MODULARITY IN DATA BASE SYSTEM DESIGN

H. Weber

Rapporteur: Dr. S.K. Shrivastava

Abstract

This lecture focuses on the application of software engineering techniques for the development of data base systems. It is aimed to explain a key concept in software engineering - the module or abstract data type concept - and to demonstrate its usefulness in data base system structuring and design. The concept is shown to provide a basis for the development of a uniform framework, called D-Graph, for the structuring of the data base, its conceptual description and of the data base management system.

1. Introduction

Data base systems, as with any other kinds of software systems, are expensive to produce and maintain, and usually of low quality. In a great number of empiric studies the lack of a rational software technology has been identified as the main reason for this so-called software dilemma. Thus, the identification and formulation of fundamental principles for the development of software systems in general, and for the production of data base systems as well, are now of growing interest.

Due to the very well-known human limitations in dealing with complexity, both the complexity of the system development task, and the complexity of the system itself, are considered to be accountable for the difficulties in the development of large or even rather small, software systems. In order to manage these complexities, technologies are now available, or are under development, which impose a discipline on software production and structuring.

A programming concept central to the discussion about a software technology for a number of years -frequently termed module or abstract data-type is considered to be the base for a solution of the principal problem mentioned above (PAR 71, PAR 72, PAR 74, LIS 74, LIS 77, WIR 77, WUL 76). Consequently, this paper is aimed at explaining this concept and demonstrating the impact of its use in data base system development.

2. The Module Concept

The recent history in programming has proven that the language used to formulate programs influence the style of programming and consequently the structure of the programs. It is therefore assumed that choosing the right language features may also encourage good programming. The module concept explained below is considered to

support the production of well-structured, reliable, robust and verifiably correct programs. The concept will be introduced here step by step. For that purpose, we first give a definition of the concept in BNF notation (for those readers who find descriptions in a meta language more comprehensible) and explain then its characteristics and advantages with a simple but nontrivial example (for those readers who prefer a more informal explanation of the subject) in a number of iterations.

2.1 Basic Defintions

In a first iteration the module concept may be defined as follows:

⟨module⟩ ::= ⟨interface⟩⟨body⟩
⟨interface⟩::= module ⟨module identifier⟩ ⟨operator list⟩
⟨body⟩ ::= begin ⟨data definition⟩{⟨procedure definition⟩;}^{n} end

(The curly braces are used to denote none or more repetions of the enclosed concepts.) According to this definition, a module consists of two parts: an interface and a body. The interface contains a set of identifiers which may be referenced to gain access to the body of the module. The body is a program in a suitable programming language.

A sample module definition using a high-level language notation would then take the following format.

module FLIGHTSCHEDULE (op_1,op_2,...,op_n)
begin modulebody
:
end modulebody

This definition introduces a module called FLIGHTSCHEDULE. It is worth noting here that the module is named after a certain sort of data, thus indicating that the module has been designed to be invoked for the creation and manipulation of data objects of this sort. The interface of the module identifies, therefore, this sort of data, and all the operations applicable on those data. Since the text of the module body is not of interest for the following discussion, we will ignore it for the moment.

A number of terms closely related to this notion of module, and equally important, for the understanding and further explanation of the module concept will be defined now in an informal way.

The desired relationship between all legal inputs and the possible outputs of all the operations of a module will be called the functions of the module.

The specification of a module is an implementation-independent description of the functions of that module.

The implementation of a module - its body - is the program text for the data definition part (denoting the implementation of the sort of data pertinent to that module), and for all the procedure definitions (denoting the implementation of the operations pertinent to that module).

A program consists of an arbitrary number of module definitions, as explained above, and an arbitrary number of statements each referencing a certain module and one operation defined in that module.

A process denotes the execution of module operations for an appropriate set of input parameters.

2.2 The Definition of a Module Interface

In a next refinement step we will first complete the definition of a module interface. Starting from the previous definition the more detailed description of the interface may then be given as follows:

⟨operator list⟩ ::= (⟨operator symbol⟩ [⟨parameter list⟩]
{, ⟨operator symbol⟩ [⟨parameter list⟩] }^{n})

⟨parameter list⟩ ::= ⟨parameter symbol⟩ {, ⟨parameter symbol⟩}^{k}

⟨module identifier⟩ ::= CHARACTER STRING
⟨operator symbol⟩ ::= CHARACTER STRING
⟨parameter symbol⟩ ::= CHARACTER STRING

According to this definition each of the n legal operations on the sort of data identified in the interface gets a set of k parameters associated with it. For the execution of an operation the k parameters declared for this operation need to be passed to the module.

The example introduced in the previous section will be used to explain this feature.

```
module FLIGHTSCHEDULE  (create-schedule [id],
                        search-flight [id, f#],
                        schedule-flight [id, f#, dest, st-t],
                        cancel-flight [id, f#]);
```

This interface identifies the FLIGHTSCHEDULE sort of data and four operations on it. It is the function of the create-schedule operation to create a data object of the sort FLIGHTSCHEDULE. For its execution the parameter "id" denoting the data object's identifier must be passed to the module in an operation call.

The function of the search-flight operation is to access and display a flight identified by a certain flight number f# which is assumed to be recorded within a flightschedule data object with identifier id. With the schedulflight operation one may record a new flight, that is, the flight number, f#, the destination, dest, and the start time, st-t, within the flightschedule data object id. Finally, with the cancel-flight operation one may delete the entry identified by f# in the flightschedule data object id.

It is obvious now that a module may be used to create and manipulate an arbitrary number of data objects with different identifiers. Since the implementation and consequently the function of the operations will be the same for all objects, a module may be considered as a template for the creation and manipulation of data objects which exhibit exactly the same properties.

Because of this characteristic, the module concept closely resembles the data type concept in high-level programming languages: A variable denoting a data object may be declared to be of a certain type thus determining the properties of the object, that is, its possible manipulations. The type implementation is part of the compiler and hidden for its users. Because of these similarities the module concept is also frequently called abstract data type concept.

The term type will therfore be used in the sequel to denote the properties of a data object and also to refer to the module which implements the operations for the creation and possible manipulation of all objects of a given type.

2.3 The Definition of a Module Body

The module body contains the implementation of the so-called abstract objects and abstract operations as identified in the interface of the module. For the implementation of abstract objects and operations they need to be represented in terms of machine-supported data objects of so-called representing type, and operations on objects of representing type. Data objects and operations of representing types may be either primitive machine types or compositions of private type.

Thus, the data definition part will be expressed in BNF notation as follows:

⟨data definition⟩ ::= rep as ⟨constructor⟩ of ⟨module identifier⟩
⟨constructor⟩ ::= CHARACTER STRING
⟨module identifier⟩::= CHARACTER STRING

The term "constructor" in the defintion above denotes a structuring concept which is applied to compose objects of representing type from objects of component type. This composition of the representing types from component types (or from one component type) is defined in a module which implements the representing type. Thus, the term constructor refers to a certain type which is called the representing type. Data and operations of an abstract type as identified in a module interface are then implemented in terms of objects and operations of the (machine-supported) representing type which defines at the same time, a composition of component types. For the previously introduced example the data definition may have the following format:

```
module FLIGHTSCHEDULE (create-schedule[id],...)
rep as FILE of FS-RECORD
```

A data object of type FLIGHTSCHEDULE is represented by an object of the type FILES which in turn is composed - in a way defined in a FILE module - of a number of objects of type FS-RECORD. Thus the implementation of the types FILE and FS-RECORD is a prerequisite for the implementation of the type FLIGHTSCHEDULE.

The procedure definition part expressed in BNF will be given as follows:

```
⟨procedure definition⟩::= proc ⟨procedure head⟩⟨procedure body⟩
⟨procedure head⟩      ::= ⟨operator symbol⟩[⟨parameter list⟩]→ ⟨result⟩
⟨result⟩              ::=     ⟨data object⟩|BOOLEAN
⟨data object⟩         ::= ⟨object identifier⟩:⟨module identifier⟩
⟨procedure body⟩      ::= {⟨statement⟩;}^n
⟨statement⟩           ::= ⟨operation call⟩|⟨conditional statement⟩|
                          ⟨unconditional statement⟩|⟨for statement⟩
⟨operation call⟩      ::= ⟨module identifier⟩. ⟨operator symbol⟩
                          [⟨parameter list⟩]
```

(The remaining undefined nonterminal symbols in this grammar have either been defined before or should be understood as in the definition of a high-level language like ALGOL 60.)

An abstract operation is implemented by a procedure which performs operations on objects of representing type. Thus, one may include calls of operations on objects of representing type - as defined in the module for the representing type - within this procedure.

On the basis of these definitions and based on the assumption that a data type FILE (create file, search record, insert record, delete record) has already been defined, one may now give a complete program text for the FLIGHTSHEDULE module in the following form:

```
module FLIGHTSCHEDULE   (create-schedule [id],
                         search-flight [id, f#],
                         schedule-flight [id, f#, dest, st-t],
                         cancel-flight [id, f#]);

begin

rep as FILE of FS-RECORD

proc create-schedule [id] ⟶ id:FLIGHTSCHEDULE;
id:FILE:=FILE. create-file [o];
end create-schedule;

proc search-flight [id, f#] ⟶ BOOLEAN;
FILE. search-record [id, f#];

proc cancel-flight [id, f#] ⟶ id:FLIGHTSCHEDULE;
FILE. delete-record [id, f#];
end cancel-flight;

proc schedule-flight [id, f#, dest, st-t] ⟶ id:FLIGHTSCHEDULE;
FILE.insert-record [id, f#, dest, st-t];
end cancel-flight;

end module
```

Each of the procedures pertinent to the FLIGHTSCHEDULE module encloses a call of an operation of the FILE module. Assuming all the called FILE operations are already implemented the FLIGHTSCHEDULE module may be executed and is then implemented as well.

3. Basic Characteristics of Modules

This "data driven" module definition is rather different from the more intuitive use of the term module in today's programming practice. It will, however, be shown throughout the rest of the paper that this notion seems to be adequate to overcome a great number of today's programming problems.

3.1 The Abstraction Principle

Among other reasons, the module concept has been defined this way to support the rather distinctive requirements of module users (application programmers) and module implementors (systems programmers).

It is the user's interest to employ the module concept to construct programs. Provided he knows the function of a previously defined module, only the interface, which contains all the information necessary to make proper use of the module, must be

visible to him. The details of the module implementation contained in its body would be an unnecessary burden and will remain hidden.

The module implementor on the other hand is responsible for an implementation in accordance to a given specification of the functions of the module. This partitioning of information according to a certain need to know, that is, the retention of the essential information for a certain purpose and the suppression of inessential details, is considered to be the key concept to master the complexity in information handling and is usually called the abstraction principle:

(An Aside in Specifications:
In both cases an implementation-independent specification of the functions of a module is necessary. The module user needs the specification to make sure the employed module has the intended functions and the module implementor uses the specification to implement this function in a complete and correct manner. Consequently, an implementation-independent specification of the functions of a module is an essential part of the module concept. It would, of course, be the ultimate goal to make the specification of the function of a module a part of the interface which can be checked automatically to ensure the proper use of the module. Since a concept for the computer representation and interpretation of function specifications does not exist at the moment, we will consider them as aside from the module.
End of Aside.)

The concept is not new in programming. All high-level languages, for example, provide means to declare and initialise data without forcing the programmer to assign data to specific memory locations. This feature helps reducing the commplexity of the programmers' task by hiding the memory allocation details in the languuage processing system.

Another very well-known abstraction mechanism supported in many high-level programming languages is the procedure concept. A procedure is designed to display its function to its users and to hide the implementation of this function. It is therefore considered as a suitable mechanism for a functional or procedural abstraction.

The data-oriented module concept defined above - as will be shown in the next section - is a generalisation of the known abstraction mechanisms. It is designed to display to its users the essential information on how to use a certain sort of data, and to hide the information on how those data are internally represented and manipulated. Hence, it provides a general data abstraction mechanism.

3.2 The Locality Principle

After the detailed definition of a module implementation in the preceding section, we are now ready to identify another basic priciple underlying the module concept. Obviously, operations and data are closely related to each other in the module concept. Each data object may be manipulated by only a certain predefined set of operations. Data not associated to a certain module but rather global to a number of modules do not exist. Thus, logical relationships between modules based on the shared use of global data cannot occur. The module concept also prohibits a module to refer to data declared in the body of another module, one module to branch into the body of another module, and one module to modify program statements within another module.

Hence, a module behaves like a self-contained entity which cannot cause non-local effects besides calls of other modules. To achieve this kind of locality is one of the goals in modular programming.

Modular programming is believed to have a number of advantages over more traditional program structuring concepts: (1) the prevention of certain types of structural relationships will force programmers to design programs of drastically reduced structural complexity. (2) Since the complexity of the environment in which a module will be used may be ignored by the implementor of that module it may also reduce the complexity of the programming task. (3) Because changes which have to be made in the implementation of a module will not affect any other part of a system, the concept will enhance system maintenance, adaptability and portability. (4) With the module concept the verification of the correctness of programs - the ultimate goal in program development - may be drastically simplified. Obviously, it is much simpler to show that the invariant properties of data will be preserved if the data is manipulated by its associated operations only, and not by other parts of a program. The verification may then be performed for each of the associated operations and not for all - usually unpredictable - uses of the data.

3.3 Protection of Data

In a modular software system, each operation may only be applied to a certain type of data object. The operations are tailored to comply with the characteristics of the data they manipulate, for example, it is common practice to manipulate integer data by a set of tailored arithmetic operations. Current high-level-languages compilers enfoirce this restricted application of operations defined in the language by means of a type checking capability for the built-in types of data. Since all the legal operations on a certain

type of data are predefined in a module, the correct use of these operations may be enforced by a similar type checking mechanism.

This approach is in contrast to the current practice in systems programming. Universal operations like 'delete', 'insert' or 'update' may be applied to data of any kind. In order to preserve the data's characteristics it is usually necessary to implement access control and protection mechanisms.

The two aforementioned approaches to preserve data characteristics in high-level programming languages and in systems programming are based on two fundamentally different philosophies: (1) Because of the awareness of the human limitations, the first approach follows the rule: anything not explicitly allowed is forbidden; (2) to guarantee the designer's freedom and flexibility, the second approach follows the rule: anything not explicitly forbidden is allowed.

After a long period of freedom and flexibility in the design of systems it now seems to be clear that a discipline is essential for the enhancement of software. The module concept and an associated type checking mechanism seem to be the natural means to avoid not intended manipulations of data.

3.4 Extensibility

A module is characterised by a mini-language: the abstract data and abstract operations of that module. A mini-language is implemented in terms of another mini-language provided by the module of representing type, for example, the mini-language

```
L1: FLIGHTSCHEDULE  (create-schedule,
                     search-flight,
                     schedule-flight,
                     cancel-flight)
```

is implemented in terms of the mini-language:

```
L2: FILE   (create-file,
            search-record,
            insert-record,
            delete-record).
```

More generally, an abstract operration may be implemented by a number of representing type operations. Each abstract operation may then be considered as an identifier for a macro operation on data objects of representing types.

This is in fact an extension capability similar to the one found in extensible languages. With the repeated definition of new modules implemented in terms of modules of representing type, which in turn may be implemented by other modules, one may define arbitrary high-level (mini) languages to suit particular users.

4. Module Interconnections

A discipline for the design and implementation of small programs (programming in the small) has been defined with the module concept. A similar discipline for the design and implementatiion of large software systems (programming in the large) must then provide rules for the interconnection of modules.

Since one module may employ other modules, one module interconnection mechanism - the nesting of modules - is already built-in in the module definition. With this structuring mechanism the overall structure of a software system may be organised in a hierarchic fashion. One may assign modules to levels in a hierarchy according to the following rules:

(1) Level 0 contains the set of modules which employ no other modules,

(2) Level i contains the set of modules which employ modules on level i-1.

An acyclic graph structure representing this interconnection of modules, so called D-graphs, has been introduced in (Web 76). See Figure 1.

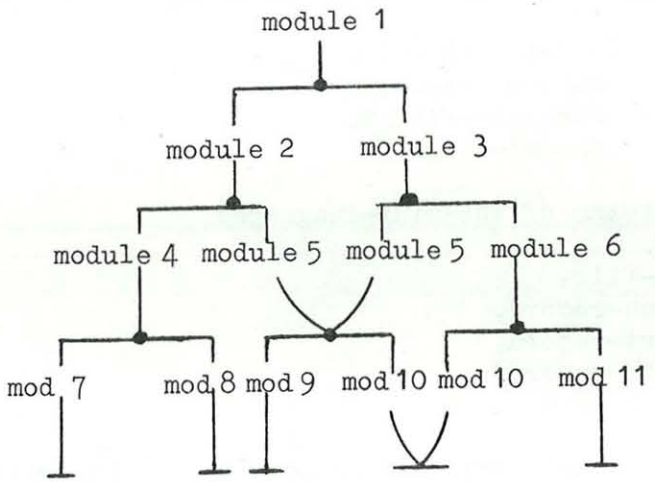


Figure 1 D-Graph

Modules are represented as nodes in the graph. The arcs represent the relationships between modules and are supposed to point downwards in the figure above. Arcs consist of a horizontal and vertical part for representation convenience only.

The hierarchic relationship between the modules may be characterised as a "uses" hierarchy because the services of one module may be used in another module (Par 74). The module interconnection will be kept simple because neither the used nor the using modules impose any restrictions on each other. They all remain self-contained system components which function the same way in all environments.

This hierarchic organisation of large software systems is accepted to provide means to keep the complexity of the system manageable and the function of the system understandable. (It may be important to note here that the hierarchic organisation of software systems does not predetermine the way they are designed: top-down, bottom-up, or in a more iterative fashion.) In order to keep the overall structure a simple hierarchy, other interconnection mechanisms - especially those neglecting the locality principle as explained above - are prohibited.

The grammar introduced in the previous sections of this paper is in fact the description of a module definition and module interconnection language. The language is therefore suitable for the programming-in-the-small and programming-in-the-large. It imposes a discipline on the programming task and supports the design of simply structured software systems. The language is hoped to be appropriate to serve as a general data base system design and programming language.

5. Object Creation and Manipulations

Modules have been defined to be invoked for the creation and manipulation of data objects. In the module concept, a data object will be brought into existence through the execution of a create operation of one particular module. The data object is then considered to be pertinent to that module. Only those operations defined in the module will ever be performed on the object. The object is said to be of the particular type defined by that module.

Because of the possible hierarchic compositions of modules, the create operation of one module may be designed to "use" the create operations of other modules to create and combine component objects. With this feature the module concept allows us to create, store and reference data on different levels of composition. At the same time, the concept automatically enforces the composition of objects out of components of the correct types as defined in the type module. (This may not seem very important for the standard compositions of data, like fixed format records in files, or files in blocks, but it is valuable for the composition of arbitrary user-defined data types).

Because of the hierarchic composition of modules, objects will be created which exhibit exactly the same hierarchic structure as the module composition graph.

An object composition may then be depicted as follows:

Nonterminal nodes represent composed objects, terminal nodes represent primitive indivisible objects. The labels on arcs denote objects identifiers and the identifiers of the type they belong to. (Figure 2.)

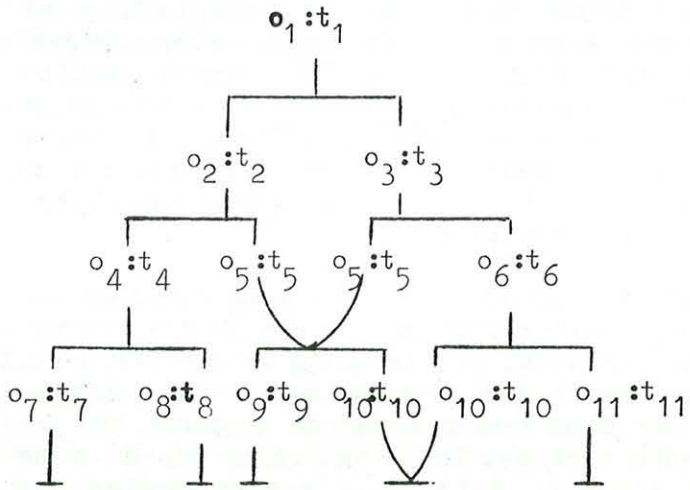


Figure 2

The execution of the create operation of module m_1 results in calls of create operations on subsequent modules $m_2 \ldots m_{11}$. The created object o_1 is of type t_1 and is composed of other objects $o_2 \ldots o_{11}$. Note, module m_5 is referenced in m_2 and m_3. Object o_5 will be created during the first call of the create operation of m_5.

A second call of this operation results in the creation of a second reference to the already created object o_5.

After its creation an object may be changed by insertions of component objects, deletions of component objects and updates of component objects. Figure 3 illustrates this when there are three consecutive operations.

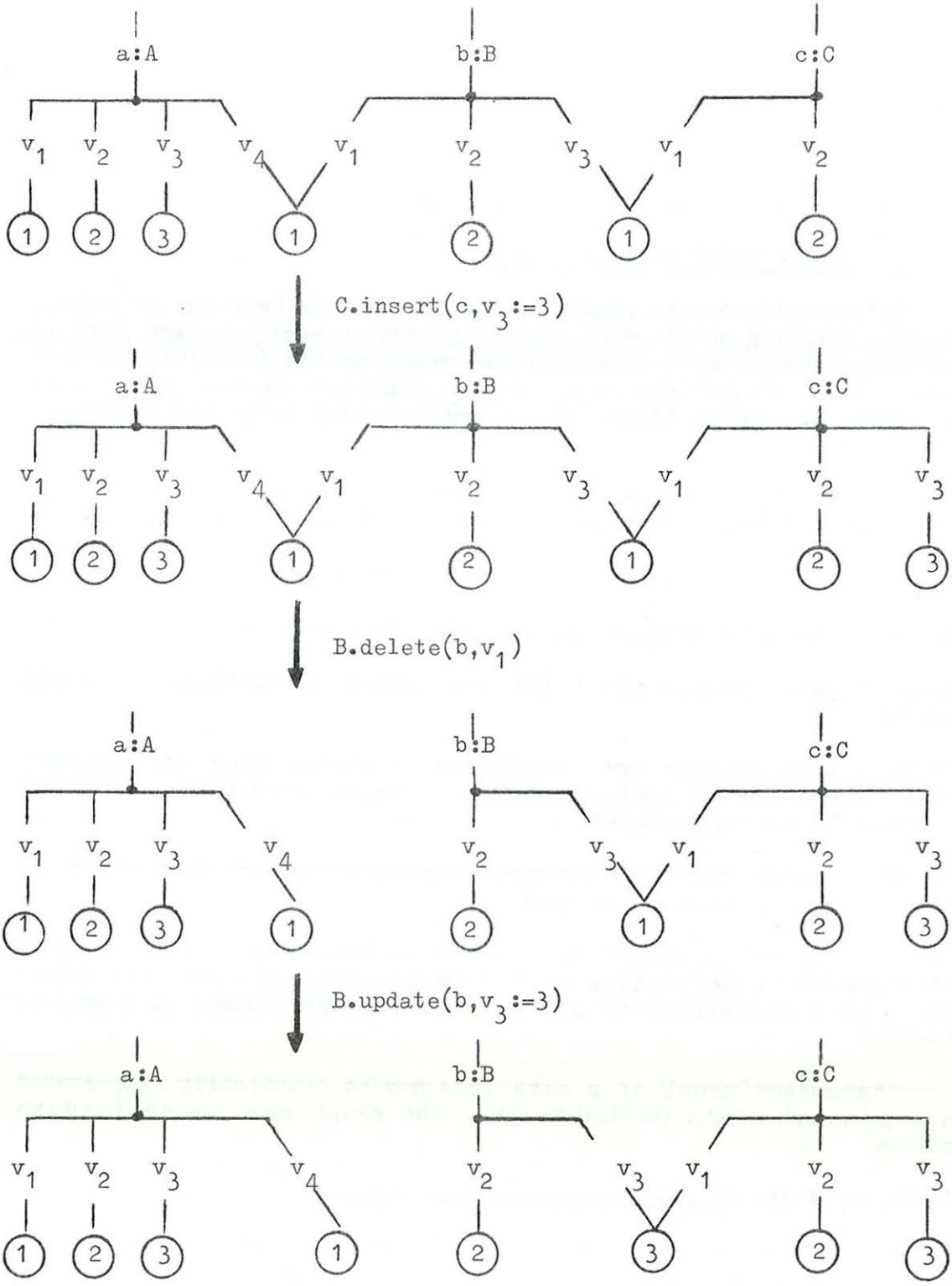


Figure 3

The insert operation leads to the insertion of a link and a node in the data object graph; a deletion results in a deletion of a link (or to the deletion of a link in a node in case the node is only referenced once); an update leads to a change of the value associated with a node.

6. The Module Oriented View of Data Base Systems

6.1 The Object Model of a Data Base

In accordance with common terminology a data base may be defined as the collection of all data objects stored on some storage devices and administered by a data base management system (DAT 75). The data base will be brought into existence through the initialisation and execution of transactions of the administering data base management system.

Within the framework of the module concept one may then define the object model of a data base in the following way (WEB 76:)

(1) A data base is a time-varying set of data base objects.

(2) Data base objects may encompass other data base objects.

(3) Data base objects may belong to a number of different enclosing objects.

(4) Data base objects are identified by names. They are uniquely identified within one enclosing object. They may be differently named in different enclosing objects.

(5) Data base objects are characterised by a type and each object is characterised by exactly one type.

(6) The type of an object is defined in the associated type module which provides a definition of the composition of object of other type and a definition of all permissible operations on composed objects.

The object model of a data base may be graphically represented as a D_o-graph (Data Object Graph). The graph may be defined as follows:

(1) Nodes of the D_o-graph represent data object.

(2) The directed arcs represent an "is part" relationship between objects. If an object i is component of an object j, a directed arc is drawn from j to i.

(3) Labels on arcs identify component objects and the type of component objects.

Clearly, the "is part" relationship between data base objects defines an acyclic graph because the whole (for example, a file) includes parts (for example, records) but a part never includes the whole. All non-root nodes may be in an "is part" relationship with a number of nodes. One may call them "shared" among a number of higher level nodes. A sample graph may then be depicted as follows: Arcs are again supposed to point downward. Nonterminal nodes represent composed objects, terminal nodes represent primitive indivisible objects. The labels on arcs denote the objects' identifier and the identifier of the type they belong to (Figure 4).

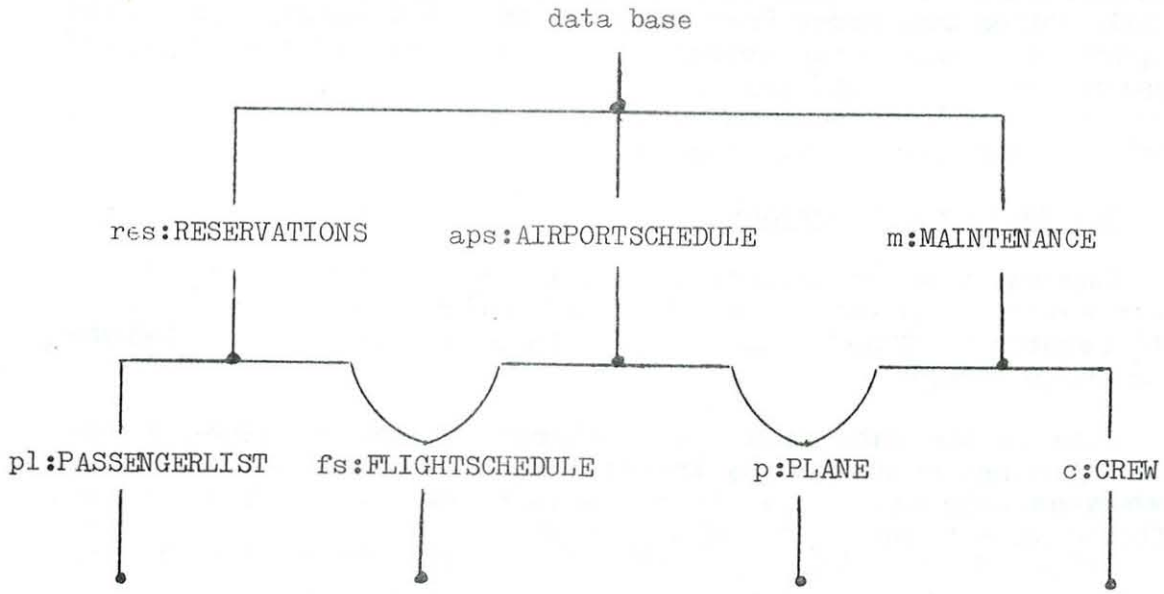


Figure 4

The graph depicts a simple data base with only a few objects representing some information about flights. The data base contains information on reservations and encompass information on passengers who have made reservations and on flights and their scheduling. The component of the data base termed airportschedule contains information on flights and their scheduling, and on planes which are allocated to those flights. The component termed maintenance contains the information on planes and on maintenance crews allocated to maintain those planes.

The possible changes of a data base by insertions, deletions and updates may be represented now in terms of modifications of the D_o-graph by insertions or removals of nodes or links and by replacements of nodes.

The object model conforms with the commonly accepted definition of a data base as quoted before, although the object model definition does not refer to the data base management system for the determination of the membership of a data object to the data base. For the object model, each module may be considered as the administering (mini-) data base management system for a particular type of data object. The entire data base management system may be thought of as the set of all defined and implemented modules. (A more elaborate definition of such a modular data base management system will follow later.)

The D_o-graph model is different from existing data base models in the following sense. Existing data base concepts support the composition of data according to one particular structuring model usually called data model (for example, the DBTG concept (DBTG 71) supports the owner-coupled-set data model, the relational concept supports the relational data model, etc.). The D_o-graph model of a data base supports the representation of differently structured data within the same structuring framework.

6.2 The Conceptual Description of a Modular Data Base

Data bases are repositories of all the data of interest in an organisation. They contain the information the organisation needs for its operations. Thus, data in the data base have a perceivable information content.

Data in the data base are of course subject to change. A data base object may be changed by insertions, deletions and updates of component objects. It is therefore important to distinguish two different aspects of its information content: the extension and the intension of the data base. The term extension refers to the instantaneous and time dependent aspects of the information content (for example, all the tuples in a relation at a certain point in time). The term intension refers to the time invariant aspects of the information content, (for example, the set of all permissible values an object can take). Data base objects may then be manipulated (that is, the extension may be changed) according to their time invariant properties (that is, according to their intension).

The D_o-graph model introduced above is clearly a representation for the data base's extension. To represent the data base's extension at different times the D_o-graph was changed through the insertion or removal of nodes and node connections. In order to manipulate the data base correctly, both its extension and its intension must be represented within the data base. The data base's intension is represented in a so-called conceptual description. Conceptual description of data bases are usually aimed to provide:

(1) a high-level user-oriented description of the data base for its users;
(2) support for a correct interpretation of the data base's information content;
(3) means for the representation of restriction on the use of data in different applications, etc. All these aspects will not be considered here. It is the sole aim of this description to refer to the support it provides for the correct manipulation of the data base. Elaborate proposals for conceptual description may be found in the literature on data dictionaries or conceptual schemata (ANS 75, BCS 77, NIJ 76, VLDB 75, VLDB 76).

The time invariant aspects of the information of interest to an organisation may be modelled in terms of the following concepts:

(1) Entity types or short entities are all concrete and abstract things or events an organisation needs to know of for its operation. An entity is meant to denote a collection of instances with identical characteristics, for example the entity FLIGHTNUMBER denotes a variable set of instances

$F\#_1$, $F\#_2$... which have the identical characteristic to identify flights.

Instances have values, $F\#_i := f\#$, or sets of values

$F\#_1 := [f\#_1, f\#_2 \ldots f\#_n]$ e.g., $F\#_i = [100, 200, 300]$.

Instances and values are subject to change. They are part ofthe extensional aspects of information.

(2) Conceptual relationships are associations between the entities of interest in an organisation, for example a conceptual relationship between the FLIGHTNUMBER and DEST NATION entitles refer to the fact that each flight identified by a flight number has a destination.

Some problems arise for the representation of the time invariant aspects of information in a communicable form in a suitable language. The representation of information in a language is in fact an assignment of labels, that is, words, to entities and relationships. Representations in a data base are then encoded representations of labels in computer store. In a simple representation each label is assigned to one entity or relationship and each entity or relationship gets only one label, thus, providing means for a unique representation and identification.

Words in a natural language however, usually do not identify things uniquely. They rather denote collections of things which are identical with respect to some properties and distinct with respect to others. For example, the word FLIGHTSCHEDULE denotes things which

are identical because they all identify flights, destinations and starting times. At the same time the denoted things may be distinct because some of them identify regular flights, and others, night coach flights. Due to the different properties of interest in different situations the very same word refers to different sets of values the things can assume in the different contexts, for example, if the FLIGHTSCHEDULE label is intended to denote flightschedules for regular flights all starting times must be within the time span 5am to 10pm; if it is intended to denote night coach flights, all starting times must be within the time span 10pm to 5am.

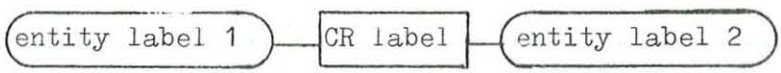
For a proper use of words for the representation of entities and relationships, the information about the intended set of legal values and their instances must be represented as well. It is common practice to represent this information in terms of constraints associated with entities and relationships, for example, the set of legal values of instances of the FLIGHTNUMBER entity may be defined by the constraint $0<f\#<100$. The computer representation of entities and relationships may then be based on the use of

(1) labels which denote entities and relationships, and

(2) constraints associated with entities and relationships.

As for the representation of the data base's extension a representational schema is usually defined for the representation of its intension.

It is common practice to depict entities and relationships by the following kind of graphs:



This building block may be used to construct entity-relationship-nets of arbitrary shape and complexity. (For the sake of simplicity, the terms entity and relationship are used now to denote both real things and their representation by labels as well.) Nets of this kind are then suitable to depict the information of interest in an organisation.

So far this schema does not permit the representation of constraints. The constraints associated with entities and relationships will be represented as conditions which will be checked whenever modifications of the values of instances take place, for example, for each execution of the operation add-number on an instance F of entity FLIGHTNUMBER the following check will be performed

```
if 0<f#<100
then FLIGHTNUMBER.add-number (F#,f#)
else return FALSE
```

to make sure that new flightnumbers f# will only be added if they are taken from the set of natural numbers between 0 and 100.

The representation of entities, conceptual relationships, and their properties may therefore be given in terms of entity-relationship-nets and constraints.

It remains to be shown now that the module concept suffices for the representation of a data base's intension in terms of entity-relationship-nets and of constraints.

6.2.1 The Representation of Entities

As one may conclude from the previous discussion, data types and entities (labels denoting entities) seem to be very similar tools for the representation of information: they both denote (real or abstract) things with common properties. Because of this similarity it seems to be natural to use modules as a means for the definition and implementation of entiities in the same way they were used to define and implement types of data.

(1) Module interfaces denote entities and the set of permissible operations on instances of entities.

(2) The implementation of entities and of operations on their instances is defined in module bodies.

(3) Modules may be invoked to create and delete instances and to modify the values associated to them.

Although there exist some obvious similarities between entities and abstract data types a couple of important differences must be kept in mind:

(1) Entities have not been defined in conjunction with the set of permissible operations on their instances.

This feature of a module to define data and operations together, however, seems to be quite adequate for the representation of the invariant properties of entities. As one may recall, those properties have been represented in terms of constraints on the values of instances and have been enforced during value modifications. Because all possible value modifications are defined in a module, constraints on the possible values may be defined within the framework of a module and enforced in module operation executions in a straightforward way: Operation execution conditions may be associated with all value changing operations of a module. The operation will be

executed only if the conditions set forth are satisfied. These conditions may be represented in a module within the procedures which are defined to implement those operations, for example,

```
module FLIGHTNUMBER (add-number, delete-number)

begin

rep as SET of NAT

proc add-number [F#, f#] --> F#;
if 0<f#<100 TRUE;
then SET.add-element [S, f#];
else return FALSE;
end add-number;
.
.
.
end module
```

The add-element operation will be executed only if the operation execution condition (0<f#<100) is satisfied.

For the representation of entities and constraints on the values of their instances a module definition may then take the following general format:

```
module ENTITY IDENTIFIER (oper 1, oper 2, ... oper n)
begin

entity representation

proc oper 1
operation execution condition 1
operation execution condition 2
      .
      .
statement 1
statement 2
      .
      .
      .
end oper 1
      .
      .
      .
end module
```

Thus, the definition and implementation of an entity by a module results in a rather implicit representation of constraints on the values of its instances in terms of constraint preserving operations. But this feature supports at the same time, the representation of the dynamic properties of the data as an essential part of the data basse intension.

(2) The second difference between entities and abstract data types results from another characteristic of modules. Modules have been designed to employ other modules for the implementation of objects and operations of abstract type in terms of objects and operations of representing type. The mapping between objects of abstract type and of objects of representing type implies always a classification: All legal values of objects of abstract type will be mapped onto a subset of legal values of objects of the representing type. This classification of the set of legal values of representing type takes place because of the likeness of the selected values with respect to a certain property, for example, an abstract object of type NIGHTCOACHFLIGHTSCHEDULE may be represented in terms of objects of type FLIGHTSCHEDULE; the set of legal values of objects of type NIGHTCOACHFLIGHTSHEDULE is a subset of the legal values of type FLIGHTSCHEDULE; the legal values of objects NIGHTCOACHFLIGHTSCHEDULE are alike with respect to a certain starting time. This property is not common to all values of objects of the FLIGHTSCHEDULE type. Thus, objects of abstract types have properties which will not be inherited by objects of representing type.

If modules are used to define and implement entities, the above capability may be exploited to define classifications of instances according to certain properties. One may call this classification abstractive, since it has been made on the basis of some distinctive properties of interest, while all other properties have been neglected. Thus, those abstractive classifications may be introduced for the definition of new entities with new common characteristics. At the same time classifications delimit the scope of attention to some properties of the entities, and ignore all others. One may refer to those classifications as schema abstraction which are equivalent to the data abstractions introduced in Section 2.2 Based on this abstractive classification, information may be represented on arbitrary levels of detail or abstraction. This supports the representation of abstraction as an essential part of the data base's intention.

6.2.2 The Representation of Relationships

At first glance the module concept does not seem to be very useful for the representation of relationships. Some considerations about the nature of entities and relationships, however, may offer some help. The distinction between entities and relationships is certainly not absolute: things may be considered as entities in one context and as relationships in another, for example, a

FLIGHTSCHEDULE may certainly be considered as an entity. It materialises, however at the same time a relationship between FLIGHTNUMBERS, DESTINATIONS, and STARTTIMES. If it is in general true that the distinction between entities and relationships is just a matter of the context and not a matter of their representation, the question may be asked whether there is any real need to represent entities and relationships in a different manner. It seems to us that no principal difficulties exist to treat relationships in the same way as entities. A relationship and the entities connected through this relationship may be considered as a composed entity in its own right. For the representation of relationships by modules, a capability is needed for the representation of compositions of entities and for the definition of the characteristics of the composed entity which reflect the relationship between the components' entities.

The module concept has been defined to offer exactly this capability: arbitrary module interconnections may be defined in the module of representing type and the characteristics of the module reflect the nature of a relationship between the component modules.

The following example illustrates such a composition. Suppose the data base contains the following two entities.

```
FLIGHTSCHEDULE (F#, DEST, ST-T)        (create-schedule,
                                        search-flight,
                                        schedule-flight,
                                        cancel-flight)

PLANE (P#,TYPE,N°S)          (create-plane,
                              search-plane,
                              reserve-plane,
                              cancel-planereservation)
```

The relationship which must hold between these two entities is of the following nature:

The scheduling of a flight requires the corresponding allocation of a plane to this flight. The data base contains for each entry in the FLIGHTSCHEDULE instance a corresponding entry in the PLANE instance.

We therefore define a new entity whose specification reflects this relationship and which encloses the entities FLIGHTSCHEDULE and PLANE as components. This new entity will be called AIRPORTSCHEDULE:

```
AIRPORTSCHEDULE (FLIGHTSCHEDULE, PLANE) (create-apschedule,
                                         search-entry,
                                         add-entry,
                                         delete-entry)
```

The relationship will be preserved if the operations on AIRPORTSCHEDULE instances are designed in such a way that the component instances of FLIGHTSCHEDULE and PLANE will be manipulated correspondingly, for example,

```
module AIRPORTSCHEDULE (..., add-entry,...)

begin

rep as INTERCONNECTION of (FLIGHTSCHEDULE, PLANE)
•
•
•
oper add-entry [aps, f#, dest, st-t, p#, type, n°s] → aps:AIRPORTSCHEDULE;
FLIGHTSCHEDULE.add-flight [fs, f#, dest, st-t];
PLANE.reserve-plane [p, p#, type, n°s];
end add-entry;
•
•
•
end module
```

The add-entry operation will be performed by performing both the add-flight and the reserve-plane operations.

6.2.3 Conclusion

Both entities and relationships were represented as modules, their instances as objects of the respective type. The different values instances may assume were represented as different instantiations of changeable objects. Modules provide representation capabilities which are shown to be general enough for the conceptual description of a data base. Because of the built-in module interconnection mechanism in modules, the representational schema consists then of a collection of modules which are hierarchically structured in a graph which we will call a D_c-graph (Figure 5). Nodes now represent entities, links point to all nodes which represent related component entities. The D_c-graph may be considered as an entity/relationship composition graph. A D_c-graph is in fact a strictly modular conceptual description of the intension of the data base. It incorporates a specification of all possible data base operations, all kinds of integrity constraints, and supports a description on different levels of abstraction.

A number of consequences follow from this rather uncommon concept, which incorporates the representation of declarative and procedural knowledge, and of schema abstractions. An elaborate comment on it must be left aside here.

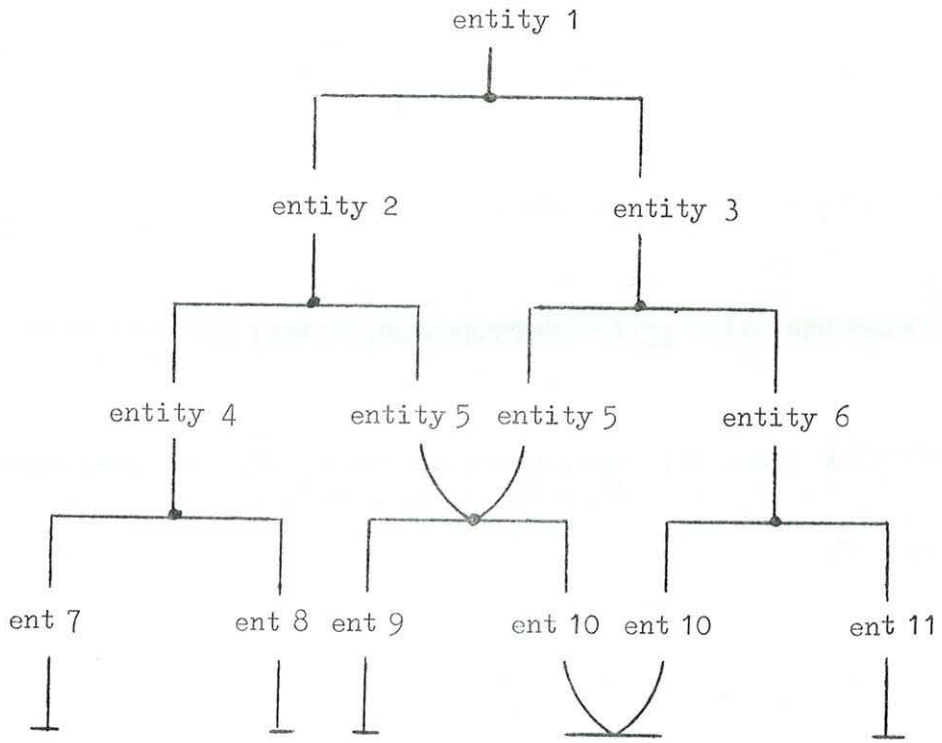


Figure 5 D_c-Graph

6.3 The Representation of Data Base Views

The term view is used here to denote a conceptual description (frequently called external description) of the data base which suits one user or one group of users. It is of particular interest to represent the data incorporated in each view in a suitable structural form and to provide an appropriate data base interrogation and manipulation language to the users of each view. For that reason it seems to be natural to define view modules which exhibit the desired characteristics to their users and call on the services of other underlying modules.

Different users may be interested in different but overlapping parts of the data base. Different views may then incorporate different subsets of data base entities may have entities in common, and may see the entities they share involved in different relationships (Figure 6).

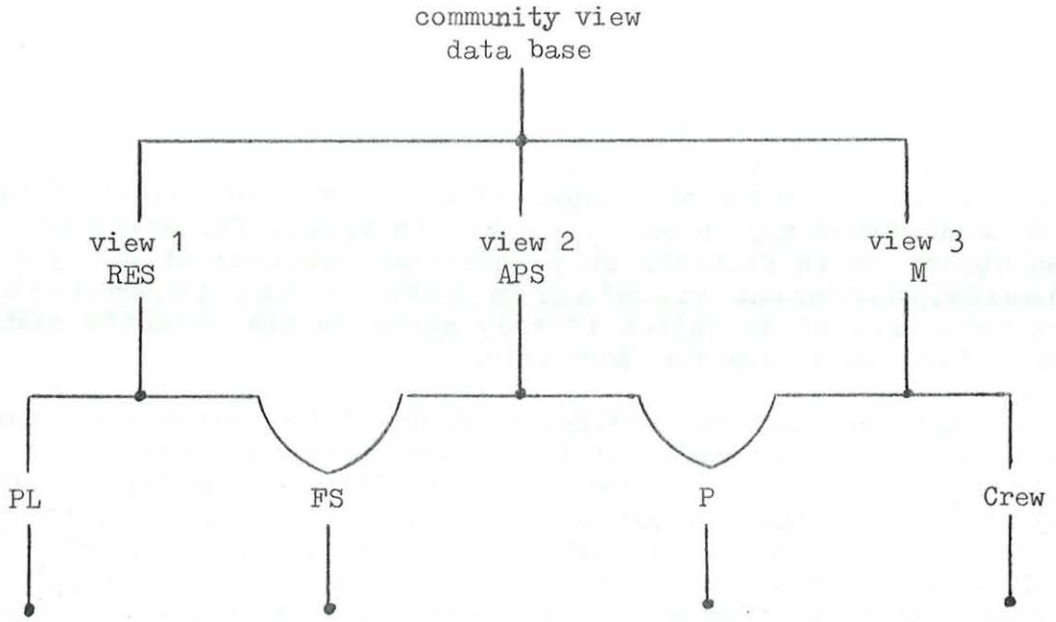


Figure 6

view 1: the flight reservation view sees a FLIGHTSCHEDULE entity in a relationship with a PASSENGERLIST entity.

view 2: the flight scheduling view sees a FLIGHTSCHEDULE entity in a relationship with the PLANE entity.

view 3: the maintenance view sees the PLANE entity in a relationship with the maintenance CREW entity.

As a consequence the same instances of entities and relationships - the data base objects - may be interrogated and - more importantly - manipulated via different views. One may call the objects seen through different user views as shared objects.

If we allow the manipulation of shared data base objects via different user views, we have to make sure that different view agree on the intensional characteristics of the shared objects. Otherwise the integrity of the data base may suffer. It is shown in (WEB 76) that modules are suitable to define these sharing properties of data base objects. A brief description of the concept follows in the next paragraphs.

6.3.1 The State Space and Substate Spaces of Data Base Objects

In order to explain the notion of agreement about the intension of data base objects, it is convenient to introduce the concept of a statespace and of substate spaces for changeable data base objects. An object has a particular set of components at a certain time. It may be changed by changes of the set of its components. Consequently, it may be considered to be in a certain state at a certain time and its state will change in object manipulations. The set of all legal states may be called the state space of an object. Each subset of the set of legal states may be called a substate space. The state space of an object is in fact the only invariant property of an object. Obviously, different views agree then on the intentional characteristics of an object if they agree on the object's state space or focus on a substate space only.

As explained before, the description of the representation of objects and the description of the associated manipulations in a module embodies in fact an implicit definition of the space of all legal states for this object. A user who inserts a new object defines, with the declaration of its type, its state space, and will be called the owner of the object.Different views may share an object in a non-conflicting manner, if they resepct the defined state space. This however, is guaranteed in the module concept because all changes of an object's state initiated from any view will be executed by (local) operations of the associated module and this module has been designed to guarantee the desired intensional properties.

It is however also legal for a view to focus on a substate space of a data base object only. This may be achieved by defining a view modula which enforces an appropriate restriction. Given this, views may be distinct with respect to the set of entities and relationships they encompass, as explained in the previous paragraph, and with respect to the set of instances of common entities and relationships. Although different views may be distinct with respect to the set of instances at a certain time (that is, with respect to their extensional properties) they must agree on the state space of common entities and relationships (that is, on their intentional properties).

6.3.2 The Image of Data Base Objects

Views have been defined to exist continuously. They will not be created and maintained temporarily but are a user's permanent window to the data base. In order to represent different permanent windows to the data base we introduce the notion "image of an object".

If the substate spaces of an object are different in different views, components of this object may exist at a certain time which may legally belong to one view (that is, to one substate space) but not to another. One may say the different views have different images

of an object, that is, they see different instantiations of an object. One may represent images of a data base object within the D_o-graph framework as in Figure 7.

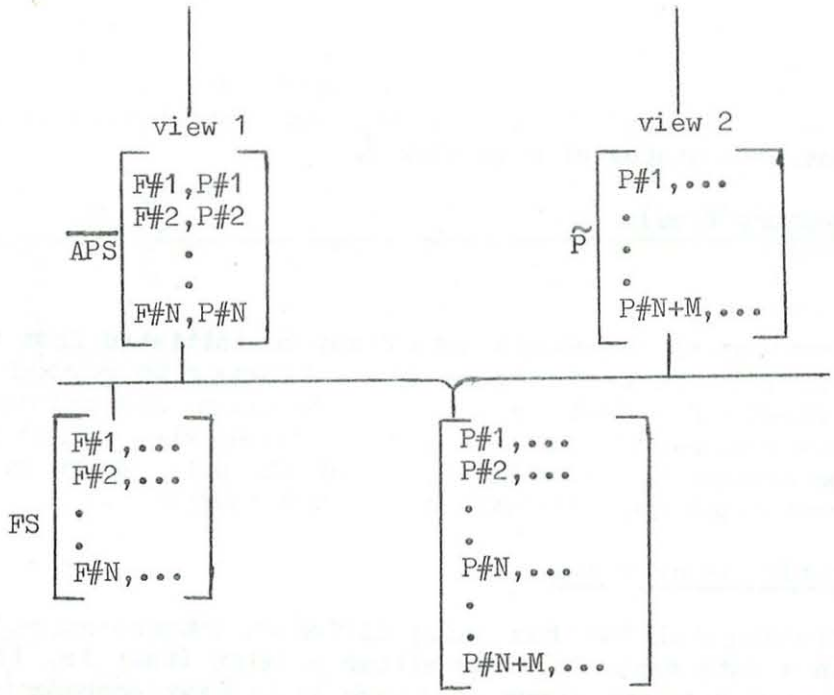


Figure 7

View 2 sees an object P which embodies an instantiation of P which represents all available planes. View 1 contains instantiations which represent all allocated planes. The different images of the object are defined through the selector relations APS (F#,P#) and P̃(P#,...). They both identify different subsets of the set of components of P and make them visible in different views.

The two different views may be created and maintained because of the following definition of the twoo view modules:

Assume the type of P is defined by a module

```
PLANE (P#,TYPE, N^OS)   (create-plane,
                         search-plane,
                         insert-plane,
                         delete-plane)
```

Different instantiations of an object P may be created and maintained in the two different views if different subsets of the set of legal operations may be called from the two different views.

VIEW 1 (The Schedulers' View)

search-plane
Semantics: None of the changing operations on P can be initiated from view 1. This is to express the fact that view 1 is only authorised to initiate the allocation of already recorded planes. The set of components of P in view 1 will then be at any time a subset of the set of components seen in the subsequently defined view 2. Consequently, the set of possible states of P in view 1 is a subset of the set of possible states of P in view 2.

VIEW 2 (The Inventory View)

search-plane;
insert-plane;
Semantics: Insertions of components into P may be initiated from view 2 without any restriction to indicate its authority to record the existence of planes for further use. The deletion and update of components, however, must be reserved to another view which has control over both views and is able to prevent the deletion or update of already allocated planes. (One may call this a superview.)

6.3.4 Communication Among Views

One may distinguish two basically different interrelationships between views in a data base. They may either coexist (that is, there is no interference between views although they have components in common) or they may cooperate (that is, a strictly controlled communication may occur by making changes of a shared object visible to all views which see the object). Both concepts will be briefly discussed in the following paragraphs.

Views may be defined to coexist because the D_0-graph concept allows one to maintain and manipulate just images of data base objects. It therefore provides means to manipulate the data base via one view without affecting any other view. Views coexist in the data base, in general, if the following operations are performed:

(1) insert (component of object). Its effect would be a modification of the image of the object and of its set of components. The component inserted via one view, however, would not be visible in other views, since the object's image seen in the other views has not been changed.

(2) delete (component of object). Its effect may be just a change of the image of the object without affecting the state.

Views cooperate if update operations on shared objects may be performed. Update operations preserve all the images and change the state of an object. The resulting changes are therefore visible in all views which reference the object. In order to keep the data base in a

consistent state, a general policy for the performance of updates via different views must be established.

(1) If a non-owner view seeks the sharing of an object, it must agree to all possible changes of this object made by the owner of the object.

(2) An owner view may grant the opportunity to change an object it owns to all or a selected number of other views. A non-owner may then accept this opportunity.

Based on a more detailed explanation in (WEB 76) one can conclude that insertions and deletions may be performed not causing conflicts among user views. Updates may be performed to enable the communication between views if proper rules for this communication are set forth.

6.4 The Module Oriented View of Data Base Management Systems

Some existing and most proposed data base management systems are structured in a layered fashion. The layers correspond to the different modes for the representation of information in the data base (usually called logical and physical in a two-level architecture, or in a three-level architecture, external, conceptual and internal). The layers are employed successively for data base interrogations and manipulations (Figure 8).

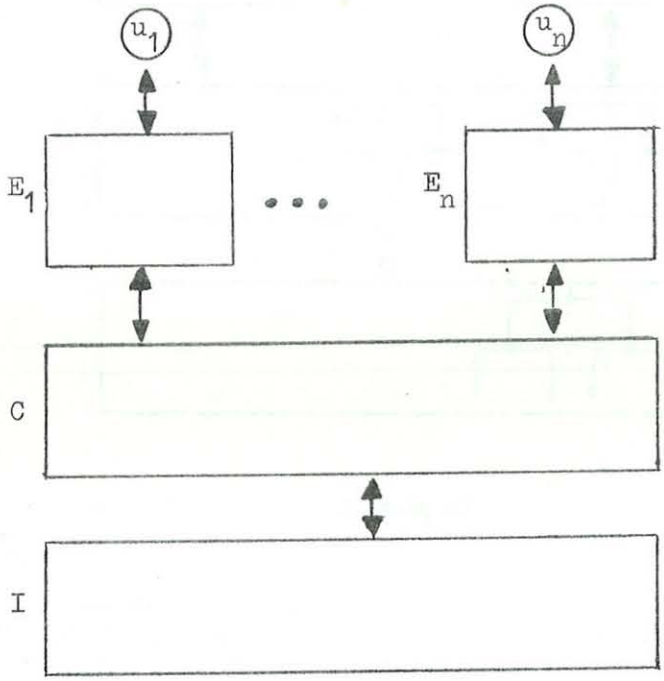


Figure 8

A number of external "machines" will be designed to process different users' interaction languages. The external "machines" will be implemented in terms of a conceptual "machine" which in turn will be implemented by an internal "machine". Usually different types of data (that is, differently structured or differently manipulatable data) will be processed by each of these "machines".

The module concept provides means for the implementation of those "machines" and conforms therefore with this principle data base management system architecture. It offers, however, some further system structuring capabilities which are appropriate to define arbitrary structural refinements for each machine. Refinements may be either functional (that is, the gross function of a machine is decomposed into sub-functions of component machines) or data driven (that is, a machine which processes a number of types of data will be decomposed in a number of component machines each processing a subset of the types of data). Such an architecture may then be depicted as in Figure 9.

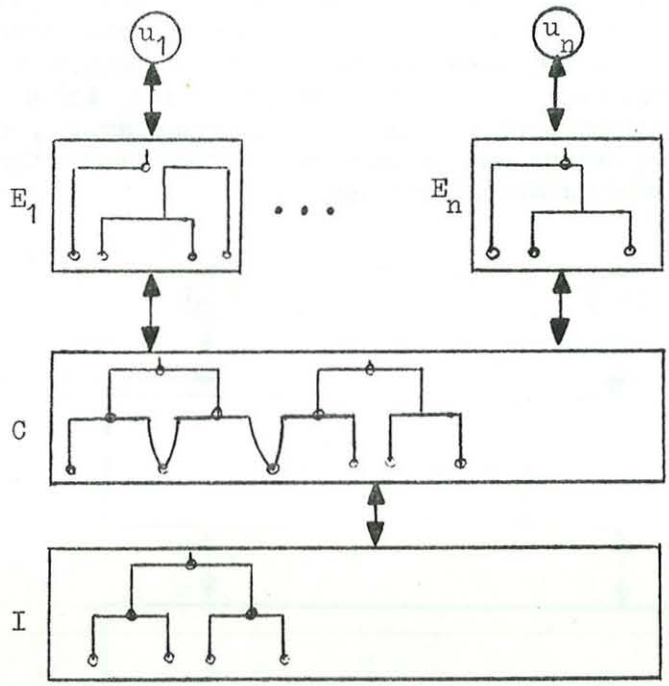


Figure 9

This architecture may be developed in a coherent way in an overall design process. It may be also the result of an extension of already existing data management systems (which will then be employed by the modularily designed levels above).

7. Other Related Work

The module concept has been applied to a number of other problems in data base management which will not be described in detail here.

(1) A software design strategy based on the module notion has been applied to develop a methodology for the design of a family of data base systems. The strategy supports a top-down design of modular data base systems (Yeh 77, Yeh 78).

(2) A few attempts have been reported to develop a methodology for the formal specification and verification of data base systems. The methodology is based on concepts for the algebraic specification of modules (PAO 77, EKW 78, BR 78).

(3) The benefits which can possibly be gained from an application of the concept in designing distributed data base systems have been described in (HEB 78).

(4) Last, but not least, the concept has been applied to model security and privacy enforcement mechanisms (MIN 76).

8. Conclusion

The module concept has been shown to be suitable to model the main features of data base systems. It should consequently contribute to the simplification of the data base system development and to the enhancement of data base systems. The use of modules as a descriptive tool does not imply any redefinition of accepted basic concepts in data base management. It offers however, in some cases, a more precise definition of the concepts.

Although an increasing number of people are doing work on the subject, experimental projects along this line have not yet been reported so far. The presentation was aimed at stimulating some further work on the application of the concept to database systems.

Acknowledgement

The author gratefully acknowledges the careful reading of an earlier draft of the paper by M. Brodie and H.-J. Kreowski, and many helpful comments by N. Roussopoulos, H. Ehrig, and K. Kreplin.

Discussion

There was some discussion about the module concept and whether it helps in specifying integrity constraints in a data base. Dr. King enquired from the point of view of data bases and their implementation in PL1 and asked what was new about the proposed methodology. In PL1, a procedure with multiple entry points can be used to obtain the effect of a module - so are there any new language features that are needed in order to program in the manner suggested? Dr. Weber replied that he is proposing a new philosophy where the notion of universal operations (insert, delete, update etc.) on a data base is absent. Rather, specific operations on objects are associated. This has the further advantage that the additional notion of integrity constraints is not needed since it is captured in the specific operations on objects. Dr. King disagreed with the last point and said that such constraints must be specified. Prof. Wassermann intervened and said that there can be two approaches to the specification of integrity constraints. One is to collect all such constraints on the data base together and then these constraints must somehow be checked; we are not sure as to how, which one and when, but we have some vague ideas about it. The second approach associated with the module concept is to associate constraints with modules such that the specified operations maintain the constraints. This considerably simplifies integrity checking.

Bibliography

(ANS 75) ANSI/X3/SPARC Study Group on Data Base Management Systems, Interim Report, ACM FDT Bulletin vol. 7, No. 2 (1975).

(BCS 77) The British Computer Society Data Dictionary, Systems Working Party Report, ACM SIGMOD RECORD, vol. 9, No. 4 (1977).

(BR 78) Brodie, M., Specification and Verification ofo Data Base Semantic Integrity, TR CSRG-91 1978, Univ. of Toronto.

(COD 70) Codd, E.F., A relational model of data for large shared data banks, Comm. ACM, vol. 13, No. 6 (June 1970).

(DAT 75) Date, C.J. "An Introduction to Data Base Systems", Addison Wesley (1975).

(DBTG 71) CODASYL Systems Committee, Report of the CODASYL data base task group, ACM (April 1971).

(EKW 78) Ehrig, H., Kreowski, H.J., Weber, H. "Algebraic Specification Schemes for Data Base Systems" Technical Report HMI-B266, Hahn-Meitner-Institut Berlin (1978).

(HEB 78) Hebalkar, P.G., "Application Specification for Distributed Data Base Systems" submitted for publication, 1978.

(LIS 74) Liskov, B., Zilles, S., "Programming with Abstract Data Types" ACM SIGPLAN Notices, vol. 9, No. 4 (April 1974).

(LIS 77) Liskov, B., "Abstraction Mechanisms in CLU," Comm. ACM, vol. 20, No. 8 (August 1977).

(MIN 76) Minsky, N., Intensional resolution of privacy protection in data base systems, Comm. ACM, vol. 23, No. 2 (April 1976).

(NIJ 76) Nijssen, G.M. (ed.) Modelling in Data Base Management Systems, North Holland (1976).

(PAO 77) Paolini, P., Pelagatti, G., "Formal definition of Mappings in s Data Base", ACM SIGMOD Conference Proceedings (1977).

(PAR 71) Parnas, D., "Information Distribution Aspects of Design Methodology", Information Processing 71, North Holland (1971).

(PAR 72) Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules", Comm. ACM, vol, 15, No. 12 (Dec. 1972).

(PAR 74) Parnas, D., "On a Buzzword': Hierarchical Structure", Information Processing 74, North Holland (1974).

(VLDB 75) Kerr, D., ed. Proceedings of the First International Confernece on Very Large Data Bases, available through ACM (1975).

(VLDB 76) Lockemann, P., Neuhold, E. Proceedings of the Second International Conference on Very Large Data Bases, North Holland (1976).

(VLDB 77) Proceedings of the Third International Conference on Very Large Data Bases, available through ACM (1977).

(WEB 76) Weber, H., "The D-Graph Model of Large Shared Data Bases: A Representation of Integrity Constraints and Views as Abstract Data Types", IBM Research Report RJ 1875 (Nov. 1976).

(WIR 77) Wirth, N., "Modula: A Language for Modular Multiprogramming", Software Practice and Experience, vol. 7 (1977).

(WUL 76) Wulf, W.A., LONDON, R.L., and SHAW, M., "An Introduction to the Construction and Verification of Alphard Programs", IEEE Transactions on Software Engineering, vol. SE-2, No. 4 (Dec. 1976).

(YEH 77) Yeh, R.T., Baker, J.W., "Toward a Design Methodology for DBMS: A Software Engineering Approach" in (BLDB 77)

(YEH 78) Yeh, R.T., Roussopoulos, N., Chang, P., "Data Base Design - An Approach and Some Issues" Technical Report SDBEG-4, University of Texas at Austin.