicates that the property must also be a property defined in that window and is used to establish the link between a specific aggregation and a specific object of that window. Furthermore, such an aggregation is not simply putting together two objects. It may have its own characteristics too. These characteristics do not belong to any single underlying object and can not be
obtained from them. Rather, they must be explicitly specified. In SDBM, they are declared in the TYPE part of the AGGREGATION WINDOW. The detailed syntax of aggregation is given in APPENDIX A.

5.3.1 SDBM SUBWINDOW HIERARCHY

In SDBM, an instance relationship is simply a relationship between a WINDOW and its objects. SDBM supports a three level instance hierarchy, i.e. object, window, and metawindow. The metawindow-window hierarchy, however, is trivial in SDBM. There is only one built-in metawindow, WINDOW, of which every specific window is declared as an instance. It is not possible to define new metawindows in SDBM.

This restriction, however, is not a defect of the model for two reasons. First, three levels of instance hierarchy are sufficient for modelling most data processing phenomena. Instance relationships above these three levels are both beyond our comprehension and of little practical use. Second, the existence of the SDBM type system partially renders a complex metawindow definition facility unnecessary.

On the other hand, the SDBM subwindow concept provides a mechanism to give an object a more detailed view by adding to the object information which can not be seen through its superwindows. We use the following example to clarify the specification of the SDBM subwindow hierarchy.

Example 5-16.

TYPE (GradStudentType IS StudentType WITH
    AdvisorName: PersonName,
    AdvisorPhone: PhoneNo
)

WINDOW (GradStudent IS Student WITH
    TYPE: GradStudentType;
A subwindow hierarchy is specified with keyword IS. The changed attributes are specified after keyword WITH. The semantics of this definition need to be clarified. Firstly, for this subwindow hierarchy to be established, data type GradStudentType must be a subtype of data type StudentType. We do not design SDBM in a way that the subtype hierarchy can determine the subwindow hierarchy automatically, because otherwise we are assuming that each type is associated with a unique window. They are two different concepts and therefore should be separated. In SDBM, subwindow relationships must be explicitly declared.

Secondly, the REFERENCE attribute of window Student is inherited by window GradStudent. Additional REFERENCE attributes can be declared to window GradStudent. The RELATIONSHIP attribute of window Student is also inherited by window GradStudent. Additional relationships between objects may exist through subwindows. The KEY attribute of window Student is inherited by window GradStudent. Extra KEY attribute may be added to window GradStudent.

Finally, as soon as the subwindow relationship is verified, internal routines will be generated to ensure that every object seen through window GradStudent can also be seen through window Student. When an object is successfully inserted into window GradStudent, it will also be inserted into window Student. Upon deletion of an object from GradStudent, it will also be deleted from window Student if such cascade is allowed.

5.3.2 CONSTRAINTS AMONG WINDOWS

In Section 4.2.7, we pointed out the limitations of specialization as a data modelling mechanism. We identified that there are three different types of constraints among windows. The last section described the SDBM subwindow hierarchy which is the mechanism to model Type 1 constraints, namely the constraints that if an object is seen
in the first window it must be seen in the second window. This section presents the
constructs for modelling Type 2 and Type 3 constraints.

In SDBM, the construct MAYBE is used to record Type 2 constraints. Let us consider
an example.

Example 4-17.
WINDOW ( Student IS Person,
       MAYBE Instructor WITH
       TYPE: StudentType
       )

The declaration specifies that if an object plays the role of a student he/she must
also plays the role of a person. By using the keyword MAYBE, the declaration also
specifies that an object may be a student and an instructor at the same time. Notice that
IS constraints and MAYBE constraints have very different implications for the semantics of
data manipulation. With the IS constraint, a student automatically ‘becomes’ a person. With
the MAYBE constraint, a student is only permitted to ‘become’ an instructor. It ‘becomes’
an instructor only after an appropriate SDBM Insertion Statement is successfully executed.
The SDBM Insertion Statement will be explained in Section 5.4.3.4. Because of the
symmetrical nature of Type 2 constraints, it will have the same effect if the MAYBE
construct is specified with the Instructor window or if the construct is specified with the
both windows.

A Type 3 constraint is declared with the keyword ISNOT. Consider the following
example.
Example 5-18.

WINDOW ( Student IS Person,
       MAYBE Instructor,
       ISNOT Administrator WITH
       TYPE: Student
     )

The above declaration enforces the policy that a full-time student can not be a university administrator. Every time an object is to be inserted into the Student window, the system will first check if it is already in the Administrator window. In other words, before the system allows an object to play the role of a student, it will first check if the object plays the role of an administrator and vice versa.

5.4 SDBM TRANSACTION SYSTEM

If a database management system is to be useful, at least four basic operations must be provided. They are object creation, object removal, object value retrieval, and object value modification. SDBM provides these four operations with automatic integrity enforcement by performing adequate typechecking and adequate window constraint checking. The database state can be permanently affected only after these two kinds of checking are completed successfully. Otherwise, the operation will be aborted and the database will be kept intact. Besides the above four operations, two additional operations should also be defined because of our conceptualization of data modelling on which SDBM is built. They are object insertion and object deletion. Object insertion is to insert an already existing object into a different window, or in other words, to give a new role to the object. Object deletion is to delete an object from a window so that it will not be seen in that window. That is, it will no longer play the role described by that window.

---------

2 3 Without causing any confusion, we use object values to mean values that are used to describe certain properties of an object.
Object creation is done by the CREATE statement; object destruction by the DESTROY statement; object value retrieval by the RETRIEVE statement; and object value modification is done by the MODIFY statement. In addition, object insertion is done by the INSERT statement and deletion is done by the DELETE statement.

Often a set of such operations need to be integrated together in order to effect certain conceptually meaningful changes to the database. In terms of abstraction, the user is usually only interested in the 'macro' effect of such a set of operations. How such an operation is constructed is not important to them. Moreover, from the user's point of view, such operation should not only be conceptually meaningful but also conceptually primitive. An operation is primitive if it either achieves its desired effect successfully or terminates without any effect on the database. Conceptually, we want changes to the database to directly reflect changes to the 'real' world being modelled. The SDBM transaction system provides a behavioral abstraction facility allowing definition of conceptually primitive operations.

As stated before, the major objective of this research is to make an investigation of our conceptualization of data modelling and SDBM is not designed to be a database programming language. For this reason, we have not attempted to provide a sophisticated transaction system to SDBM. Rather, the transaction constructs are centered around the basic behavioral operations such as those mentioned at the beginning of this section.

5.4.1 TRANSACTION AS AN ABSTRACTION OPERATION

An SDBM transaction closely resembles the subprogram concept in programming languages. The specification allows a user to understand it without digging into how it is implemented. The syntax of a transaction declaration consists of the following four aspects:

1. the name of the transaction;
2. the arguments, their order, and their data types;
3. the results, their order, and their data types;
4. the body of the transaction.

The syntax is borrowed from the typical syntax for the parameter transmission subprogram of PASCAL-like programming languages. The prototype of the transaction declaration is

```plaintext
TRANSACTION ( <TransactionDesignator>
   <TransactionBody>

)```

Below, we will first address issues related to TransactionDesignator. Its semantics will be clarified. In section 5.4.3, we will discuss TransactionBody in detail.

### 5.4.2 TRANSACTION DESIGNATOR

The TransactionDesignator gives a means to identify a transaction and specifies data for the database manipulation and data to be returned from the manipulation. The identification of the transaction is simply accomplished by giving each transaction (pre-defined set of operations) a unique identifier.

In an SDBM transaction designator, following the transaction identifier is a set of formal parameters together with their type identifiers or window identifiers. These formal parameters are further separated into IN parameters and OUT parameters. The former designates parameters that are not modifiable by the transaction, while the latter designates results to be returned.

When a formal parameter is given a type or a window, the type or the window must be previously declared in the SDBM type or window system. Any declared type or window is legal.
The prototype of the transaction designator specification is

\[
\text{<TransactionID>}(\{\text{<ParameterDefinition>}, \text{<ParameterDefinition>}\})
\]

SDBM transactions can be procedural operations which are specified when there is no \text{OUT} parameter or when there are more than one \text{OUT} parameter definitions. Functional operations are specified when there is exactly one \text{OUT} parameter. But they are specified uniformly. \text{<ParameterDefinition>} is specified as follows.

\[
\begin{align*}
\text{[IN]} & \quad \text{<ParameterID>:<TypeIDOrWindowID>} \\
\text{or} & \\
\text{OUT} & \quad \text{<ParameterID>:<TypeIDOrWindowID>}
\end{align*}
\]

For \text{IN} parameters, keyword \text{IN} may be omitted.

Below, we make some further comments about the transaction designator of the SDBM transaction facility. Firstly, all the formal parameters are local to the transaction and are invisible outside. Secondly, actual parameters are given when the transaction is called. The correspondence between the actual parameters and the formal parameters is established by pairing actual and formal parameters according to their respective positions in the actual and formal parameter lists. Thirdly, for the \text{IN} parameters, transmission-by-value method is used. That is, when the transaction is called, the value of the actual parameter is copied into the formal parameter and the actual parameter cannot be modified by the transaction. This method is justified because in the database system the primary concern is with modifying the persistent objects rather than modifying variables local to a transaction. In addition, objects are always globally accessible. For the \text{OUT} parameters, the transmission-by-result method is used. That is, when the transaction call is terminated, the final value of the formal parameter will be copied to the actual parameter. Fourthly, a transaction may have no formal parameters. Finally, When \text{<TypeIDOrWindowID>} is a window identifier, the formal parameter is specified to have objects as its values. For such
a formal parameter, the value of the actual parameter is in fact a pointer to some specific object. However, how such binding is implemented should not be a concern of the user.

### 5.4.3 TRANSACTION BODY

A Transaction Body, corresponding to the body of a subprogram definition, consists of three parts, type declaration, variable declaration and statements. The prototype of the definition is

```
[<TypeDeclaration>]
[<VariableDeclaration>]
BEGIN
<Statements>
END
```

We will first describe the type declaration part, then the variable declaration part, and finally the statement part in turn. The prototype of the type declaration part is

```
<TypeDeclaration>::=
  TYPE <TypeID>
  [=<TypeDefiniton>|<SubtypeDefinition>]+{, <TypeID>
  [=<TypeDefiniton>|<SubtypeDefinition>]+}
```

The prototype of the variable declaration is

```
VARIABLE <VariableList>:<TypeSpecification>|<WindowID>{,<VariableList>:<TypeSpecification>|<WindowID>};
```

This part is provided to ease the design of SDBM transactions. Two kinds of variables can be declared, type variables and object variables. A type variable is specified with a type identifier and an object variable is defined with a window identifier. An object variable usually has objects of the specified window as its values. However, if the window does not directly model an object but an aggregation of two or more objects, the value
of the object variable is this aggregation. In effect, an object variable is a surrogate of some existing objects or their aggregations. Two issues should be clarified. They are scope rules and initialization.

In SDBM, (static) scope rules are straightforward. Temporary data are always visible only within their transaction and persistent data are always globally visible. All the variables declared inside a transaction are temporary and local. They are destroyed when the transaction is terminated and they are only visible inside the transaction. Object variables are no exception even though objects are permanent and global.

For both formal parameters and variables, uninitialized values could be dangerous to the database design because the computer cannot distinguish between an initialized-value and an uninitialized-value. In SDBM, all the parameters and variables are initialized immediately upon creation to be NULL, which indicates that they have not been purposely bound to anything.

The major part of a transaction is a set of statements. These statements are used to accomplish database manipulation functions. Some statements provide actions that actually make changes to persistent data in an SDBM database and actions that retrieve data from a database. Others provide control facilities necessary to define a conceptually meaningful transaction. SDBM statements are listed as follows.

```xml
<AssignmentStatement> | <TransactionCallStatement> | 
<CreateStatement> | <DestroyStatement> | 
<InsertStatement> | <ModifyStatement> | 
>DeleteStatement> | <RetrieveStatement> | 
<AbortStatement> | 
<CompoundStatement> | <ConditionalStatement> | 
<IterationStatement>|<AddStatement>|<RemoveStatement>
```
Below, we will discuss these different types of statements in turn.

5.4.3.1 ASSIGNMENT

The syntax of the assignment operation is

\[ <\text{AssignmentStatement}> ::= <\text{VariableID}> = <\text{Expression}> \]

Assignment is the operation used to change the binding between a value and a data object. It is defined in SDBM to have a meaning similar to the assignment statement in PASCAL. The meaning of the above syntax is that the value of the SDBM expression specified by \(<\text{Expression}>\) is copied to variable \(<\text{VariableID}>\) and no explicit result is returned. Therefore, an SDBM assignment operation can only be used as a statement.

Because SDBM is typed, both the result of expression on the right side and the variable on the left side have specific data types. Only when both sides have the same data type can the operation pass type checking. When the type is an SDBM data type, the assignment has the same semantics as that of PASCAL. When the type is an SDBM window, the result of the right must be bound to a specific object (or aggregate of objects) seen through the window and the statement causes the object variable on the left to bind to that object (or aggregate of objects).

5.4.3.2 CREATION

The prototype of \(<\text{CreateStatement}>\) is

\[
\text{CREATE } <\text{VariableID}> \text{ INTO } <\text{WindowID}> \text{ WITH (}<\text{PropertyValueList}>)
\]

This built-in operation primitive creates a new object whose component values are given by \(<\text{ClassValueList}>\). Conceptually, variable \(<\text{VariableID}>\) will have the object as its value. This variable, which could be viewed as a container of objects, is specified to provide some operational convenience. For example, we often need to do some additional
operations on the object subsequently. If no such variable is defined, we have to retrieve it from the database each time we need it.

In order to create a new object, several conditions must be satisfied. Firstly, variable \(<\text{VariableID}>\) must have been declared in the transaction to be an object variable of the window designated by \(<\text{WindowID}>\). Values in \(<\text{PropertyValueList}>\) must conform to the data type of window \(<\text{WindowID}>\). Secondly, the execution of an SDBM CreateStatement must also check the uniqueness constraints on the window, if the KEY attribute has been specified in the SDBM schema. Any attempt to create an object that violates key constraints will be rejected. Thirdly, before creating a permanent object in the database, SDBM will check the REFERENCE attribute. If the property is specified, the system will use the relevant data values given in the value list of the statement to retrieve the referenced objects. If any one of them does not exist, no new object will be created. If the RELATIONSHIP attribute is specified and some properties are dedicated to establish the relationship, then appropriate objects in some other window will be searched to determine their existence. If the search fails, corresponding property values will be set to NULL and therefore no relationship will be established as the result of the object creation. Notice the RELATIONSHIP attribute will not affect the creation of objects. Finally, it should be reiterated that a new object can be created through any legitimate window, namely any window that has been declared for the object. The successful execution of a CreateStatement will have its effect propagated to all its generalizations (or superwindows). Namely, it will be seen through all its generalized windows.

We will use the following two examples to further explain the CreateStatement.
Example 5-19.

CREATE x INTO Student WITH
  ( StdNo="1011",
   StdName="Gary Utter",
   DeptName="Computer Science",
   CrsNo="CPSC511",
   Section="001",
   Address=
   Phone=""
  )

CREATE y INTO Course WITH
  ( CrsNo="Comm534",
    CrsTitle="System Analysis",
    DeptName="Commerce",
    Credit="1.5"
  )

Let us explain the first example. If the Department object named 'Computer Science' exists and all the data type requirements are met, an object will be created. This object is currently playing the role of a student. His student number is 1011, his name is Gary Utter, his department is Computer Science, his address is 4460 W12th. Van., and his phone number is 224-6578. If the course CPSC511 exists and it has section 001, then the relationship that he takes CPSC511 in Section 001 will be established. Otherwise, properties CrsNo and Section will have value NULL which indicates he 'takes' no course section or we do not know what course section he 'takes'. After the creation is completed, variable x will be bound to this object.
5.4.3.3 DESTRUCTION

The prototype of `<DestroyStatement>` is

\[ \text{DESTROY <VariableID>} \]

This built-in behavioral primitive, which reverses the effect of the CREATE statement, destroys an object from the entire database. That is, when an object is destroyed, it can no longer be seen in any window. In order to destroy an object from a database, two conditions must be satisfied. First, the variable `<VariableID>` must be already bound to the desired object. Second, the object is not referenced by any other objects in the database. As long as there is an object referencing the object from somewhere in the database, the DESTROY statement will be aborted.

When an object is destroyed from a database, all the relationships it participates in will be terminated. Viewed from a related object side, the effect is that certain values of properties that are dedicated to establish the relationship will be modified into NULL.

When the DESTROY statement is completed successfully, the variable `<VariableID>` will be bound to NULL.

5.4.3.4 INSERTION

The prototype of `<InsertStatement>` is

\[ \text{INSERT <VariableID>}(<VarID>,<VarID>) \text{ INTO <WindowID> [WITH (} <PropertyValueList>)]) \]

This built-in behavioral primitive inserts an existing object into a different window. That is, the object is given a different role with this operation. The additional data values needed are given by `<PropertyValueList>`. `<PropertyValueList>` is optional because the new window may have the same data type as the original one. In this case, all the necessary
values are already in the database, and we need not specify them again in this statement. In order to insert an object into a different window, several conditions must be satisfied. First of all, the variable `<VariableID>` must be declared in the transaction to be an object variable of the window designated by `<WindowID>`. The format of the value `<PropertyValueList>` must conform to the data type of window `<WindowID>`. Secondly, before execution, the variable `<VarID>` must already be bound to the object to be inserted. Such a binding can be achieved through operations such as object creation and object retrieval. Such binding locates the object which we want to view from a different angles. Thirdly, the execution of an SDBM InsertStatement must also check the uniqueness constraint on the window, if the KEY attribute has been specified in the SDBM schema. Any attempt to insert an object that violates the key constraint will be rejected. Fourthly, before making permanent changes to the conceptual structure, SDBM will check the relevant REFERENCE attribute, if any. The system will use the data given in the InsertStatement to retrieve the referenced objects. If the retrieval returns a NULL value, which means that the referenced object does not exist, insertion will be rejected. Finally, when the RELATIONSHIP attribute is specified, an INSERT statement may cause new relationships to be established. Based on the values of those properties dedicated to establish certain relationships, a new relationship will be established if the appropriate object exists. Otherwise, no relationship will be established. Notice that no matter a relationship can be established or not, it has no effect on the successful execution of an INSERT statement.

The following is an examples of the InsertStatement.
Example 5-20.
INSET x1(x) INTO GradStudent WITH
( AdvisorName="Jim Jackson",
 AdvisorPhone=
)

In this example, variable x is already bound to a student whose student number is 1011 when it was created earlier in Example 5-19. The above statement shows that this student is also a graduate student with an advisor whose name is Jim Jackson and phone number is 228-9990.

It is time to further clarify the concept of binding between the SDBM transaction variable and the SDBM object. In SDBM, the variable-object binding is always achieved through the binding between the variable and the window and the binding between the window and the object. This is because every variable in SDBM belongs to exactly one type. For instance in the previous example, variable x is bound to an object through window Student. When we want to insert the object into window GradStudent, we can not specify the insertion statement as follows.

Example 5-21.
INSET x INTO GradStudent WITH
( AdvisorName="Jim Jackson",
 AdvisorPhone=
)

If forced to use the above specification, we have to choose between two choices. First, we can dynamically change the typing of x so that after the execution of this statement it will be a variable of window GradStudent. In this case, SDBM is no longer typed and we lose all the benefits expected from static data typing. The second alternative is to keep the type of variable x. However, this choice is also awkward because we can not

\textsuperscript{2} This type of binding will be called variable-object binding.
retain access to the object through window GradStudent any more unless we make another retrieval.

Therefore, x1(x) is used in the insertion statement. Variable x is as described above, and variable x1 must be declared to be a variable of window GradStudent in the variable declaration part of the transaction. With this mechanism, a new variable-object binding is established through window GradStudent. Of course, the object is the same one. This time it is bound to object variable x1.

5.4.3.5 MODIFICATION

The SDBM ModifyStatement is primarily designed to change the property values of an existing object. The prototype of the ModifyStatement is

\[<\text{VariableID}>,<\text{PropertyID}> = <\text{NewValue}>\]

\[<\text{NewValue}> := "<\text{CharacterString}>"|<\text{Expression}>\]

<CharacterString> is defined in APPENDIX A. <Expression> will be explained in Section 5.4.4.

Modification is accomplished through modifying a local object variable designated in the prototype by <VariableID> which serves as a surrogate of the object to be modified. <PropertyID> indicates the property to be modified. The object is designated by <VariableID>. New values are obtained either by directly specified values in the case of <CharacterString> or through evaluation of an expression in the case of <Expression>.

In order to successfully accomplish a modification operation, several conditions must be satisfied. First of all, various relevant data types must be consistent. For example, the type of the new value must conform to the type of <VariableID>,<PropertyID>. Secondly, the uniqueness constraint specified by the KEY attribute may be affected by the
operation, because in SDBM any property is allowed to be modified, even when it is specified to be a KEY. Therefore, the uniqueness constraint must be checked, if it is so specified. Thirdly, the referential constraint specified by the REFERENCE attribute may be violated by the operation. That is, an object may be modified to reference a non-existing object. For example, a graduate student may change his advisor. We have to make sure that the new advisor exists. Finally, the MODIFY statement can be used to modify the relationships among objects through modifying values of those properties that are dedicated for the purpose of establishing relationships. In this case, we terminate an old relationship and start a new one.

Let us take a look at an example.

Example 5-22.

x1.AdvisorName = "Larry Peters"

After this operation, the object bound to x1 through its role as the graduate student will have Larry Peters as his advisor. It should be emphasized that only the property value of an object variable is modified with the ModifyStatement and such modification is permanent. The value of a type variable or an object variable can be simply modified through an assignment. Such modification, however, is temporary and only has effect within the transaction. The modification statement can be considered as an extension to the assignment statement discussed in Section 5.4.3.1.

Notice that the modification statement will not create a new object no matter what changes have been made. Its effect is to get the property value of an object in a certain window changed so that the object will look different, or when the property happens to dedicate certain relationship, existing relationship may be terminated and a new relationship may be established.
5.4.3.6 DELETION

The prototype of `<DeleteStatement>` is

```
DELETE <VariableID>
```

`<VariableID>` designates the object to be deleted from a window. Symmetric to the problem of insertion, deletion of an object requires that it is not currently referenced by any other objects in the database. This is essential to prevent the database from having 'dangling references'.

It is also required that none of its specializations is referenced by any other objects because of the inheritance rule of the SDBM window hierarchy. The deletion of an object will propagate to all its specializations. An alternative is that a variable can be deleted only if it has no specialization. However, we believe that deletion cascade should be allowed because it is more natural. For example, suppose that an object plays the role of a student, the role of a graduate student, and the role of a PHD student at the same time. If he ceases to be a student, he must also cease to be a graduate student, and therefore cease to be a PHD student.

Deletion of an object will not destroy the object. The object will still be in the database until it is explicitly destroyed. Deleting an object from a window only means that some aspects of the object will no longer be seen.

When the RELATIONSHIP attribute is concerned, the DELETE statement has the same effect as the DESTROY statement.

Deletion of an object will set the variable designated by `<VariableID>` to its initialized value, i.e. NULL. That is, the variable-object binding is terminated so that the variable will not indicate any object.
5.4.3.7 RETRIEVAL

The SDBM RetrieveStatement is straightforward. The prototype is

```
RETRIEVE <VariableID>[.<PropertyID> TO <VariableID>{, <VariableID>.<PropertyID> TO <VariableID> }]
[ WHERE (<Expression>) ]
```

When some window property values of an object are to be retrieved, the object
variable identifier should be followed by a dot and the window property identifier. Here
dot '.' is a window property selection operator. Below are two examples of the
RetrieveStatement.

Example 5-23.
```
RETRIEVE x.StdName TO stdname, x.Address TO stdaddress
   WHERE ( x.StdNo='1234'

RETRIEVE x WHERE ( x.StdNo='1234' )

RETRIEVE x.Phone TO stdphone
```

The first retrieval statement will search the values of properties 'StdName' and
'Address' of object x whose student number is 1234. The values obtained will be stored
in variable stdname and variable stdaddress, respectively. Also as a result of this operation,
object variable x will be bound to the retrieved object through the window Student.

The second retrieval statement will retrieve a Student object whose student number
is 1234. If such a student exists, it will be bound to the variable x.

---

25Do not confuse this operation with the record component selection. In record
component selection, the variable identifier before the dot must be bound to an SDBM
record type instead of an SDBM object through a window.
The third retrieval statement will retrieve the value of property 'Phone' of the object bound to the variable \textsuperscript{2}\textsuperscript{6} and assign the value to the variable stdphone. Three additional points in regard to this example should be made in order to further clarify the semantics of the SDBM RetrieveStatement.

First, \(x\) must have been previously declared to be an object variable of window Student. And upon the execution of this statement it must have been bound to a specific object or a specific set of objects seen through window Student, otherwise execution will be aborted. The binding may be obtained through using a WHERE clause of the RetrieveStatement. Only those objects that evaluate the Boolean expression after keyword WHERE to be true will be retrieved. Obviously, if \(x\) has been bound to the object we do not need the WHERE clause any more.

Second, the property identifier following the dot must be consistent with the window designated by the variable identifier.

Third, variable stdphone must be declared to have the same data type as the property Phone of window Student. This variable is used to store the value of the retrieved student phone number. The retrieval statement itself will not return an explicit result. The dot operator used for property selection will be further discussed in Section 5.5.

Apparently, a single RetrieveStatement alone can only provide answers to very simple queries. More powerful data manipulation functions are achieved by combining it with some sequence control facilities which will be discussed in detail subsequently. There are three different sequence control facilities in SDBM. They are CompoundStatement, ConditionalStatement, and IterationStatement.

\footnote{For short, object \(x\) will be used to mean the object that is bound to variable \(x\).}
5.4.3.8 COMPOUND STATEMENT

The prototype of the CompoundStatement is

```
BEGIN
<Statements>
END
```

A compound statement is a sequence of statements which are enclosed with BEGIN and END and are arranged in the order in which they are to be executed. Statements are separated by commas. A compound statement is treated in an SDBM transaction as if it was a single statement. The compound statement enables the database designer to define a transaction incrementally. The internal structure may be changed without altering its overall effects. This concept is in fact an application of the aggregation principle.

5.4.3.9 CONDITIONAL STATEMENT

A conditional statement is one that provides alternation of two or more different sequences of statement execution. A conditional statement is also treated as a single statement in SDBM transaction construction. Its prototype is

```
IF <Expression>
THEN <statement>
[ELSE <statement>]
ENDIF
```

The execution sequence is controlled by a test on the Boolean expression. If the expression is evaluated to be true, the statement after THEN is evaluated; if the expression is evaluated to be false and there is an ELSE specification, the statement after ELSE is executed; otherwise, the conditional statement is entirely skipped. The statement after THEN or ELSE must be a single statement or something that is treated as single statement. The conditional statement is ended with the keyword ENDIF which is used to eliminate any
syntactic ambiguity.

5.4.3.10 ITERATION STATEMENT

There are two kinds of iteration statements:

<ForStatement> | <WhileStatement>

To suit database applications, SDBM has borrowed two different kinds of iteration statement from programming languages with a little modification. One kind of iteration provides iteration over all objects of a window. The prototype of the statement is

FOR EACH <VariableID> DO
   <Statement>

This statement resembles counter-controlled repetition in programming languages such as the FOR statement in FORTRAN and ALCOL. The difference is that this iteration is over each object of a window, whereas FOR statement in programming languages is over integers.

Three points are worth mentioning about the FOR statement. First, variable <VariableID> must be declared in the variable declaration part of the transaction to be an object variable. Second, the iteration is unconditional. That is, variable <VariableID> will be bound to each object of the window in turn. For each object, statement <StatementID> is executed and the binding can not be changed by the execution of the statement. Third, statement <Statement> must be a single statement or something that is treated as a single statement, e.g. a compound statement.

The other type of iteration statement provides a facility to terminate the iteration on a specified condition. The prototype of the statement is:

WHILE <Expression> DO
   <Statement>
In the WhileStatement, the expression, which must be a Boolean expression, will be evaluated first. If it is evaluated to be true, statement <Statement> is executed. Otherwise, the WhileStatement will be skipped. Each time after statement <Statement> is executed, the expression is re-evaluated.

5.4.3.11 ABORT STATEMENT

As stated previously, we do not intend to provide a user-specifiable exception handling facility in SDBM. However, we need at least one mechanism to stop the execution of an SDBM transaction if certain condition cannot be satisfied. AbortStatement is provided for this purpose. Its syntax is simple.

<AbortStatement> ::= ABORT

The meaning of this statement is that it will stop the transaction wherever it is specified and undo whatever has been done. Upon the completion of this statement, the database should be in the same state as if the transaction had never been executed.

5.5 EXPRESSIONS

Viewed as a programming language, SDBM is basically statement oriented. Expressions are one of the basic syntactic constituents of SDBM statements. They are used in statements when a value needs to be computed. The formal syntax of SDBM expressions is given in APPENDIX A and will not be repeated here.

The basic issues associated with SDBM expressions are the syntactic form, the precedence of SDBM operators, and typing of SDBM expressions. Below, we discuss these issues in turn.

SDBM expression are constructed in the mixed syntactic form with the infix form for arithmetic, boolean, and relational operations and the prefix form for transaction calls and
ADT operation calls.

In SDBM expressions, the following operations are permitted. They are relational operations including EQ (equality), NE (inequality), GT (greater than), GE (greater than or equal to), LT (less than), and LE (less than or equal to), Boolean operations including AND (conjunction), OR (inclusive disjunction), and NOT (negation), arithmetic operations including + (addition), - (subtraction), * (multiplication), and / (division), unary operations including - (negation) and + (absolute value), the dot operation, transaction calls, and ADT operation calls.

Following the convention of many programming languages, SDBM defines the precedence of these operations as follows:

<table>
<thead>
<tr>
<th>Lowest Priority</th>
<th>Highest Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADT Call</td>
<td>Dot Operator</td>
</tr>
<tr>
<td>LT</td>
<td>OR</td>
</tr>
<tr>
<td>GE</td>
<td>+</td>
</tr>
<tr>
<td>GT</td>
<td>*</td>
</tr>
<tr>
<td>LT</td>
<td>-</td>
</tr>
<tr>
<td>LE</td>
<td>+</td>
</tr>
<tr>
<td>LT</td>
<td>/</td>
</tr>
</tbody>
</table>

The above precedence is incorporated into the grammar for SDBM expressions. As indicated in the grammar, the precedence of an expression can be changed with parentheses.

SDBM expressions are typed. That is, every SDBM expression has a type which can be statically determined at compilation time of an SDBM schema. Syntactically, expressions in SDBM are well typed. Therefore, there will be no execution-time type errors with an SDBM expression that has been type checked to be syntactically correct.

---

27 + and - operations that have the same precedence as the OR operation are binary operation.
To conclude this section, we make a further discussion of the dot operator used in SDBM expressions. The following two examples are used for this purpose.

**Example 5-24.**

\[
y = x.\text{DeptName} \\
z = x.\text{BelongTo}
\]

In the first example, the dot operator is used for property selection. Here, variable \(x\) is already bound to a Student object. \(x.\text{DeptName}\) will return the value of property \text{DeptName} of the object. In the example, the value is then assigned to variable \(y\) which must be declared to have the same data type as the property \text{DeptName} of the student object. In the second example, the dot operator is used for the relationship selection. \(x.\text{BelongTo}\) will return an object that participates in the relationship \text{BelongTo} with object \(x\). The object returned here is a department which the student \(x\) belongs to. In the example, the returned department object will be bound to variable \(z\). Variable \(z\) must have been declared to be an object variable of Department.

**5.6 OPERATIONS ON MULTI-VALUED PROPERTIES**

In Section 2.1.3, we identified the inadequacy of the restriction in the relational data model that property domains must be atomic. We, therefore, permit multi-valued properties in SDBM. Such properties are specified with the keyword MULTIPLE. In SDBM, multi-valued properties not only provide a natural description of one-to-many and many-to-many relationships between objects and symbolic string values but also provide a natural description of one-to-many and many-to-many relationships between objects.

However, at the same time we improve the modelling of static structures of the 'real' world, we have to deal with some difficulties associated with the behavioral aspect of multi-valued properties. For example, the modification operation defined in Section 5.4.3.5 is only valid for single-valued properties. This section discusses operations provided
for manipulating multi-valued properties.

5.6.1 ADDITION

A new value may need to be added to a multi-valued property, for example, when an employee joins an additional project. The syntax of this operation is as follows:

\[
\text{ADD } \left[ \text{<VariableID> | "<CharacterStrings>"} \right]^+ \text{ TO } \text{<WindowVariableID>,<PropertyID>}
\]

Where <VariableID> should have been declared to have the same data type as the property <PropertyID> and should be already bound to a specific value of that data type. If the property is a REFERENCE property, then the new value to be added must lead to an existing object so that the reference can be established. If the property is a RELATIONSHIP property, the value to be added must also lead to an existing object. Otherwise, the relationship can not be established. The KEY constraint must be checked, if the unique requirement is specified with the property. The ADD statement is illustrated in the following two examples:

Example 5-25.
ADD x TO t.Project
ADD "'Blue','Yellow'" TO y.Color

Where in the first variable x has the data type used to define the Project property of Employee and it should be already bound to a value of that type. In the second example, variable y is bound to a Part object.

5.6.2 REMOVAL

To remove a value from a property reverses the effect of adding that value to the property. For example, an employee may drop a project, or we may find out that a part has only one color instead of two as recorded previously. Therefore, we want to disassociate the employee from the project and correct the color information recorded for
the part. These desired operations can be expressed as follows:

Example 5-26.
REMOVE p FROM t.Project
REMOVE "Yellow" FROM y.Color

Where p is a variable that has the same data type as the project property and is bound to a value indicating the project to be removed. Variable t is bound to the Employee object and variable y is bound to the Part object.

The syntax of the REMOVE statement is:

REMOVE [<VariableID>|"<CharacterStrings>"]+ FROM <VariableID>..<PropertyID>

Where <VariableID>, <StringList>, <VariableID>, and <PropertyID> have the same meaning as defined in Section 5.6.1. If the property is a REFERENCE property, this statement will terminate a reference(s). If the property is a RELATIONSHIP property, this statement will terminate a relationship(s), too.

5.6.3 MODIFICATION

The modification operation to be defined in this section is an extension of the assignment operation in Section 5.4.3.1 and the modification operation in Section 5.4.3.5. There only a single value can be operated on. The syntax of the modification operation for multi-valued properties is:

<VariableID>,<PropertyID>=
    NULL|"<StringList>"|<VariableID>..<PropertyID>

The following are three examples of the modification operation:
Example 5-27.

x.Color = NULL
x.Color = "'yellow','White','Black'"
y.Project = z.Project

As the result of this first modification, the multi-valued property color of parts will have a NULL value, which means that we do not know what color(s) the part object x has. The second modification indicates that the part has three colors, yellow, white, and black. With the third modification, object y will have the same project values as object z has.

The modification operation discussed here has the same implications for KEY, REFERENCE, and RELATIONSHIP properties as the modification operation discussed in Section 5.4.3.5 has. From now on, we will not make difference between assignment statement and modification statement.
6. AN APPLICATION EXAMPLE OF SDBM

In chapter 5, we outlined the basic elements and structures of a proposed semantic database model, SDBM. To further clarify the model and the underlying rationale, we shall use SDBM to design a database. The database is designed to manage information related to a hypothetical project management system.

The basic methodology of using SDBM to design databases is the same as that used to design SDBM itself, namely the two-view conceptualization of data modelling and the principles of data abstraction. The design of SDBM databases begins with identifying objects and their roles.

Constraints are classified into categories. Some constraints are only related to the external data representation aspect, and therefore should be modelled with SDBM types. Others are related to objects, roles, and their structures and therefore are naturally modelled with SDBM windows. Then, these windows are refined so that different views of the same objects may be generated according to the different access requirements.

The refinement is carried out through studying objects and roles they play. All the possible roles that we are interested in should be identified, and then the relationships or constraints are analysed and classified.

The further refinement is accomplished through the specialization mechanism. To begin with, the most general situations are considered to generate windows at this level. Then, with or without added detail, more specialized windows are generated until the desired detail is arrived at. The same process is also applied to SDBM types that serve to provide the data representation of the database contents and maintain semantic integrity constraints that are related to the data representation. The mechanism of aggregation is also used in
the design of an SDBM database. The application of this mechanism is demonstrated by allowing an object window (or a type) to participate in certain relationship without forcing the user to know the details of that window (or type).

In this chapter, we will first give a general description of the hypothetical project management system. Then, more detail will be added, where necessary, when we go through the actual design. Because the system is used only to demonstrate the features of SDBM in a concrete manner, we have tried to make the example as simple and short as possible. For example, some data types will not be defined if they are obvious and if no confusion is caused.

6.1 A HYPOTHETICAL PROJECT MANAGEMENT SYSTEM

A large manufacturing company often undertakes projects besides its normal operations. Some projects are launched internally. For example, the computer service department may install an electronic accounting system for the department of finance. Other projects are contracted from outside clients.

For normal operations, the company has an ordinary organizational structure. Namely, it is divided functionally into several departments such as administration, finance, marketing, engineering consulting, R&D, and manufacturing. A different structure is adopted for project management. For each project, a team of employees is set up by assigning employees from individual departments to the project.

Sometimes, extra people are hired specially for a project. When the project is finished, their employment is terminated. In this chapter, we call these people temporary employees, and other employees will be called permanent employees. They are hired by individual departments. A permanent employee of the company must belong to a department. When the project is completed, they will go back to their own departments.
For each project team, there is a principal manager. It is the company’s policy that only a permanent employee who has completed his/her training period is allowed to be a principal manager. A teams may also have an assistant manager who could be a temporary employee.

Permanent employees are paid monthly by their own department and temporary employees are paid hourly by their projects. For their overtime work, both of them will be paid hourly and rates vary with projects. The company does not allow any employee to work on more than two projects at the same time.

For outside clients, we have restricted their locations to be within North America. This restriction is solely for the purpose of simplifying the demonstration. A contract project will be terminated if the client ceases to exist. A team will be dissolvled if the project is terminated.

The operation of interest is assigning employees to projects, which includes assigning a permanent employee to a project and hiring a temporary employee for a project.

6.2 AN SDBM PROJECT MANAGEMENT DATABASE

Let us assume that all the necessary information requirements analysis has been completed so that we have the basic knowledge of the conceptual structure consisting of objects, properties, and relationships. To design the project management database in SDBM, we start with identifying different roles of objects and proceed to analyse the relationships and constraints among these roles. To simplify the definition process, we begin with the most general roles (views) of objects. Here, employees, projects and clients may be considered the most general roles, because many other roles can be derived through the specialization mechanism. To define the SDBM database schema, we have to give two kinds of specifications:
1. the conceptual structure of the application to be modelled. This concerns how objects, properties, and relationships among objects are viewed conceptually by the user.

2. the computer data representation. That is how the conceptual structure is represented in computer processable symbolic strings.

The former is specified by using the SDBM window concept and the latter is dealt with by defining proper data types in SDBM.

Let us start with employee objects. We want to record the ID, name, address, and phone number for each employee. These employees can be modelled as follows.

```plaintext
TYPE ( EmployeeType = RECORD WITH
    EmpID: EmpID,
    EmpName: EmpName,
    Address: EmpAddress,
    Phone#: Phone#
)

WINDOW ( Employee = TYPE: EmployeeType;
    KEY: ID
)
```

The meaning of data types EmpID, EmpName, EmpAddress, and Phone# are obvious. To give an example, we define data type Phone#.

```plaintext
TYPE ( Phone# = CHARACTER WITH '('999') '999'-'9999 OR
    999'-'9999
)
```

The legitimate telephone numbers are for example (604) 228-8651 and 224-1973. We have taken into consideration that it may not be necessary to record the area code for local phone numbers.
Assume that we are interested in the following information about each project: its name, code, category, start-date, end-date, budget, and total cost to date. The following are the definitions of the corresponding data type and window.

\[
\text{TYPE ( ProjectType } = \text{ RECORD WITH }
\begin{align*}
PjtName &: \text{ PjtName}, \\
PjtCode &: \text{ PjtCode}, \\
Category &: \text{ PjtCategory}, \\
Start-Date &: \text{ Date}, \\
End-Date &: \text{ Date}, \\
Budget &: \text{ Money}, \\
Cost &: \text{ Money}; \\
\end{align*}
\]

\[
\text{WINDOW ( Project } = \text{ TYPE: ProjectType; }
\begin{align*}
\text{KEY } &: \text{ PjtCode}
\end{align*}
\]

Data type PjtCategory can be defined as an enumeration data type.

\[
\text{TYPE ( PjtCategory } = \text{ ENUMERATION WITH }
\begin{align*}
\{\text{Consulting, Repair, Manufacturing, Research}\}
\end{align*}
\]

To model client objects, we assume that each client will be assigned a special ID. Other interesting information includes client name, address, and phone#. We have the following specification.

\[
\text{TYPE ( ClientType } = \text{ RECORD WITH }
\begin{align*}
\text{CltID } &: \text{ ClientID}, \\
\text{CltName } &: \text{ ClientName}, \\
\text{Address } &: \text{ ClientAddress}, \\
\text{Phone# } &: \text{ Phone#}
\end{align*}
\]
WINDOW ( Client= TYPE: ClientType;
    KEY: ID
 )

Among the data types ClientID, ClientName, ClientAddress, and Phone#, ClientAddress is the most interesting. ClientAddress can be modelled with the SDBM record data type. Its definition must take into consideration the difference between the American postal code and the Canadian postal code.

TYPE ( Address= RECORD WITH
    No : IntegerNo,
    Street : StreetName,
    City : CityName,
    VARIANT
    State : StateName,
    Country : {U.S.A.},
    PostalCode : USCode,
    VARIANT
    Province : ProvinceName,
    Country : {Canada},
    PostalCode : CanadianCode;
  )

Postal codes are defined as follows.

TYPE ( USCode = CHARACTER WITH 9(5)
  )

TYPE ( CanadianCode = CHARACTER WITH 9A9' '9A9
  )

The above specification dictates that if the country is U.S.A., we must write American state names and use the American postal codes, and if the country is Canada, we must write Canadian provincial names and use the Canadian postal codes.
So far, we have been concerned with information at a very general level. More details are needed. Let us consider project objects in a further step.

There are two types of projects. They are internal projects and contract projects. For internal projects, we need to know the names of their clients in addition to those properties defined for project objects in general.

```
TYPE ( InternalProjectType IS ProjectType WITH
       CltName: InternalClientName
    )
```

```
WINDOW ( InternalProject IS Project WITH
         TYPE: InternalProjectType
    )
```

For contract projects, additional properties which we are interested in include client ID, client name, contract code, and amount of money. Contract projects can be modelled as follows.

```
TYPE ( ContractProjectType IS ProjectType WITH
       CltID: ClientID,
       CltName: ContractClientName,
       ContractCode: ContractCode,
       Amount: Money;
    )
```

```
WINDOW ( ContractProject IS Project WITH
         TYPE: ContractProjectType;
         REFERENCE: Client(CltID=CltID)
    )
```

In the definition of window ContractProject, we have given a referential constraint. This constraint states that a contract project must have at least one outside client if it is
to be created in the database. At the same time, an outside client object cannot be
destroyed if it has at least one contract with the company.

Now, we consider some different roles of an employee. Like any other real world
objects, when viewed from different angles, an employee may appear to play different
roles at the same time. For example, in our case, an employee may be a permanent
employee or a temporary one, a trainee or a manager.

For temporary employees, the corresponding object class can be defined as follows.

\[
\text{WINDOW ( TemporaryEmployee IS Employee WITH}
\]
\[
\quad \text{TYPE: EmployeeType}
\]
\[
\)

Here, the data representation aspect of window TemporaryEmployee is exactly the
same as that of window Employee. This is an example which illustrates that it is not
necessary to add more details in order to generate a specialization. This example also
demonstrates the flexibility gained from separating the type concept and the window
concept. Here, data type EmployeeType has two sets of instances.

For a permanent employee, we need to know his/her department and the date he/she
started working for the company.

\[
\text{TYPE ( PermanentEmployeeType IS EmployeeType WITH}
\]
\[
\quad \text{DeptName: DepartmentName,}
\]
\[
\quad \text{Start-Date: Date}
\]
\[
\)

\[
\text{WINDOW ( PermanentEmployee IS Employee WITH}
\]
\[
\quad \text{TYPE: PermanentEmployeeType}
\]
\[
\)

\[
\text{TYPE ( DepartmentName= ENUMERATION WITH}
\]
\[
\quad \{\text{Administration, Finance, Marketing,}
\]

Among permanent employees, some are newly hired trainees. In addition to the properties that an employee and a permanent employee has, a trainee has a special property that he/she has a permanent employee as his/her supervisor. Let us first define those employees who are not trainees.

\[\text{WINDOW ( Nontrainee IS PermanentEmployee, ISNOT Trainee WITH TYPE: PermanentEmployeeType)}\]

\[\text{TYPE ( TraineeType IS PermanentEmployeeType WITH Supervisor: EmpID)}\]

\[\text{WINDOW ( Trainee IS Permanentemployee WITH TYPE: TraineeType; REFERENCES: Nontrainee(EmpID=Supervisor)}\]

In the above definition, we have specified constraints that the supervisor must exist, he/she must be a permanent employee, and he/she must not be another trainee. A special group of non-trainees are the principal managers of projects. They are defined as follows.

\[\text{TYPE ( ManagerType IS PermanentEmployeeType WITH PjtCode: ProjectCode,)}\]

\[\text{WINDOW ( Manager IS PermanentEmployee, ISNOT Trainee WITH TYPE: ManagerType; REFERENCE: Project(PjtCode=PjtCode)}\]
Obviously, the existence of a manager can only make sense if the project to be managed exists. All the objects we have so far dealt with are somewhat concrete. Now, let us define a collection of more abstract objects which are assignments. An assignment is specified to be the relationship that an employee is assigned to a project. The properties of an assignment include employee ID, employee name, project code, and date of the assignment.

\[
\text{TYPE (AssignmentType= RECORD WITH}
\]

\[
\begin{align*}
\text{EmpName: EmpName,} \\
\text{EmpID: EmpID,} \\
\text{ProjectCode: ProjectCode,} \\
\text{Date: Date}
\end{align*}
\]

\[
\text{WINDOW (Assignment}
\]

\[
\begin{align*}
= \text{TYPE: AssignmentType;} \\
\text{REFERENCE: Employee(EmpID= EmpId),} \\
\text{Project(PjtCode= ProjectCode);} \\
\text{KEY: EmpID, ProjectCode}
\end{align*}
\]

Conceptually, an Assignment object is defined to be an aggregation of an employee object and a project object. We enforce that both the employee object and the project object must exist in order to have the assignment. Alternatively, we can define an assignment to be an aggregation of an employee object and a project object without creating any new object.

We notice that assigning a temporary employee is different from assigning a permanent employee. To assign a temporary employee is to hire him or her for a project. He or she may be hired for several projects at the same time. Assume that the temporary
employee is paid by hour and that the minimum rate is $3 per hour. The hourly rate may be different for different projects. For simplicity, temporary-hiring is defined to be an assignment with the additional property Rate.

\[
\text{TYPE ( TemporaryHiringType IS AssignmentType WITH Rate: Rate; )}
\]

\[
\text{WINDOW ( TemporaryHiring IS Assignment WITH TYPE: TemporaryHiringType )}
\]

and data type Rate is defined to be

\[
\text{TYPE ( Rate = NUMERIC WITH 3..99 )}
\]

Another interesting concept is that of a project team which is the result of assigning employees to a project. It is reasonable to assume that a project has only one team and a team is formed only for one project.

Though this one-to-one relationship suggests that a project and its team may be represented by a single object, we notice their conceptual differences. Consider for example a project may exist before a team is created for it. Properties of the team object include project code, project name, principal manager, assistant manager, and number of employees in the team. Because a team may not have an assistant manager, the AssistantManager property is signified to be optional.
TYPE ( TeamType= RECORD WITH
    ProjectCode: ProjectCode,
    ProjectName: ProjectName,
    PrincipalManager: EmpID,
    AssistantManager: EmpID (OPTIONAL),
    NoOfEmployees: Integer
  )

WINDOW ( Team= TYPE: TeamType;
    REFERENCE: Manager(EmpID= PrincipalManager),
               Manager(EmpID= AssistantManager),
               Project(PjtCode= ProjectCode);
    KEY: ProjectCode
  )

The data types and windows declared above formed the basic structure of the hypothetical project management database. This part is sufficient to establish an empty database in the computer. However, we also need to define several operations in order to manipulate the database. SDBM database operations can be defined with the SDBM transaction facility.

6.3 TRANSACTIONS FOR END USER OPERATIONS

As we pointed out previously, transactions are the only means for the end user to operate on an SDBM database. Transactions are designed to ensure that certain behavioural constraints are observed when the database is altered by the end user. In this section, we will define several end user operations in terms of transactions.

First of all, we will define the operation of assigning an employee to a project team. To complete such an operation, we must first test that the employee has not been assigned to more than two teams. If this is true, a proper object is created and inserted into the corresponding window. The number of employees in the corresponding project
team must also be increased by one.

TRANSACTION ( Assign ( id: EmpID,
        n: EmpName,
        pc: ProjectCode,
        d: Date,
        OUT a: Assignment );

VARIABLE
        i: Integer,
        x: Assignment,
        t: Team;

BEGIN
        Count(id, i),
        IF i GT 3
            THEN ABORT
            ELSE CREATE x WITH
                ( EmpName=n,
                EmpID=id,
                ProjectCode=pc,
                Date=d ),
            RETRIEVE t
                WHERE ( ProjectCode=pc),
                t.NoOfEmployee= (t.NoOfEmployee + 1)
        END
)

Of course, we assume that the project has been initiated. In the transaction Assign, another transaction Count is called to find out how many teams the employee has already been assigned to. This is a very simple transaction. In fact it does not alter anything in the database.
TRANSACTION ( Count ( id: EmpID, OUT i: Integer );

VARIABLE
   y: Assignment;

BEGIN
   i=0,
   RETRIEVE y WHERE (EmpID=id),
   WHILE y NE NULL DO
      BEGIN
         i = i + 1
         RETRIEVE y
         WHERE (EmpID=id)
      END
   END
)

More realistically, when we assign an employee to a project team, we usually know what kind of employee he/she is. Therefore, the assignment operations special to that kind of employee should also be defined. Consider for example the operation of hiring a temporary employee, which is defined as follows.

TRANSACTION ( HireTemporary ( id: EmpID,
                                 n: EmpName,
                                 ta: EmpAddress,
                                 ph: Phone#,
                                 p: ProjectCode,
                                 d: Date,
                                 r: Rate );

VARIABLE
   a: AssignmentType,
   e: Employee,
   te: TemporaryEmployee,
   th: TemporaryHiring;
BEGIN
CREATE e WITH
( ID=id,
  Name=n,
  Address=ta,
  Phone#=ph),
INSERT te(e) INTO TemporaryEmployee,
Assign(id,n,p,d,a),
INSERT th(a) INTO TemporaryHiring
  WITH ( Rate=r)
END
)

6.4 SUMMARY

In this chapter, a hypothetical project management database is designed to demonstrate the concept, the structure, and the modelling methodology of the semantic database model SDBM. Although many examples have been given in chapter 5, they may appear to be fragmented. Our goal in this chapter is to give a more coherent picture of the model.
7. CONCLUSIONS

In this thesis, we have addressed the shortcomings associated with conventional data models, particularly the relational data model. We have also discussed the features of recently proposed semantic data models. Based on this investigation, a semantic database model, SDBM, is developed.

SDBM is designed to attack the modelling problems primarily of the relational data model and to offer some improvements over other semantic data models. The rest of this chapter will summarize some major ideas presented in this thesis.

First of all, we conceptualize data modelling in a way different from other data models. The conceptualization itself is simple and effective. What is more important is that our model directly reflects this conceptualization.

We believe that data modelling should be viewed from two sides. From one side, we see objects, properties, and relationships among these objects. This image is a one-to-one mapping of the real world situation to be modelled. From the other side, we see their computer data representations. We realize that it is not objects, nor properties, nor relationships among objects that should be or can be directly reflected by these computer data representations. Rather, it is their roles that should and can be directly reflected by these computer data representations. Roles are certain aspects of objects. Viewed from a more general point, a role is an abstraction of a collection of properties. Roles can be modelled with SDBM windows. Notice that there are similarities between the SDBM window concept and the class concept of other semantic data models. However, because the SDBM window concept is built on a different conceptualization, it is conceptually different from the class concepts of other semantic data models.
As mentioned before, we envision that there exists a conceptual structure or world consisting of objects, properties, and relationships, which is a one-to-one correspondence of the 'real' world. Open a window. We see some part of the structure. This part is a role which is characterized by a collection of properties. These properties could manifest either an object or a relationship. Open another window in a different direction, and we see another role possibly being played by the same object. No matter how many roles we have seen, the underlying structure as a whole is the same.

Clearly, the concept of window in SDBM extends the concept of class in other semantic data models. Talking about instances of a class as in other semantic data models, we find that 'instances' of an SDBM window can be either objects or relationships among these objects, while the instances of a class of other semantic data models can only be objects. Therefore, SDBM accommodates views which are more flexible for the user.

The object concept in SDBM is also different from the object concepts found in other semantic data models. In other semantic data models, an object is always rigidly tied to one role or several roles if these roles have compatible external data representations in terms of the data subtype hierarchy. However, in SDBM such rigidity is removed as a result of applying our two-view conceptualization. An SDBM object can have totally different external data representations. This extension of the object concept greatly enhances the data modelling ability.

Based on our conceptualization, we investigated the relationship between the type concept and the window concept. The type concept in SDBM is actually a tool to make it possible to display our vision of the 'real' world being modelled. We found that there should be explicit separation between the type concept and the window concept and that a semantic data model should support the many-to-many relationship between these two concepts. However, this has not been supported by many recently proposed semantic data models.
We have developed the SDBM so that such separation is built into it. It has been demonstrated that this separation and this many-to-many relationship give the model enhanced modelling power. Following such separation, we classify semantic integrity constraints into two categories, those associated with types and those associated with windows.

Secondly, a database design methodology based on our two-view conceptualization is presented. This methodology expands the design methodology proposed in TAXIS (Mylopoulos et al [1980]). Like TAXIS, our methodology also promotes the concept of specialization. However, unlike TAXIS, our approach does not just start with the most general level. It starts with objects that are both general and conceptually significant on their own. The design of an SDBM database also needs to go through a role identification process. In this process, we identify all the interesting roles for objects and analyse relationships that may exist among them. Thus, we expect that this approach will improve the database specification because of the enhanced modelling power.

Thirdly, because the model has a built-in type system, well-established type checking techniques in programming languages can be readily introduced to database applications.

Finally, several new modelling constructs are introduced to support our conceptualization of data modelling. The SDBM record data type is developed as the basic external data representation tool for SDBM roles. As demonstrated in examples, this data structure is flexible in modelling various situations.

The relationships among windows have been investigated. As a result, constructs like MAYBE and ISNOT are introduced to model the constraints between different roles that might be played by the same object. For the same purpose, dynamic modelling constructs such as INSERT and DELETE are also introduced. To cope with the difficulty associated with multi-valued properties, several behavioral operations are provided. They can add a
value to the property, remove a value from the property, and modify a multi-valued property. We have pointed out that the specialization process may be carried out without additional details.

Due to time constraint, the semantic database model SDBM has not been implemented on the computer. It should be interesting to investigate how our two-view conceptualization can be realized internally. It should also be interesting to investigate the user specifiable exceptional handling facility at the conceptual level. Currently, such facility is not supported.
BIBLIOGRAPHY


Bachman, C.W. [1977], 'The Role Concept in Database Models', Proc. of Int. Conf. on Very Large Data Bases, Tokyo, Japan, October 1977.


Brodie, M.L. [1978], 'Specification and Verification of Data Base Semantic Integrity', Technical
Report CSRG-91, Univ. of Toronto, April 1978.


Codd, E.F. [1974b], ‘Seven Steps to Rendezvous with the Casual User’, In Klimbie, J.W. et al. (Eds.), DATA BASE MANAGEMENT, North Holland, 1974.


Date, C.J. [1983], AN INTRODUCTION TO DATABASE SYSTEM, Vol.II, Addison-Wesley, 1983.

Fagin, R. [1975], 'The Decomposition Versus the Synthetic Approach to Database Design', Proc. of Int. Conf. on Very Large Data Bases, Tokyo, Japan, October 1977.


Hammer, M.M. and McLeod, D.J. [1975], 'Semantic Integrity in a Relational Data Base system', Proc. of Int. Conf. on Very Large Data Bases, Framingham MA, USA, Sept. 1975.


Kent, W. [1977], 'Entities and Relationships in Information', In Nijssen, G.M., (Ed.),
ARCHITECTURE AND MODELS IN DATA BASE MANAGEMENT SYSTEMS,


McLeod, D.J. [1976a], 'High Level Domain in a Relational Data Base System', Proc. of ACM SIGPLAN/SIGMOD Conf. on Data: Abstraction, Definition, and Structure, Salt Lake City, March 1976.


Rowe, L. and Shoens, K. [1979], 'Data Abstraction, Views and Updates in RIGEL', Proc. of ACM SIGMOD Int. Conf. on the Mgt. of Data, Boston MA, 1979.


Schmidt, J.W., [1978], 'Type Concepts for Database Definition', Proc. of Int. Conf. on Data Bases, Haifa, Israel, 1978.


APPENDIX A. SYNTAX OF SDBM

<Letter> :=
A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<Digit> := 0|1|2|3|4|5|6|7|8|9

<LetterOrDigit> := <Letter> | <Digit>

<SpecialCharacter> :=
+|-|*|/|=|,|;|#|$|&|{}|[]|[|"|(|)|(|)

<Character> := <LetterOrDigit> | <SpecialCharacter>

<CharacterString> := <Character>{<Character>}

<Identifier> := <Letter>{<LetterOrDigit>}

<KeyWord> := NUMERIC | CHARACTER | BOOLEAN | VARIANT | COMMON | BEGIN | END | NULL | ISNOT | MAYBE | LOCAL | IS | WITH | ADT | TYPE | ADTOP | FUNCTION | ACTION | MULTIPLE | OPTIONAL | KEY | OF | INTO | TRANSACTION | OUT | IN | VARIABLE | INSERT | MODIFY | DELETE | RETRIEVE | DESTROY | CREATE | TO | WINDOW | REFERENCE | RELATIONSHIP | WHERE | IF | THEN | ELSE | ENDIF | FOR | EACH | DO | WHILE | GT | GE | LT | LE | EQ | NE | AND | OR | NOT | <--> | ADD | REMOVE | TRUE | FALSE

<UnsignedInteger> := <Digit>{<Digit>}

<PositiveInteger> := <UnsignedInteger> | +<UnsignedInteger>
<NegativeInteger> ::= -<Digit>{<Digit>}

<Integer> ::= <PositiveInteger> | <NegativeInteger>

<UnsignedReal> ::= <Digit>{<Digit>}.{<Digit>}

<Real> ::= <UnsignedReal> | +<UnsignedReal> | -<UnsignedReal>

<Number> ::= <Real> | <Integer>

<Constant> ::= <UnsignedInteger> | <UnsignedReal> |
             "<CharacterString>" | NULL

<SignedConstant> ::= <Number> | "<CharacterString>" | NULL

<SDBMSchema> ::= <TypeDeclarationPart>
       <WindowDeclarationPart>
       [ <TransactionDeclarationPart> ]

<TypeDeclarationPart> ::= <TypeDeclaration>
       { <TypeDeclaration> }

<TypeDeclaration> ::= TYPE ( <TypeID>
       [ = <TypeDefinition> | <SubtypeDefinition> ]+
       )

<TypeDefinition> ::= <AnyTypeID> WITH <TypeConstraint>

<AnyTypeID> ::= NUMERIC | BOOLEAN | CHARACTER |
               ENUMERATION | RECORD | <TypeID>

<TypeConstraint> ::= <NumericRanges>|<BooleanPicture> | <BooleanPicture> |
                   <CharacterPictures>|
                   <EnumerationPicture>|
                   <RecordPicture>
<NumericRanges> ::= <NumericRange> \{OR <NumericRange>\}

<NumericRange> ::= <Number>..<Number>

<BooleanPicture> ::= "{"TRUE|FALSE|TRUE,FALSE|FALSE,TRUE"}"

<CharacterPictures> ::= <CharacterPicture>
                      \{OR <CharacterPicture>\}

<CharacterPicture> ::= <NumericCharacter> | <AlphabeticCharacter> | <AlphanumericCharacter>

<NumericCharacter> ::= 9(<Integer>)

<AlphabeticCharacter> ::= A(<Integer>)

<AlphanumericCharacter> ::= X(<Integer>)

<CharacterList> ::= \{A|X|9<QuateList>\}+

<QuateList> ::= "<Character>{<Character>}" 

<EnumerationPicture> ::= "{"<EnumerationList>"}"

<EnumerationList> ::= <CharacterString>{, <CharacterString>}

<RecordStructure> ::= [[VARIANT|COMMON] 
                      <ComponentAndTypePair> {, 
                      [VARIANT|COMMON] 
                      <ComponentAndTypePair> }];]

[BEGIN
  <ConstraintAssertion>{, 
  <ConstraintAssertion>}


```plaintext
<ComponentAndTypePair>::=
    <ComponentID>::<TypeSpecification>[MULTIPLE|OPTIONAL]

<ComponentID>::=<Identifier>

<TypeSpecification>::=<TypeID>|<TypeDefinition>|<SubtypeDefinition>

<ConstraintAssertion>::= <ComplexAssertion>|<SimpleAssertion>

<SimpleAssertion>::=
    <ConTerm> {<BooleanOperator> <ConTerm>}

<BooleanOperator>::= AND | OR | NOT

<ConTerm>::=<Element>
    {<RelationalOperator> <Element>}

<RelationalOperator>::= GE | GT | LE | LT | EQ | NE

<ConPrimary>::= ( <ConstraintAssertion> ) |<ComponentID> | <Constant>

<ComplexAssertion>::= IF <ConstraintAssertion>
    THEN <ConstraintAssertion>
    [ ELSE <ConstraintAssertion> ]
    ENDIF

<SubtypeDefinition>::= IS <TypeDefinition>

<ADTDefinition>::=
    ADT(<ADTID>=TYPE <TypeID> [LOCAL];
        ADTOP [<ADTFункцияOP>|<ADTПроцедураOP>]^{,}
        [<ADTFункцияOP>|<ADTПроцедураOP>]^{,})
```

ACTION:

\[
\text{<ADTProcedureOrFunction>}
\]

\[
\text{<ADTFunctiOP> := \text{<ADTFunctiID>(<FunctionParameters>):<TypeSpeciication>}}
\]

\[
\text{<ADTFunctiOP> := \text{<ADTFunctiID> := <Identifier>}}
\]

\[
\text{<ADTID> := <Identifier>}
\]

\[
\text{<FunctionParameters> := <ParameterID>:<TypeSpeciication>{,}
\]

\[
\text{<ParameterID> := <Identifier>}
\]

\[
\text{<ADTProcedureOP> := <ADTProcedureID>(<ProcedureParameters>)}
\]

\[
\text{<ADTProcedureID> := <Identifier>}
\]

\[
\text{<ProcedureParameters> := [OUT]<ParameterID>:<TypeSpeciication>{,}
\]

\[
\text{[OUT]<ParameterID> := <Identifier>}
\]

\[
\text{<ADTProcedureOrFunction> :=}
\]

\[
\text{<ADTProcedure> | <ADTFunctiOP> | <Procedure> |}
\]

\[
\text{<Function>}
\]

\[
\text{<ADTProcedure> := PROCEDURE <ADTProcedureID>}
\]

\[
\text{<Body> END PROCEDURE <ADTProcedureID>}
\]

\[
\text{<Body> := [ <ADTTypedeclarationPart> ]}
\]

\[
\text{[ <ADTVariabuleDeclaration> ]}
\]

\[
\text{<ADTStatementPart>}
\]

\[
\text{<ADTTypedeclarationPart> := TYPE <ADTTypedeclaration>}{,}
\]
\[ \text{ADTTypeDeclaration} \]

\[ \begin{align*}
\text{ADTTypeDeclaration} & \::= \text{TypeID} \\
& \quad [= \text{TypeDefinition} \mid \text{SubtypeDefinition}]^+ \\
\text{ADTVariableDeclarationPart} & \::= \\
& \quad \text{VARIABLE} \text{VariableList} : \text{TypeID}, \\
& \quad \text{VariableList} : \text{TypeID} \\
\text{ADTStatementPart} & \::= \text{BEGIN} \\
& \quad \text{ADTStatement}, \\
& \quad \text{ADTStatement} \text{END} \\
\text{ADTStatement} & \::= \text{ADTAssignment} \mid \text{ADTPrecedureCall} \mid \\
& \quad \text{ADTCompound} \mid \text{ADTCondition} \mid \\
& \quad \text{ADTIteration} \\
\text{ADTAssignment} & \::= \text{VariableID} = \text{ADTExpression} \mid \\
& \quad \text{FunctionID} = \text{ADTExpression} \\
\text{ADTPrecedureCall} & \::= \text{PrecedureID} ([\text{ActualParameter}, \\
& \quad \text{ActualParameter}]]) \\
\text{ActualParameter} & \::= \text{ADTVariable} \mid \text{SignedConstant} \\
\text{ADTVariable} & \::= \text{ADTVariableID} \mid \text{ADTVariableID}.\text{ComponentID} \\
\text{ADTVariableID} & \::= \text{VariableID} \\
\text{ADTCompound} & \::= \text{BEGIN} \\
& \quad \text{ADTStatement}, \\
& \quad \text{ADTStatement} \text{END} 
\end{align*} \]
\[ \text{<ADTCondition>:= IF <ADTExpression> THEN <ADTStatement> [ELSE <ADTStatement>]} ENDIF} \]

\[ \text{<ADTIteration>:= <ADTFor>|<ADTWhile}> \]

\[ \text{<ADTFor>:= FOR <VariableID>=<Initial> TO <Final> DO <ADTStatement>} \]

\[ \text{<Initial>:= <Integer>} \]

\[ \text{<Final>:= <Integer>} \]

\[ \text{<ADTWhile>:= WHILE <ADTExpression> DO <ADTStatement>} \]

\[ \text{<ADTExpression>:= <ADTTerm> <RelationalOperator> <ADTTerm>| <ADTTerm>} \]

\[ \text{<ADTTerm>:= <ADTUnaryTerm> <AddingOperator> <ADTUnaryTerm>| <ADTUnaryTerm>} \]

\[ \text{<AddingOperator>:= +|-|OR} \]

\[ \text{<ADTUnaryTerm>:= <ADTFactor>|+<ADTFactor>|-<ADTFactor>} \]

\[ \text{<ADTFactor>:=<ADTPrimary> <MultiplyingOperator> <ADTPrimary>| <ADTPrimary>} \]

\[ \text{<MultiplyingOperator>:= */|AND} \]

\[ \text{<ADTPrimary>:= NOT <ADTElement>|<ADTElement>} \]

\[ \text{<ADTElement>:= (<ADTExpression>)|<ADTVariable>| <FunctionCall>|<ADTFunctionCall>} \]
<FunctionCall> ::= <FunctionID>([<ActualParameter>{, <ActualParameter>}])

<ADTFunctonCall> ::= <ADTFunctonID>([<ActualParameter>{, <ActualParameter>}])

<ADTFuncton> ::= FUNCTION <ADTFunctonID>
   <Body>
   END FUNCTION <ADTFunctonID>

<Procedure> ::= PROCEDURE <ProcedureID>
   <Body>
   END PROCEDURE <ProcedureID>

<WindowDeclarationPart> ::= <WindowDeclaration>
   {<WindowDeclaration>}

<WindowDeclaration> ::= <SimpleWindowDeclaration> | <AggregateWindowDeclaration>

<SimpleWindowDeclaration> ::= WINDOW (<WindowID> [IS <WindowID> WITH | =]+
   [TYPE: <TypeID>{,<TypeID>;}
   [REFERENCE: <References>;]
   [RELATIONSHIP: <Relationships>;]
   [KEY: <KeyIDs>]
   )

<WindowID> ::= <Identifier>

<KeyID> ::= <PropertyID> {,<PropertyID> }
<PropertyID> := <Identifier>

<References> := <WindowID>({<PropertyPairs>}),
               <WindowID>({<PropertyPairs>})

<PropertyPairs> := <PropertyID>=<PropertyID>{,<n
                  <PropertyID>=<PropertyID>}

<Relationships> := <Relationship>{,<n
                  <Relationship>

<Relationship> :=
               <RelationshipID>({<WindowID>({<PropertyPairs>})
        ) <--> <RelationshipID>

<AggregateWindowDeclaration> :=
       WINDOW (<WindowID>= TYPE: <AggregateTypeDefinition>{,
            <AggregateTypeDefinition>};
            AGGREGATE: <WindowID>,
            <WindowID>[,<n
                <WindowID>

<AggregateTypeDefinition> :=
       <ComponentID> [OF <WindowID>|: <TypeSpecification>]*

<TransactionDeclarationPart> := <TransactionDeclaration>
          {[<TransactionDeclaration>]

<TransactionDeclaration> :=
       TRANSACTION (<TransactionDesignator>
          <TransactionBody>

<TransactionDesignator> :=
       <TransactionID> ([<ParameterDefinitionList>])]
<TransactionID> := <Identifier>

<ParameterDefinitionList> := <ParameterDefinition>{, <ParameterDefinition>}

<ParameterDefinition> := <OUTParameter> | <INParameter>

<OUTParameter> := OUT <ParameterID>:<TypeIDOrWindowID>

<INParameter> := [IN] <ParameterID>:<TypeIDOrWindowID>

<TypeIDOrWindowID> := <TypID> | <WindowID>

<TransactionBody> := [<TransactionTypeDeclaration>] [<TransactionVariableDefinition>] BEGIN <Statement>{, <Statement>} END

<TransactionTypeDeclaration> := TYPE <TypeID> [=<TypeDefinition>|<SubtypeDefinition>]^{, <TypeID> [=<TypeDefinition>|<SubtypeDefinition>]^{}}

<VariableDeclaration> := VARIABLE <VariableList>: <TypeSpecification>|<WindowID>{, <VariableList>: <TypeSpecification>|<WindowID>};

<VariableList> := <VariableID>{, <VariableID>}

<VariableID> := <Identifier>

<Statement> := <AssignmentStatement> |
<CreateStatement> | <DestroyStatement> |  
<InsertStatement> | <ModifyStatement> |  
<DeleteStatement> | <RetrieveStatement> |  
<AbortStatement> |  
<CompoundStatement> | <ConditionalStatement> |  
<IterationStatement> | <TransactionCall>|  
<ADTPredicateCall> | <AddStatement> |  
<RemoveStatement> |  
<Multi-valuedPropertyModifyStatement>  

<AssignemntStatement>:= <VariableID>=<Expression>  

<CreateStatement>:=  
CREATE <VariableID> INTO <WindowID> WITH  
<PropertyValueList>)  

<PropertyValueList>:= <PropertyValuePair>{,  
<PropertyValuePair>}  

<PropertyValuePair>:=  
<PropertyID>="<CharacterStrings>"|<VariableID>  

<CharacterStrings>:= '<CharacterString>'{,'<CharacterString>'}  

<DestroyStatement>:= DESTROY <VariableID>  

<InsertStatement>:=  
INSERT <VariableID>({;<VarID>{,<VarID>}})  
INTO <WindowID> [ WITH  
(<PropertyValueList>)]}  

<VarID>:= <Identifier>  

<ModifyStatement>:=  
<VariableID>.<PropertyID>= <NewValue>
<NewValue>:= "<CharacterString>" | <Expression>

<DeleteStatement>:= DELETE <VariableID>

<RetrieveStatement>:=
    RETRIEVE <VariableID>[.<PropertyID> TO <VariableID>],
    [ WHERE <Expression> ]

<AbortStatement>:= ABORT

<AddStatement>:= ADD [<VariableID>|"<CharacterStrings>"]^ TO <VariableID>.<PropertyID>

<RemoveStatement>:=
    REMOVE [<VariableID>|"<CharacterStrings>"]^ FROM <VariableID>.<PropertyID>

<Multi-valuedPropertyModifyStatement>:=
    <VariableID>.<PropertyID>=
    [ NULL|"<CharacterStrings>" ]<VariableID>.<PropertyID>

<CompoundStatement>:= BEGIN
    <Statement> { ,
    <Statement> }
    END

<ConditionalStatement>:= IF <Expression>
    THEN <Statement>
    [ ELSE <Statement> ]
    ENDIF

<IterationStatement>:=<ForStatement>|<WhileStatement>

<ForStatement>:= FOR EACH <VariableID> DO
    <Statement>
<WhileStatement> ::= WHILE <Expression> DO
  <Statement>

<Expression> ::= <Term> {<RelationalOperator> <Term>}

<Term> ::= <UnaryTerm> {<AddingOperator> <UnaryTerm>}

<UnaryTerm> ::= <Factor> | +<Factor> | -<Factor>

<Factor> ::= <Primary> {<MultiplyingOperator> <Primary>}

<Primary> ::= NOT <Element> | <Element>

<Element> ::= (<Expression>) | <VariableID> |
  <VariableID>.<ComponentID> | <VariableID>.<PropertyID> |
  <ADTFunctionalCall> | <Constant>

<TransactionCall> ::= <TransactionID>({<ActuralParameter>{,
  <ActuralParameter>}})
## APPENDIX B. COMPARASONS BETWEEN SDBM AND TAXIS

### B.1 SIMILARITIES

<table>
<thead>
<tr>
<th>SDBM</th>
<th>TAXIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modelling primitives are objects.</td>
<td>Modelling primitives are objects.</td>
</tr>
<tr>
<td>There can be dependency constraints.</td>
<td>There are dependency constraints among objects.</td>
</tr>
<tr>
<td>Objects have properties.</td>
<td>Objects have properties.</td>
</tr>
<tr>
<td>Object property constraints can be expressed with assertions.</td>
<td>Object property constraints can be expressed with assertions.</td>
</tr>
<tr>
<td>Relationships exist between objects.</td>
<td>Relationships exist between objects.</td>
</tr>
<tr>
<td>Basic operations are available, including insertion, deletion, modification, and retrieval.</td>
<td>Basic operations are available, including insertion, deletion, modification, and retrieval.</td>
</tr>
<tr>
<td>Variables may be used as object surrogates in transactions.</td>
<td>Variables are used as object surrogates in transactions.</td>
</tr>
</tbody>
</table>

### B.2 DIFFERENCES

<table>
<thead>
<tr>
<th>SDBM</th>
<th>TAXIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects have no types. The external data representation is handled by windows.</td>
<td>Objects are categorized into classes.</td>
</tr>
<tr>
<td>Instance hierarchy is trivial.</td>
<td>Instance hierarchy is extensively used.</td>
</tr>
</tbody>
</table>
Dependency (reference) constraints exist only if specified explicitly.

Many-to-many relationships between two objects may be established directly without introducing an artificial construct.

The existence of references is different from the existence of relationships.

Semantics associated with the name of a relationship can be explicitly expressed.

Properties can be multi-valued.

Specialization mechanism is only a special constraint among different windows of an object.

Data types are used extensively to specify the external data representations of objects.

Intra-data-type constraints are specified within data type definitions.

Data types are differentiated from windows. A window can have more than one data type and a data type can be used with more than one window.

Dependency constraints always exist.

Many-to-many relationships between two objects must be established through creating a third object (class).

The existence of references is not differentiated from the existence of relationships.

Semantics associated with the name of a relationship can be implied through syntactic combination of the properties of a class.

Properties must be single-valued.

Specialization mechanism is the only relationship that can be specified among classes.

Only data type is the TAXIS class.

Intra-relation constraints are specified through transactions.

Data types and classes are mixed up. They are all Taxis classes.
Objects and symbolic strings are explicitly differentiated.

Symbolic strings are treated as objects.

Abstract data types are supported.

Abstract data types are not supported.

The underlying conceptualization is the two-view conceptualization.
Appropriate constructs are introduced to support this conceptualization.

The two-view conceptualization is not fully supported.

SDBM is typed.

SDBM is typed.

TAXIS is not typed.