Specification, Verification, and Enforcement of Semantic Integrity Using Behavioral Abstraction

Nancy G. Leveson
Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 833-5233

and

Anthony I. Wasserman
Medical Information Science
University of California, San Francisco
San Francisco, CA 94143
(415) 666-2951

Technical Report 157

This work was supported in part by an IBM Dissertation Fellowship to Nancy Leveson at UCLA and by National Science Foundation grant MCS78-26287 to UCSF.

Abstract

This paper presents a method for the specification, verification, and enforcement of semantic integrity using behavioral abstraction. The specification contains both the abstract invariant of the abstract object (the static characteristics of the data) and the legal operatios on the objects as defined by pre and post conditions (the behavioral characteristics) which preserve this invariant. The integrity specifications can be verified by proving that the abstract operations preserve this invariant. A practical means for enforcing integrity leads naturally from this integrity specification technique.

Key Words and Phrases: semantic integrity, data abstraction, formal specification, program verification, information system, views, abstract data types, behavioral abstraction, database systems

CR categories: 4.0, 5.24, 4.33, 4.6

1. Introduction

Data integrity involves ensuring that the data in a data base is accurate. Currently there is no satisfactory method for ensuring that data integrity is not violated. Most methods which have been proposed are either incomplete or impractical in terms of overhead.

Data may become inaccurate due to hardware or system failure, illegal alteration or destruction of data (violations of security), improperly controlled concurrent access to shared data, and invalid alteration or destruction of data. *Semantic integrity* deals primarily with the latter, i.e., invalid operations on the data base. A data base is not just a collection of values, but should be a symbolic representation of the knowledge (a model) of some real world system. At every point in time, the contents of the data base represents some configuration of the application domain. The semantic integrity of data is violated when the data base no longer represents a legal configuration of the system it is intended to model.

2. Semantic Integrity

A real world system has an intrinsic logic, i.e., a set of rules that determine which of its configurations are valid. These rules or assertions are called integrity constraints. As an example, a constraint in a university data base might be that no student can be enrolled in two different classes that meet at the same time. It seems reasonable that to test integrity, it is necessary to have a well-documented description of these logical rules upon which the data base is being modelled. Most systems in current use, however, scatter this information in three places:

1) the conceptual schema presents some semantic integrity information on individual data items, e.g., the number of classes in which a student is enrolled is represented by an integer, a student id (key) uniquely identifies a student, a name is a string of characters.

2) Other semantic information is often embedded in the structures used in the data model upon which the system is built. For example, in the hierarchical model, semantic integrity information in the form of one-to-many relationships between data is expressed in the basic tree structure of the data base. In the network model, many-to-many relationships within the data base are expressed by appropriately constructing network structures. The basic structure of relations normally includes the specification of functional dependencies between data items. In an attempt to incorporate even more semantic integrity information in the data model, so-called higher data models have been devised. A proliferation of data models has resulted from this attempt, each purporting to incorporate more semantic information than the others. In general, however, the type of information which can be specified is severely limited by the model.

3) Finally, some semantic information may be incorporated into the procedures which implement operations on the data.

Thus most of the semantic integrity information which can be included is implicit, i.e., it is never explicitly stated in the conceptual schema. The disadvantages of such an approach to integrity are:

1) Because the conceptual schema and the data models do not enforce sufficient integrity, there is no guarantee that integrity cannot be compromised by some update to the data base.

2) Because the integrity information is scattered throughout the model and

the operations, there may be inconsistencies and redundancies which are very difficult to detect and locate. Also, modification of integrity information is difficult.

3) Semantic integrity information which is embedded in the structure of the data model is not easily modified, and a change in the real world system being modelled may necessitate a large amount of reprogramming.

4) It is difficult to optimize the integrity checking process.

Because of these disadvantages, a complete specification of the semantic integrity information in the conceptual schema seems appropriate. There are two major approaches to specifying semantic integrity: specifying the data base states which are permissible, called static integrity constraints, or, specifying the legal manipulations or changes allowed on the data base, called dynamic integrity constraints.

In a system using static integrity constraint specification, the user specifies the constraints and leaves it to the system to enforce them. Since updates may consist of several steps, it may actually be necessary for the data base to be temporarily in an invalid state. Specification of constraints on relational data bases has been investigated by Hammer and McLeod [5,9] and Brodie [2].

A formal specification of the static constraints, although necessary, is not sufficient in itself to guarantee integrity. There must be some way to enforce the constraints. This enforcement can be accomplished in three ways:

1) Make periodic checks of all of the stored data by means of some kind of "audit" program. The audit routines, however, are usually complex, low-level, and in order to improve performance, closely tied to the physical representation of the data in storage. Thus it becomes extremely difficult to modify the programs or to change the physical storage structure. Further, when an audit fails, it is difficult for an audit program to identify the exact error and the process which caused it, other processes can be exposed to the error before the audit occurs, and possibly enormous recovery problems can ensue when integrity errors are found.

2) After every update to the data base, check to make sure that no static integrity constraints have been violated. This is the approach of System R [1], a relational data base system developed by the IBM Research Division. For this approach to work, it must be possible to undo the effects of an illegal operation (some kind of backout). The transaction need not necessarily be completely backed out; System R allows the specification of special "integrity points" where the user specifies that integrity is to be checked. In the event of a subsequent integrity failure, the transaction can be backed out to the last integrity point. Further, with multi-step updates where several changes must be made to the data base in response to a single input transaction, intermediate states may occur in which some static integrity constraints are violated. Therefore, there must be some means of delaying the checking of integrity until after some specified series of updates. System R handles this problem by checking and enforcing integrity assertions at the end of each transaction unless the assertion is specified as "immediate." The integrity checking system proposed by Hammer and McLeod works similarly.

3) check every update to see if it would violate integrity and to see if the system should take some consequential action (e.g., automatically modify the data or signal an error). This approach prevents the execution of operations that violate constraints instead of allowing any operation to take place, checking afterwards, and then, if necessary, undoing the effect of any illegal operation.

This is the approach taken in Query-by-Example [18] and INGRES [13]. Both of these systems handle only relational data bases and are limited in the kinds of constraints that may be specified and checked. It is not clear whether all possible static integrity constraints can be handled in this fashion.

The biggest drawback to enforcing static integrity as described above is the problem of excessive overhead. Stonebraker [13] states that although simple assertions have a negligible cost in INGRES, others are considerably more costly in terms of complexity, overhead, and execution time. Present data base systems in the market place perform only limited types of integrity checking, if any, and the developmental systems mentioned above which have actually been implemented have performance problems. It may be that the overhead necessary to verify static integrity constraints at runtime is simply too excessive to be practical in most environments.

Other problems with systems using static constraints include: 1) most proposed systems are based on the relational model and focus primarily on relation constraints, thereby dealing only with a subset of semantic integrity problems, 2) they are inflexible with respect to when assertions are checked, and 3) the possible specification of the actions to be taken upon detection of an integrity violation are limited.

The alternative to static integrity constraints is the specification of dynamic integrity constraints, i.e., specification of the permitted manipulations on the data base. If the initial state of the data base satisfies the static integrity constraints and the only manipulations allowed on the data base have been shown to maintain integrity, then the present state of the data base must also satisfy the static integrity constraints.

One way of specifying dynamic integrity is to use abstract data types. The abstract data type has been a central concept in the use of behavioral abstraction in programming languages. Essentially, a data abstraction hides (abstracts from) the data representation of a data object and characterizes objects by operational (behavioral) attributes. Thus it is possible to distinguish between the "what" specification and the "how" implementation of a data type. Generally, all information about the type is hidden from the user except the name of the type itself, the names of the operations defined on the data type, and the specifications of the defining operations of the type. The access of the user to objects of the data type is restricted to the defined operations of the type; the internal representation and structure of the objects of the type is unknown to the user and cannot be directly manipulated. Therefore the representation and implementation of the data type may be modified, provided only that the behavior of the operations defined on the type is preserved.

As an example, the data abstraction "stack" can be defined in terms of operations such as "push," "pop," "top," and "test-empty." The user can manipulate objects of type stack by these stack operations without having to know how the stack is actually implemented, e.g., as an array, a list, or another data structure. Note that the implementation of the data type is given in terms of already defined types, e.g., a stack might actually be implemented as a list. The operations on the objects of the abstract data type are programmed in terms of operations on the already defined underlying types. In turn, these underlying types may also be user-defined data types and so on until some lowest-level or "primitive" types are reached which have been defined by the hardware or by the implementation of some programming language.

When applying abstract data type methodology to dynamic integrity, the data base is essentially specified as an abstract data type with predefined operations. Since these predefined operations are guaranteed to preserve the

integrity of the data base, and no other operations are allowed, much of the overhead of checking static constraints at run time is, in theory, eliminated. Of course, not all runtime checking can be eliminated, i.e., runtime data must be checked to make sure it does not violate any integrity constraints. However, since the integrity checks can be tied to particular operations, no unnecessary checking need be done, and thus the enforcement overhead is minimized. Further, the abstract data type allows local exceptions to integrity constraints, i.e., integrity can be violated during the execution of an operation as long as its truth is restored before the operation is concluded. Thus problems are eliminated, as in System R, where the programmer has to specify not to check integrity between certain points.

Two approaches to abstract data type specification have been applied to data bases – operational specification and axiomatic specification. In operational specifications, an operation is described by giving an algorithm or program which computes the intended operational behavior on an abstract model of the type. This program differs from an implementation because simplicity is important while efficiency is not. It is not even necessary to have an implementation for the specification language, although it must have a precisely defined meaning. Weber [14,15] uses an operational approach to define a data base as a graph structure where the nodes are abstract data types and the arcs represent an "is part" relationship between abstract objects.

Several problems exist with the operational approach. One is that there is no explicit description of the integrity specifications of the operations. This leads to problems in checking for accuracy, consistency, and completeness, and difficulty in modification of integrity rules. Further, since constraints are not specified explicitly, severe restrictions must be made on the sharing of data structures between views since there is no way to guarantee that one view will not change an invariant of another view. Finally, hiding the integrity specifications in the operations causes problems for the user since the invariant properties of the data base objects provide information that the user needs to know in order to use the operators correctly and to guarantee that integrity is not inadvertently compromised.

Many researchers have attempted to described a data base or data base model axiomatically. In the axiomatic approach to abstract data type specification, the data type is defined by giving axioms relating the operations.

Melkanoff and Zamfir [10] have described the hierarchical, network, and relational models as abstract data types. Maibaum [8] and Colombetti, Paolini, and Pelagatti [3] have applied many-sorted algebras to data base modelling. Also, Ehrig, Kreowski, and Weber [4] and Lockemann, Mayr, Weil, and Wohlleber [7] have applied the axiomatic method to data bases and have defined integrity constraints as pre-conditions, coincidence conditions, and post-conditions using axioms for the operators defined on the type.

There are still many outstanding problems with the axiomatic approach. For example, axiomatic and algebraic specifications are difficult to construct and read and may not be realistic in a commercial environment. Further, attempts to axiomatically specify a data base have in general not dealt with the problems of the sharing of abstract objects with axiomatic type definitions. Obviously shared data is an important and integral part of a data base system, and therefore must be dealt with before the axiomatic approach to data base design can be considered practical.

The specification of the behavioral abstraction in this research is done using I/O specifications in the style of Alphard [17]. In this approach, invariants are specified for the abstract type while preconditions and postconditions on

each operation relate the state of the representation of the type before execution of the operation to the state after execution of the operation. I/O specifications were chosen because they are comparatively easy to construct and understand, they provide for a complete static specification of integrity constraints while also tying specific constraints to operations (thus limiting the amount of constraint checking necessary in enforcement), and they provide a solution for the difficult problems of shared data.

3. Requirements of an Information System

Considering the current state-of-the-art of information system design and integrity and the remaining problems as discussed above, what is needed is a methodology which has the following characteristics:

1) The methodology should be data model independent. Most of the proposed systems for specifying integrity are based on the relational model. The scope of the system should be general enough to include any data model, and should allow for subsequent changes of model.

2) The methodology should provide for a complete, formal specification of integrity constraints. Having a concentrated collection of integrity specifications will impose structure and discipline on the integrity specification process. Further the specification should be high-level and abstract, i.e., in a manner which is relevant to the problem domain. Both the static constraints and the legal manipulations compatible with these constraints should be part of the specification. Finally, the specification should be complete. Completeness implies that the structural constraints, i.e., those usually included as part of the data model, are expressed explicitly. In this way, data model independence will be possible. If the underlying data model upon which the implementation of the data base is based is changed at some future time, for example from a hierarchical to a relational data base structure, it should not be necessary to start again from scratch. In fact, the integrity specification should need to be modified only if something in the real world system being modelled is altered.

3) Different users of the data base may need different abstract views of the data base with very different sets of semantically meaningful operations. It must be possible to ensure the consistency of these operations, e.g., to ensure that one user does not change a data value in a way which conflicts with another user's constraint for that data value.

4) It is not enough to just specify the integrity constraints. The system must provide an efficient means to enforce them. To achieve this efficiency, it must be possible to prevalidate the operations when possible to ensure that they preserve the constraints. Sometimes this prevalidation is not possible, for example, when the integrity constraint involves runtime input data or when the legality of a particular operation depends on the state of the data base at the time that the operation is performed (op1 may be legal if the data base is in state S1 but not legal if the data base is in state S2). In order to reduce the run time overhead for integrity checking, it must be possible to associate the integrity constraints with operations in such a way as to carry out only the absolute minimum number of checks in conjunction with particular updating operations. Further, it should be possible to prevent the execution of operations that violate the integrity of the data base instead of allowing illegal operations to take place and then dealing with recovery problems. The system should also provide flexibility as to when the assertions are checked. In the middle of a complex update, the assertions may not hold, and checking must be deferred until after a group of data base changes has been completed. To provide complete flexibility the writer of the operation should be able to specify when

the checking is to be done. Finally, the system should provide freedom in the specification of the violation action that is to occur when an integrity violation is detected.

5) The uses of a data base are not fixed, but evolve over time. Real world systems also change over time. Therefore, there must be some way for the user's abstract view of the data base to evolve over time. It must be possible for constraints and operations to be added, modified, or dropped.

6) The system should provide control over the operations performed on the data and the context in which they can occur. Privacy protection in data base systems has traditionally been thought of as control over what information a user can obtain from a data base. Minsky [11] has defined another type of privacy called "intentional resolution" or control over what the user is allowed to do with a piece of information supplied to him. In today's data base systems, there is no way to condition the supply of information on its intended use, and the choice is to release either too much information or too little. Some information retrieved from the data base is needed by the program, but not the programmer. In order to get some information that the user needs, the user's program may need to obtain information which should not be released to the user.

4. Applying Behavioral Abstraction to Information Systems

An abstract data type defines data objects that can be manipulated by using predefined operations without needing to know the details of how the data is represented or how the operations are implemented. A data base management system also makes it possible to manipulate data without having to worry about storage details. Thus it appears that the abstract data type of the programming language field and the abstraction involved in data base management systems are both used to accomplish the same goals. In general, data abstraction is most important when the data has a complex structure, when the data might get into an inconsistent state, and when the data needs to be protected for reasons of privacy or security [6].

Abstraction in the data base field has been used to achieve data independence by letting the application programs relate to the data in terms of a logical representation rather than a physical representation. In most data base systems available today, this logical representation, then, is an abstraction of the actual physical representation and structure of the data in storage. The structure is abstracted in terms of some data model, for example, a hierarchy, network, or set of relations. These data models all have the common feature of being representational-- although they provide conceptual data structures, rather than physical ones, these models still describe the structure of the data at some level, and the operators provided are tied to these structures. The usual abstract operators of GET, PUT, DELETE, and MODIFY are defined for the data base user in terms of these structures and thus are tied to the underlying data model. For example, the user of a network structured data base must "navigate the data base" by following chains of pointers in order to manipulate the information in the data base. The amount of data independence will therefore depend on how independent the logical structure is from the physical representation. In most commercially available data base management systems, actual data independence is very limited. Further the ability to express properties of objects varies from one data model to another. Some data models are too restrictive, e.g., a many-to-many relationship is difficult to represent in both the hierarchical and network models, while other models are too general, e.g., the relational algebra operations in the relational data model

do not necessarily produce meaningful results.

Instead, what is proposed is an abstraction concentrating on the operations associated with the data *objects*, instead of the operations associated with the data *structures*. Data base objects are defined using the mathematical viewpoint which defines a data type by the definition of the operations that are applicable to objects of that type. It is argued that since the user of the data base is interested only in the behavioral characteristics of certain objects, then the abstraction of the data presented to the user should involve only these behavioral characteristics. The introduction of other properties can serve only to confuse the user. For example, a user or application program of a library data base may deal with the abstraction "book" and operations defined on this abstraction such as checkout, reserve, etc. These abstractions are relevant to the user, but information about how the books are stored, e.g., as a set of relations, is not. This information is important only to the person who actually implements the operations defined on the object "book." This implementation may perhaps be done using a data management system and one of the standard data models. However, the interactions of the users with the data base will be in terms of the abstract operations defined on the abstract objects, and will therefore be on a semantically high-level.

Views can be provided to different classes of users by creating behavioral abstractions which specify the operations which can be performed by those users. For example, a telephone operator might be able to access information about phone numbers, but an operation to make changes in the directory might not be provided in the operator's view.

The methodology being presented in this research does not imply any redefinition of accepted data base concepts. Instead it provides a conceptual framework for the precise definition and understanding of these basic concepts. Abstraction is a powerful tool for understanding and coping with complexity, and this paper presents a method of behavioral abstraction which will be of benefit in dealing with complex application and data base systems. The methodology does not define a new data model– in fact, the term information model might be used to stress the fact that semantic rather than representational aspects are being emphasized. Any of the conventional data models might be used in the implementation of the behavioral abstraction. Because behavioral abstraction involves not only the design of the data base, but also the programs which use it, the term information system will be used to denote this more general context.

The methodology being proposed involves a closed system, i.e., one in which the update operations on the data base are predefined. All possible update operations need not be predefined, but only a characterizing or minimal set. This characterizing set must be such that any other operations on the data may be achieved by a combination of operations from the set. Integrity problems in information systems arise from the fact that users have complete control over the operations on the data. It is instead proposed that behavioral abstractions provide control over the operations performed and the context in which they can occur. The alternatives are 1) to let the user have complete freedom by providing him with traditional update operations like add, delete, and modify and somehow have the information system catch his errors, or 2) to provide the data semantics as part of the data structure model so that integrity errors cannot be introduced. Neither of these approaches is satisfactory or realistic.

Minsky [11] has demonstrated that the traditional update and retrieval operations used for files (such as add, delete, and modify) are not satisfactory as primitives for interaction between a user and the data base. He argues that there are fundamental differences between files and data bases besides size and complexity. Whereas a file is essentially a meaningless collection of bits whose

interpretation is completely determined by the user, a data base has an intrinsic meaning which is independent of its interaction with users -- it is a representation of knowledge about some real life application system. With files, the programmer is working within an isolated, closed system and has complete control and complete knowledge. But the data base user operates with incomplete knowledge about the environment. In other words, the user sees only an abstract image of the data base, and therefore needs special abstract operations which are appropriate for his specific abstract image of the data. That is, the interaction between the user and the data base must be controlled by the data base, or the intrinsic structure of the data might be violated. Access control is not enough; the data base must be able to "control" the users program.

One way of providing this control is to specify integrity constraints, i.e., the legal states of the data base. Whenever an update occurs, the data base management system can check to make sure that no integrity constraint is violated. But if every update is considered as a primitive operation and is checked for legality, updating may not be possible at all, i.e., in the process of a multi-step update, the data base may be temporarily in an illegal state. Therefore, updating individual objects cannot be the *only* primitive operation available. The user must be able to group primitive operations into logical operations, or "transactions." More important, since the integrity rules must be checked every time any change is made to the data base, enforcement can be so inefficient as to be unrealistic in most cases.

Another option is to build the semantics of the data base into the data model. Existing data models support only a limited number of abstractions and semantic specifications. Attempts are being made to build more complex data models, but it is not clear that a universal data model exists. Just as there is no universally accepted programming language which is adequate for all situations, universal acceptance of a data model may not be possible. The other alternative is to provide users with a vast number of data models from which to choose one which suits their particular situation. Currently, the data model solution to the problem is not realistic.

So we are left with pre-specified and pre-verified operations. This is not really so terribly unrealistic – for most systems the operations on a data base can be predetermined. Furthermore, such pre-specification is an advantage from a design standpoint. Several database design methodologies rely, at least to some extent, upon such an approach [12,16]. In addition, current philosophies for software system design, e.g., top-down design, recommend that the operations on logical data structures be specified prior to the actual physical design of the structures. It is necessary, however, that our system somehow provide for "unpredicted" or new operations to be added to the specification, i.e., the specifications must be modifiable.

5. A Behavioral Abstraction Model

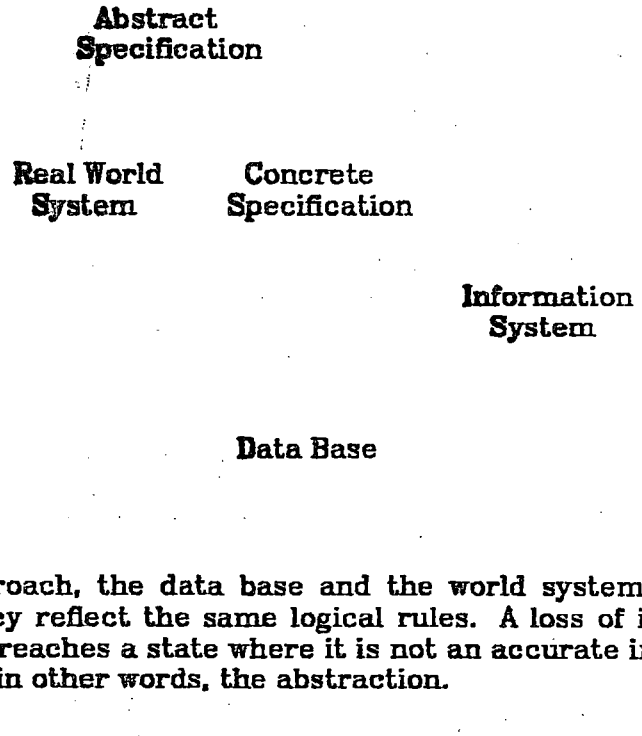
A data base is not just a collection of values stored on some computer storage device. Rather it is also a symbolic representation of knowledge about some world system. But this representation is accurate only if there is some detailed correspondence between the contents of the data base and the world system. Thus the data solely within the computer cannot be said to be accurate (to have integrity) or inaccurate. Stored data takes on the attribute of accuracy only when considered with respect to some world state. At any given time, if the stored data describes a permitted state of the world system being represented, then the data base has integrity. But if some feature of the world

system is misrepresented, then the data base is inconsistent with respect to the system being modelled.

Consistency then implies that a correspondence exists between the application system being modelled and the contents of the data base. Instead of trying to make a direct mapping between the world state and the data base, an abstraction should first be made of the relevant properties of the world system. Then both the stored data and the world state at any time are taken to be models of this same abstraction. This abstraction is a formal statement of the properties which are shared by the real world system and by its representation inside a computer. Since an application system is governed by a set of rules that determine which of its configurations are reasonable, every correct version of the data base must preserve these rules. Furthermore, the operations defined on the data base must also preserve these rules so that the results of these operations can be successfully applied back to the real world.

Thus the first step in data base design using this model is to produce an abstract specification. This abstract specification is essentially a logical statement of the relevant properties which the real world and the information system must have in common. The users who interact with this abstraction would theoretically be unable to tell whether they were dealing with the real world system or the computerized system. Once the abstract specification is developed, then the mapping (implementation) of this abstraction into computer terms, or the concrete specification, can be done. Note that no matter what changes are made to the concrete specification, the abstract specification remains valid. The only circumstance in which the abstract specification must be altered is if the real world system which it models is changed.

Diagrammatically this model looks like:



Using this approach, the data base and the world system are allowed to differ as long as they reflect the same logical rules. A loss of integrity occurs when the data base reaches a state where it is not an accurate interpretation of the logical rules, or in other words, the abstraction.

6. The Abstract Specification

The user of the information system is not really interested in the storage or representation of information, but is instead concerned with data objects and the things that can be done to these objects, i.e., how they can be manipulated. For example, the teller in a bank, when dealing with the object "bank account," is unconcerned about whether the bank accounts are stored hierarchically or as a set of relations. Instead, the teller (or application program writer) has a need to interact with the "bank account" object using certain operations such as withdrawal, deposit, or change-address. These operations in turn are abstract operations in that they may actually have quite complex, multi-step implementations which contain elaborate provisions to make sure that the integrity of the data base is maintained. However, the way the operation is carried out is irrelevant to the user who needs only to know the types of the objects available, and the "meaning" or behavioral semantics of the operations.

Thus the abstract specification must contain the behavioral semantics of the operations. The users of the abstract objects will be unaware of how the operations are actually carried out, but will be guaranteed that the implementation will obey these behavioral semantics. The implementor of the operations will have complete freedom in how the objects and operations are actually implemented with the restriction that the implementation must conform to the behavioral semantics defined for the operation.

As an example, in a library data base, the abstract object "book" might have the operations "checkout," "checkin," "renew," and "is-checked-out?" defined on it. Each of these operations might also have parameters:

checkout(book, cardholder, checkout-date, due-date)
checkin(book)
renew(book)
is-checked-out(book) **returns** boolean

Since a data base is initially empty, all data objects must be brought into existence through the execution of a "create" operation. This create operation can be thought of as a generator for instances of the abstract object.

In order to provide control over the use of an abstract object, it is convenient to specify the operations one object may perform on another object. For example, the operation "checkout" may *import* separate rights for each parameter of the operation, e.g.,

checkout (book, cardholder<verify>, checkout-date, due-date)

which states that checkout may invoke the verify operation defined on the abstract object cardholder.

The abstract specification must also contain the logical rules or invariants of the real world system. For example, in a library data base, certain rules or constraints are expected to be followed: books are checked out only to cardholders, no book can be checked out to two different people at the same time, no cardholder can have more than the maximum number of books checked out at any given time, etc. It is necessary that the behavioral semantics of the operations defined on an abstract object preserve the invariants of the abstract object.

The data base system should allow the specification of any consistency and validity constraint. Further, the specification should be in the terms used by the application specialists. Specification of constraints in "computerese," such as in terms of relations, means that the people who understand the application

system best and are best prepared to judge the validity and completeness of the constraint specification, are the least likely to understand the specification.

A complete specification must include the relevant properties which are shared by the application system, the abstract specification, and the concrete specification (implementation). When a data base is defined using behavioral abstraction, this specification includes both the static characteristics (semantic invariants) of the abstract object, and the behavioral characteristics (operations) which are compatible with (preserve) these invariants. Thus, instead of defining integrity using either a static specification or a dynamic specification, as in previous approaches to the problem, it is proposed that the semantics of a data base be defined as *both* the semantic invariants and the changes allowed.

In order to completely specify a change, without giving an algorithm, input-output specifications for the operations will be used. That is, the operations will be specified in terms of the conditions under which the operation is allowed (the preconditions) and the changes that will result (the postconditions). Input-output specifications were chosen instead of an algorithmic specification in order to make the specifications less implementation-oriented, and thus to specify "what" without "how." Using this approach, the problem of static constraints not holding during multi-step updates is eliminated since an abstract data type allows for local exceptions to the invariants as long as their truth is restored before control leaves the operation. In summary, specifying the semantics of a data base which has been defined using behavioral abstraction involves specifying the legal initial states of the abstract objects, an abstract invariant, and input and output constraints for each abstract operation. In order to do this, it is necessary to first define an abstract image for each abstract object.

6.1 Abstract Image

The question now arises as to how the semantics of operations will be specified. We are restricted by the fact that we do not want to describe the behavior of an operation by the effect it has on an implementation or storage representation. Instead, the semantics of the operations will be defined in terms of the effect of the operation on an abstract image of the object. It is convenient to define the abstract image in terms of abstractions from mathematics, such as sets, sequences, and tuples, which have well-defined properties. Using the university example, the abstract image of class might be defined as the tuple

<id:class-id, cname:name, prof:professor, loc:room,
ctime:time, max#stud:integer, students:set of student>

and the abstract image of the object "schedule" might be a set of classes.

Note that although an abstract image is defined in the abstract specification, the implementation is not constrained by the abstract image. The implementation must only have the same *behavior* as the abstract specification. That is, to describe the properties of an abstract specification, it is convenient to imagine a specific image, but this image is only hypothetical and does not describe any real implementation.

6.2 Abstract Invariant

The abstract invariant, Ia, specifies the properties that the real world application system and the data base system must share. These invariants limit the allowable states in the system being modelled and are of two types– value

constraints and inter-object constraints. The value constraints are the set of legal values of instances of the data abstraction. For example, an associate professor's salary might have legal values from $20,000 to $25,000, or the legal values of a student's year-in-school might be the integers from 1 to 5. The data object "null" or "undefined" may be present in the description of the legal values an object may assume. There may also be inter-object constraints, or constraints on the relationships allowed between objects. The inter-object constraints of a particular abstract object would consist of assertions which completely specify the possible relationships among the objects in the abstract image of the object being defined. An example of an inter-object constraint is that the number of students enrolled in a class is less than the maximum number of students allowed for that class.

It is possible to write abstract specifications without an abstract invariant. The abstract invariant is eliminated by including the constraints specified by the abstract invariant in the input and output constraints for the abstract operations of the type. This is analogous to saying that a static specification of integrity can be replaced by a dynamic specification. However, the specifications required to do so become quite complicated, and therefore it is recommended that an abstract invariant be used.

6.3 Abstract Operations and Pre and Post Conditions

The abstract operations are defined, in this methodology, by the input and output constraints which characterize the effects of the operations. In Hoare's notation, this is written

$$C_{pre_j} \{S_j\} C_{post_j}$$

where C_{pre_j} and C_{post_j} are the input and output constraints respectively for operation j, and S_j is the code for operation j.

These pre and post constraints specify the legal manipulations on the abstract object. The integrity of the data can be violated in two ways: 1) illegal changes in values, and 2) legal changes which are made in an illegal order. The pre and post conditions on the operations of the type must prevent these two events from happening. As as example of ensuring that illegal values are not introduced, suppose that the maximum number of students in a particular class, in the above example, is 30. Then a pre-condition for the operation "enroll-student" would be "number of students < maximum number of students" and a postcondition is "number of students ≦ maximum number of students."

Specifying the legal sequence of operations is important when the set of legal operations depends on the characteristics of the data base state such as the occurrence or non-occurrence of certain operations in the past, at the same time, or in the future [7]. As an example, a precondition for dropping a class is that the value of "enrolled?" is true (and the postcondition is that "enrolled?" is false). To specify that certain actions must take place at the same time, for example adding a student to a class means that the number of students enrolled is incremented by one, a postcondition can be used – e.g., the postcondition for the operation "enroll-student" is "number of students equals previous number of students + 1."

Every abstract object must have a create operation defined for it. In certain cases, as abstract object may be created only when certain conditions

are satisfied by the environment in which it is created. Information about the environment is passed to the type in parameters, and thus the initial preconditions are constraints on the values of the parameters used to create an object. If there are no preconditions, then the specification of the initial preconditions is merely "true." The postcondition in this case specifies assertions about the value of the instance returned by the create operation.

An example of a complete abstract specification for the type course can now be given. Although for purposes of example one particular method of formally writing assertions has been selected, any other method could be used.

```
type class
 abstract specifications
  abstract image
   class=<id:class-id, cname:name, prof:professor,
          loc:room, ctime:time, max#stud: integer,
          students:{student}
  abstract invariant
    [number of students enrolled is less than or
    equal to the maximum allowed]
    0≤cardinality(students)≤max#stud
  operations
    create-class (id: class-id, n:name, p:professor,
        r:room, t:time, max:integer) returns c:class
     pre  10<max≤100
     post c=<id:id, cname:n, prof:p, loc:r,
            ctime:t, max#stud:max, students:{}>
    assign-professor (c:class, p:professor)
     pre true
     post c.prof=p
    add-student (c:class, s:student)
     pre 0≤cardinality(c.students) < c.max#stud &
       enrolled?(c,s) = false
     post c.students = c.students' ∪ {s}
    delete-student (c:class,s:student)
     pre enrolled?(c,s) = true
     post c.students = c.students' - {s}
    enrolled?(c:class,s:student) returns b:boolean
     pre true
     post (b= s  c.students)
    change-room(c:class, r:room)
     pre true
     post c.room=r
    change-time(c:class, t:time)
     pre true
     post c.time=t
```

The specification of the pre and post constraints for the operations is important for three reasons: 1) the checking of particular constraints is tied to particular operations, 2) the pre and post constraints for the operations act as a guide for the implementor, and 3) the pre and post constraints will be used to prove that the implementation is correct.

7. Verification

Verification of integrity will be defined as proving that the abstract operations preserve the abstract invariant. Thus verifying the integrity of the data base involves induction -- first prove that the abstract objects have the specified properties (satisfy the abstract invariant) when they are first created, and then show that all operations which change the object preserve these properties.

Formally, prove that:

1) The create operation establishes the abstract invariant

$$C_{pre_i}(a) \;\&\; C_{post_i}(a) \Longrightarrow Ia(a)$$

where C_{pre_i} and C_{post_i} are the pre and post conditions

respectively on the create operation, and a is an abstract object.

2) Show that each operation j preserves the abstract invariant,

$$C_{pre_j}(a') \;\&\; Ia(a') \;\&\; C_{post_j}(a) \Longrightarrow Ia(a)$$

where a' is the abstract object prior to the execution of the operation.

Using the class example specified above, the proof would consist of the following steps:

1) Prove that the abstract invariant, Ia, holds after the creation of the abstract object class:

Show: 0<max≤100 & c=<cid:id, cname:n,
prof:p,loc:r,ctime:t,max#stud:max,students:{}>
==> 0≤*cardinality*(c.students)≤c.max#stud

Proof: 0<max#stud≤100
==> 0=*cardinality*({})<c.max#stud≤100
==> 0≤*cardinality*(c.students)≤c.max#stud

2) Prove that the abstract invariant holds after each abstract operation. The only two operations for which the proof is not trivial are add–student and delete–student.

a) for add–student

Show: 0≤*cardinality*(c.students')<c.max#stud &
enrolled?(c',s)=false & c.students= c.students' ∪ {s} &
0≤*cardinality*(c.students')≤c.max#stud
==> 0≤*cardinality*(c.students)≤c.max#stud

Proof: *cardinality*(c.students' ∪ {s} = *cardinality*(c.students') + 1
Since 0≤*cardinality*(c.students')<c.max#stud,
then 0≤*cardinality*(c.students)≤c.max#stud

b) for delete–student

Show: enrolled?(c',s) = true &
0≤*cardinality*(c.students')≤c.max#stud
& c.students=c.students' - {s}
==> 0≤*cardinality*(c.students)≤c.max#stud

Proof: 0≤*cardinality*(c.students)≤c.max#stud
cardinality(c.students')= *cardinality*(c.students) - 1
Since *cardinality*(c.students')≤c.max#stud,
then *cardinality*(c.students)≤c.max#stud
Also since enrolled?(c',s)=true, then 0<*cardinality*(c.students')
and therefore 0≤*cardinality*(c.students).

8. Enforcement of Integrity

Enforcement of integrity requires that no update operations be allowed which can cause integrity violations. Integrity problems can occur from an illegal sequence of legal operations or from faulty operations that make updates which violate integrity rules. Preconditions on operations can eliminate illegal sequences of operations, and verification techniques can prove that the abstract operations preserve the abstract invariant. It remains to implement the abstract specification in the concrete specification and to verify that this implementation satisfies the abstract specification.

The concrete specification describes an implementation of the abstract specification. This implementation must have the same behavior as the abstract specification, but is not constrained to using the same representations and algorithms for realizing that behavior. Thus the abstract specification acts as a guide for constructing the concrete specification in terms of some lower-level data model. It is important to realize that the concrete specification merely represents a different level of abstraction and that any number of levels of abstraction are possible.

Verifying the correctness of the concrete specification includes proofs of the correctness of the implementations of the operations with respect to their pre and post conditions using assertions for runtime checks. Programmer control over where assertions are placed within the operations allows for flexibility in where the runtime checks are actually done.

In order to verify that the concrete specifications correctly implement the abstract specifications, it is necessary for the concrete specifications to contain a representation, a concrete invariant, an abstraction function abs (the mapping between the concrete or physical representation and the abstract image), and concrete input and output constraints for each of the implementations of the abstract operations.

A possible implementation for the abstract type course follows:

```
concrete specifications
  representation: db record (key is cid)
                    cid: integer;
                    title: name;
                    cprof: professor;
                    croom: location;
                    hour: time;
                    max: integer;
                    no-of-stud: integer;
                    stud-list: list of student
                  end record
  concrete invariant
    0≤no-of-stud≤max
  abstraction function
   (cid, title, cprof, croom, hour, max, no-of-stud,
    stud-list)= <id:cid, cname:title, prof:cprof,
    loc:croom, ctime:hour, max#stud:max,
    studlist:{s |   i, 0≤i≤no-of-stud,s=stud-list[i]}>
  operations
  assign-professor (out c.prof=p)
  add-student (in enrolled?(c,s)= false & c.no-of-stud<c.max
          out c.stud-list= add(c.stud-list'+1 & c.no-of-stud=c.no-of-stud+1)
  delete-student (in enrolled?(c.s)=true
```

```
                out c.stud-list= delete(c.stud-list',s)
                    & c.no-of-stud= c.no-of-stud-1
enrolled?(c,s) (out b= searchlist (c.stud-list,s))
```

Note that the above abstraction function may be many-to-one. That is, more than one concrete object may represent the same abstract object. As one case of this, sometimes it is convenient to maintain some information in the data base to help simplify the checking or enforcement of invariants. For example, if an invariant states that a student cannot enroll in more than seven classes, it is helpful to maintain the total number of classes for each student and update the total as necessary rather than to continually recalculate it.

In the Alphard methodology, the proof of the correctness of the concrete realization consists of four steps:

1) show that the data structures used in the implementation constitute a valid representation of the abstract concept, i.e.,

$$Ic(x) \Longrightarrow Ia(abs(x))$$

where abs is the abstraction function.

2) show that the initialization performed when an object of the type is created produces a legitimate representation of an abstract object. That is, if the initial precondition, C_{req}, is satisfied before executing the initialization S_{init}, then the concrete invariant as well as the assumption about the initial value, C_{init}, hold after the initialization code is executed.

$$C_{req} \{S_{init}\} C_{init}(abs(x)) \ \& \ Ic(x)$$

And for each operation j and concrete object x,

3) show that each operation body satisfies its input and output constraints and preserves the concrete invariant

$$C_{in_j}(x) \ \& \ Ic(x) \ \{S_j\} \ C_{out_j}(x) \ \& \ Ic(x)$$

In order to verify all the constraints, it may be necessary to generate and insert assertions into the code of the implementation.

4) show that the concrete operation is applicable whenever the abstract precondition holds, and that if the operation is performed, the results correspond properly to the abstract specifications, i.e., establish the relationship between the concrete input and output constraints and the abstract pre and post conditions.

$$C_{pre_j}(abs(x)) \ \& \ Ic(x) \Longrightarrow C_{in_j}(x)$$
$$C_{pre_j}(abs(x')) \ \& \ Ic(x') \ \& \ C_{out_j}(x) \Longrightarrow C_{post_j}(abs(x))$$

where x' stands for the value of the concrete object x before the concrete operation is executed.

For the course specification, step 1 requires showing that:

0≤no–of–stud≤max
==> 0≤*cardinality*(abs(students))≤abs(max#stud)

Proof: *cardinality*(abs(students)) =
cardinality(no–of–stud,stud–list)= no–of–stud

Steps 2 and 3 can be accomplished using standard verification methods and thus mechanical verification aids. Step 4 establishes the relation between the concrete input and output constraints and the abstract pre and post constraints for each operation. For the step 4 proofs, the following relationships will be assumed between list and set operations:

add(s:list, e:element) ==> s:=s' ∪ e
delete(s:list, e:element) ==> s:=s' - {e}
searchlist(s:list, e:element) = (e s)

The proofs for step 4 are:

1. for the operation assign–professor
   show: 0≤c.no–of–stud≤c.max & true ==> true
   proof: immediate
   show: 0≤c.no–of–stud≤c.max & true & c.cprof=p ==>true
   proof: immediate
2. for the operation add–student
   show: 0≤c.no–of–stud≤c.max & enrolled?(c,s)=false
   & *cardinality*(abs(c.students))<abs(c.max#stud)
   ==> enrolled?(c,s)=false & c.no–of–stud<c.max
   proof: immediate except for
   cardinality(abs(c.students)) < abs(c.max#stud)
   ==> *cardinality*(c.no–of–stud,c.stud–list)<c.max
   ==> c.no–of–stud < c.max
   show: cardinality(c.students')<abs(c.max#stud') &
   enrolled?(c,s)=false & 0≤c.no–of–stud'≤c.max' &
   c.stud–list=add(c.stud–list',s) & no–of–stud=no–of–stud'+1
   ==> abs(c.students)= abs(c.students)' ∪ s
   proof: abs(c.students)==> (c.no–of–stud,c.stud–list)
   ==> (c.stud–list'+1, add(c.stud–list',s))
   ==> (c.no–of–stud',c.stud–list') ∪ c.studlist[no–of–stud]
   ==> abs(c.students)= abs(c.students') ∪ {s}
3. for the operation delete–student
   show: enrolled?(c,s)=true & 0≤c.no–of–stud≤c.max
   ==> enrolled?(c,s)=true
   proof: immediate
   show: enrolled?(c,s)=true & 0≤c.no–of–stud'≤c.max'
   & c.no–of–stud=c.no–of–stud'-1 & c.stud–list=delete(c.stud–list',s)
   ==> abs(c.students) - {s}
   proof: abs(c.students) = (c.no–of–stud,c.stud–list)
   ==> (c.no–of–stud'-1,delete(c.stud–list',s))
   ==> (c.no–of–stud',c.stud–list') - {s}
   ==> abs(c.students)= abs(c.students') - {s}
4. for the operation enrolled?
   show: true & 0≤c.no–of–stud≤c.max ==> true
   proof: immediate
   show: true & 0≤c.no–of–stud'≤c.max' & b=searchlist(c.stud–list,s)
   ==> b= s (c.students)
   proof: b=searchlist(c.stud–list,s)

==> b= s (c.stud–list,s)
==> b= s abs(c.students)

9. Views and Shared Data Issues

A user does not interact directly with a data base, but instead interacts with an image, or view, of it. The view becomes in effect a virtual data base [11]. A number of views of a data base will usually exist because a data base is shared, and different users may need to view the data in logically different ways since they are using the data base for different purposes. Further, a user usually has incomplete knowledge about the data base on which he operates. This is the result of practical reasons (data bases are usually too large for a user to be expected to be familiar with the entire data base or even to be interested), and for security reasons (certain data may be deliberately hidden from particular users).

In the behavioral abstraction model of a data base, a view may be considered as an abstraction which is determined by the operations that can be performed by the class of users. These operations are, of course, abstract operations which are specific to the view of the data base with which the user interacts. Different views may then incorporate different subsets of data base objects, may have both nonoverlapping objects and objects in common, may import only certain rights, and may see the objects they share involved in different relationships. Therefore, a view is just a behavioral abstraction and really needs to be treated no differently than any other object in the data base.

For the most part, different classes of users are given different and nonoverlapping views that have a very small, and possibly empty, intersection of data objects. In cases where the views involve disjoint portions of the data, they may be specified and implemented as described so far. However, if the views are allowed to manipulate shared objects (and not just copies of the same data), the views must agree, at least to some extent, on the semantics of the shared objects. Otherwise, integrity may be lost when one view alters an object in a way which makes the integrity constraints of another view no longer true.

The problem then is that of ensuring that the invariants of the abstract type are consistent, that is, that the operations of one type do not violate the invariants of other types which are bound to the same instance. There are two ways of ensuring consistency -- through a data abstraction of a "composed" object or by methods for ensuring the consistency of separate abstractions.

9.1 Ensuring Consistency

If a shared object d1 is in some relationship with another object d2, then this relationship (composed object) may be defined as a separate abstract object where the relationship is specified in the abstract invariant of the composed object. All operations allowed on the composed object must, of course, be defined in the abstract type for the object, and thus must preserve the invariant, i.e., the relationship. Views or abstractions which contain the composed object need not see it identically. It is possible to control how an object is shared by using access rights as described above. Thus, one view may only have the right to invoke operations i and j on the shared object, while another may have the right to issue calls to operations j and k.

It should be noted that the distinction between entities and relationships often made in the data base literature is a specious one. Relationships can be treated as merely composed objects. In fact, every entity represents a

relationship among its attributes. As an example, the relationship "is-reserved-by" which, in a library, connects a cardholder and a book, could also be defined as an abstract object "reservation." Thus, entities and relationships can be represented in the same way as long as the capability exists for defining an entity as the composition of entities and for defining the characteristics of the composed entity which comprise the relationship between the component entities [2,14].

As an example, let d1 be the number of employees and d2 be the set of employees. When a new employee is hired or an employee is fired or quits, i.e., the set of employees, d2, is changed, it is necessary to alter d1 accordingly. Thus any view which is allowed to alter the set of employees is forced to go through the operations defined on a composed object "employees" which would ensure that the relationship between d1 and d2 is maintained. As another example, consider the type course defined previously. A relationship between the maximum number of students in the class and the number enrolled is defined in the abstract invariant. It is possible to specify that any changes to these objects must occur through operations defined on the object course, or it is possible to define a separate abstract object "class list" which contains this data and which enforces the proper relationship. Then the operations defined on course (and in the other views which share the object "class list") which need to modify part or all of the class list object would be forced to do so by invoking abstract operations defined on class list.

A second way to ensure consistency between abstract specifications involving shared data is to ensure that the implementations of the abstract specifications are *compatible*, i.e., ensure the properties of the shared data. The question then is what relationship must hold between the two concrete specifications in order for the types to be considered compatible.

The concrete representation of an abstract type will consist of several variables c_1, c_2,...,c_n where $n \geq 1$. The concrete representation is related to an abstract object A by the abstraction function such that

$$A = abs(c_1, c_2, ..., c_n)$$

The concrete invariant Ic defines the relationship between the concrete variables c1,c2,...,cn to ensure that they fall within the domain of abs. That is, Ic is the characteristic function of the domain of abs:

$$domain(abs) = \{c_1, ..., c_n \mid Ic(c_1, ..., c_n)\}$$

If the same concrete implementation participates in the construction of multiple views, then an abstraction function will be required for each view where the domain of each abs is the same (although their ranges may obviously differ), i.e.,

$$domain(abs_1) = domain(abs_2) = ... = domain(abs_n) = \{c_1, c_2, ..., c_n \mid Ic(c_1, c_2, ..., c_n)\}$$

If each abstract view has a different implementation, i.e., concrete specification, then the implementations must preserve the properties of and relationships among the shared variables. That is, the invariants must place the same constraints on the possible combinations of values that shared variables

may take. Here shared variables must include not only the concrete variables in the intersection of the domains of the abstraction functions, but also those concrete variables which are related to these variables.

Formally, let k implementations share the objects $c_1,c_2,...,c_n$ for i,k≥1. The k implementations will be said to be *compatible* if and only if the intersection of the domains of the abs function for each implementation is $\{c_1,...,c_i \mid Ics(c_1,...,c_i)\}$ where Ics is the characteristic function of the shared objects. For implementations which have no shared variables, the intersection will be empty, and the implementations will be trivially compatible.

In order to prove that several implementations are compatible, it is necessary to define an invariant Ics which specifies the constraints on the shared objects and to show that

$$Ic_j(c_1,...,c_i) \Longrightarrow Ics(c_1,...,c_i)$$

for each implementation j.

As an example, assume that a relation student exists in the data base with domains $D = \{D_1,D_2,...,D_n\}$, and that three views are defined on this relation:

1) View 1 defines operations only on lower-division students, e.g.,

   Ic_1 = 0≤#units<60 & ...

2) View 2 operates only on upper division students, e.g.,

   Ic_2 = 60≤#units & ...

3) View 3 sees all students but not all of the domains of the relation

   Ic_3 = 0≤#units & ...

To prove that the views are compatible, it is necessary to define the invariant of the relation student

   Istud = 0≤#units & ...

and then to show that Ic_1, Ic_2, and Ic_3 imply Istud.

If a new view which uses the relation student is added in the future, then it will be necessary only to show that the concrete invariant of the added view implies Istud in order to be sure that it is compatible with all other views.

Since the implementations of the operations of the abstract types preserve the concrete invariants, Ic_i, and thus the shared concrete invariant, Ics, these operations must maintain the integrity of the shared object.

9.2 Initialization

One last problem remains. Each abstract specification defines a C_{req} which defines the conditions of the environment necessary to create an abstract object, and these conditions must be shown to be true before the initialization code S_{init} is executed, that is, before an object is created:

$$C_{req}\{S_{init}\}C_{init}(abs(x)) \& Ic(x)$$

where x is a concrete object. Since only one of the abstract objects will actually create the shared constituent objects and C_{req} may differ for the views, it is necessary to include another operator, a "bind," in each view. This operator would be used when a create operation would normally be invoked but the concrete object already exists. B_{ind} guarantees that the initial preconditions assumed by abstract specifications actually hold for the shared objects, i.e.,

$$C_{req} \{B_{ind}\} Ic$$

If the shared objects constitute only a part of the concrete object of the view, the bind operator may be invoked by the create operation.

10. Summary

The methodology presented in this paper provides not only for a complete specification of the semantic integrity constraints, but also includes a practical means for verifying both the specification and implementation of the integrity constraints. Integrity constraints are associated with particular semantically high-level operations so that no unnecessary checking need ever be done. Not only are the integrity constraints involved in an operation explicitly stated so that modifications to constraints can be made, but verification techniques can be used to ensure that both the abstract and concrete operations preserve the constraints.

Some constraints will be proved to always hold, thus eliminating any necessity for execution time checking and greatly reducing runtime overhead. For those constraints which cannot be proved to always hold, enforcement is accomplished by assertions incorporated at the programmer's discretion into the code used to implement the abstract operations. This means that the appropriate assertions can be checked at the appropriate time. There is complete flexibility as to when within the operation the assertion is checked, since an abstract data type allows local exceptions to integrity constraints as long as their truth is restored by the time control leaves the operation. Also, a "violation action" can be specified by the programmer with the assertion so that complete flexibility is possible for specifying what execution time action should take place when a constraint is violated. An integrity system cannot be expected to guarantee the correctness of every value in the data base, but only to enforce the constraints which have been made explicitly. The system being proposed allows complete flexibility in constraint checking. If it is decided that the checking of certain constraints during particular operations is not worth the extra runtime overhead, then the designer has the flexibility of omitting these checks. In fact, the designer has complete control over which constraints are checked in which operations -- although if some checking is omitted, then the complete integrity of the data base cannot be guaranteed. In summary, this methodology allows a complete specification of integrity constraints, pre-verification of constraints, and flexibility as to what and when runtime checks must be made and what violation-action should be taken in case of assertion failure.

It is also important to note one other advantage of applying this methodology to data integrity. Many of the integrity specification schemes involve using computer terminology in the specification of the constraints, e.g., functional relationships between the domains of a relation. But the abstract

specifications proposed herein can be written entirely in terms that application experts use and understand. The integrity and correctness of the information system being designed is thus enhanced by the fact that application experts, who have a natural understanding of the world being modelled, can fully participate in the abstract design and specification of the system.

Finally, it should be understood that just as a program can be tested and used without ever being formally verified, an information system can be designed and implemented and the integrity specified and enforced using the methodology being presented without actually formally verifying the specification or the implementation. Verification is certainly desirable, but in some situations may not be practical. Further, the verification consists of several steps and again not all of these steps need actually be formally completed. Some of the verification can be accomplished with the aid of mechanical verification systems, and when these systems have been perfected, perhaps verification of large systems will become more practical and frequent.

References

[1] Astrahan, M.M. *et al.*, "System R: Relational Approach to Data Base Management", *ACM Transactions on Database Systems*, vol. 1, no. 2 (June, 1976), pp. 97-137.

[2] Brodie, M.L., "Specification and Verification of Data Base Semantic Integrity," Technical Report CSRG-91, University of Toronto, 1978.

[3] Colombetti, M., P. Paolini, and G. Pelagatti "Non Deterministic Languages Used for Definition of Data Models," Internal Report, Instituto di Electronica, Politecnico di Milano, 1978.

[4] Ehrig, H., H.J. Kreowski, and H. Weber "Algebraic Specification Schemes for Data Base Systems," *Proc. 4th Int'l Conf. on Very Large Data Bases*, Berlin, 1978, pp. 427-440.

[5] Hammer, M.M. and D.J. McLeod, "Semantic Integrity in a Relational Data Base System," *Proc. Int'l. Conf. on Very Large Data Bases*, 1975.

[6] Linden, T.A. "The use of abstract data types to simplify program modifications," *Proc. of Conference on Data: Abstraction, Definition, and Structure, ACM SIGPLAN Notices*, vol. 11, Special Issue (1976), pp. 12-23.

[7] Lockemann, P.C., H.C. Mayr, W.H. Weil, and W.H. Wohlleber, "Data Abstractions for Database Systems," *ACM Transactions on Database Systems*, vol. 4, no. 1 (March, 1979), pp. 60-75.

[8] Maibaum, T.S. "Mathematical Semantics and a Model for Data Bases," in *Information Processing 77*, ed. B. Gilchrist. Amsterdam: North Holland, 1977, pp. 133-138.

[9] McLeod, D.J., "High Level Expression of Semantic Integrity in a Relational Data Base System," Technical Report TR-165, MIT Laboratory for Computer Science, Cambridge, MA, 1976.

[10] Melkanoff, M. and M. Zamfir. "The Axiomatization of Data Base Conceptual Models by Abstract Data Types," Technical Report UCLA-ENG-7785, University of California, Los Angeles, January, 1978.

[11] Minsky, N. "On interaction with data bases," *Proc. of ACM 1974 SIGMOD Workshop on Data Description, Access, and Control*, Ann Arbor, Mich., May, 1974, pp. 51-62.

[12] Smith, J.M. and D.C. Smith, "Conceptual Database Design," in *Infotech State of the Art Report on Data Design*, ed. M. Atkinson. Maidenhead, England: Infotech International, 1980.

[13] Stonebraker, M. "Implementation of integrity constraints and views by query modification," *Proc. ACM 1975 SIGMOD Conference*, San Jose, May 1975, pp. 65-78.

[14] 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, San Jose, November 1976.

[15] Weber, H.J., "Modularity in Data Base System Design: a Software Engineering View of Data Base Systems," in *Issues in Data Base Management*, ed. H.J. Weber and A.I. Wasserman. Amsterdam: North-Holland, 1979, pp. 65-91.

[16] Weldon, J.-L., "Using data base abstractions for logical design," *Computer Journal*, vol. 23, no. 1 (1980), pp. 41-45.

[17] Wulf, W. A., R.L. London, and M.Shaw, "An Introduction to the Construction and Verification of Alphard Programs," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4 (December, 1976), pp. 263-265.

[18] Zloof, M.M. "Query-by-example: the invocation and definition of tables and forms," *Proc. Int'l Conf. on Very Large Data Bases*, 1975, pp. 1-24.