CONTENTS

Dedication

Preface

Session 1: Chairman: Dines Bjørner, Tuesday, June 1, 1982

1. H. Ganzinger
   Denotational Semantics for Languages with Modules
   Questions and Answers
   3

2. M.C. Gaudel
   Correctness Proof of Programming Language Translations
   Questions and Answers
   25

3. P. Moses
   Abstract Semantic Algebra
   Questions and Answers
   45

4. N. Christiansen and N. Jones
   Control Flow Treatment in a Simple Semantics-Directed Compiler Generator
   Questions and Answers
   71

Session Discussion Responder: P. Lucas

Session 2: Chairman: C. Böhm, Wednesday, June 2, 1982

1. R.G. Wagner
   Functional Hierarchies of Functional Languages
   Questions and Answers
   107

2. M. Broy
   Fixed Point Theory for Communication and Concurrency
   Questions and Answers
   122

Session Discussion Responder: A. Blikle

Session 3: Chairman: J.B. Dennis, Wednesday, June 2, 1982

1. R. Kuiper and W.P. de Roever
   Fairness Assumptions for CSP in a Temporal Logic Framework
   Questions and Answers
   149

159
### Contents

2. M. Broy and H. Wirsing  
   *On the Algebraic Specification of Flat-tary Infinite*  
   *Communicating Sequential Processes*  
   Questions and Answers  
   171  

3. G.D. Plotkin  
   *An Operational Semantics for CSP*  
   Questions and Answers  
   199  

4. M.C.B. Hennessy and M. Li  
   *Translating a Subset of Ada into CCS*  
   Questions and Answers  
   224  

Session Discussion  
**Responder:** J. de Bakker  

**Session 4: Chairman:** M. Paul, Thursday, June 3, 1982  

1. A. Maggio-Schettini and J. Winkowski  
   *Towards a Programming Language for Manipulating*  
   *Relational Data Bases*  
   Questions and Answers  
   265  

Session Discussion  
**Responder:** E. Neuheld  

**Session 5: Chairman:** J.B. Dennis, Friday, June 4, 1982  

1. J.A. Bergstra and J.W. Klop  
   *Formal Proof Systems for Program Equivalence*  
   Questions and Answers  
   289  

2. P. Pedawitz  
   *Equational Data Type Specifications and*  
   *Recursive Program Schemes*  
   Questions and Answers  
   305  

3. J.-P. Jouannaud, P. Lescanne, and F. Reimig  
   *Recursive Decomposition Ordering*  
   **Session Discussion**  
   **Responder:** G. Cousineau  
   329  

4. J.-P. Jouannaud, P. Lescanne, and F. Reimig  
   *Recursive Decomposition Ordering*  
   **Session Discussion**  
   **Responder:** G. Cousineau  
   331  

5. J.-P. Jouannaud, P. Lescanne, and F. Reimig  
   *Recursive Decomposition Ordering*  
   Questions and Answers  
   349  

**Session Discussion**  
**Responder:** G. Cousineau  

**Session 6: Chairman:** P. Lauer, Friday, June 4, 1982  

1. H.J. Genrich and P.B. Thalagorejan  
   *Well-Formed Flow Charts for Concurrent Programming*  
   Questions and Answers  
   357  

2. I. Castellani, P. Franceschi, and U. Montanari  
   *Labeled Event Structures: A Model for Observable*  
   *Concurrency*  
   Questions and Answers  
   381  

Session Discussion  
**Responder:** G. Cousineau  

**List of Participants**  

**Author Index**  

**Accepted Papers Not Presented**  

- M.A. Casanova, J.M.V. de Castilho, and A.L. Pires  
  *Properties of Conceptual and External Database Schema*  
  409  

- E. Rent  
  *Relational Semantics of Concurrent Programs*  
  (With Some Applications)  
  431  

- R. Rapport  
  *Some Remarks on the Semantics of Concurrent Programs*  
  453  

**Author Index**  

**List of Participants**  

**Session Discussion**  
**Responder:** G. Cousineau  

**Session 6: Chairman:** P. Lauer, Friday, June 4, 1982  

1. H.J. Genrich and P.B. Thalagorejan  
   *Well-Formed Flow Charts for Concurrent Programming*  
   Questions and Answers  
   381  

2. I. Castellani, P. Franceschi, and U. Montanari  
   *Labeled Event Structures: A Model for Observable*  
   *Concurrency*  
   Questions and Answers  
   400  

Session Discussion  
**Responder:** G. Cousineau  

**Session 6: Chairman:** P. Lauer, Friday, June 4, 1982  

1. H.J. Genrich and P.B. Thalagorejan  
   *Well-Formed Flow Charts for Concurrent Programming*  
   Questions and Answers  
   400  

2. I. Castellani, P. Franceschi, and U. Montanari  
   *Labeled Event Structures: A Model for Observable*  
   *Concurrency*  
   Questions and Answers  
   401
The design of database applications is investigated by identifying a number of properties that the conceptual and external schemas must satisfy. The list of properties covers, among others, basic issues in database design, such as consistency of the set of integrity constraints, the design of schemas with built-in operations, the interaction between the conceptual and external schemas and the interplay between several external schemas. The definition of each property, with some restrictions, may be expressed within one of the three formalisms: first-order predicate calculus, dynamic logic and modal logic. The relational model is adopted throughout the development.

INTRODUCTION

This paper addresses the design of database applications within the framework proposed by the ANSI/SPARC Study Group of DBMSs [2]. The ANSI/SPARC proposal divides the database description into three levels: the conceptual schema, describing the enterprise as a whole; the internal schema, describing the physical organization of the database; and the external schemas, describing the way each group of users sees the enterprise. (For alternative structures see [27, 39].)

The basic goal of this paper is to investigate under what conditions the design of the conceptual and external schemas of a database can be considered adequate. This goal is achieved by formally defining a series of properties that the various schemas must satisfy. As a consequence, the relationship between database design and database theory is clarified, and results in database theory are assessed for the design perspective.

Included in the discussion are schemas that contain a predefined repertoire of updates. This extension is both natural and important since it accounts for a large number of "closed" or "menu" database applications, such as airline and hotel reservation, point-of-sale inventory control, electronic banking, and others. Even more important is the fact that it can also be used with "open" database applications where users are still allowed to write full-fledged programs, except that the programs are limited to modify the database only by invoking updates from the predefined repertoire. This is a convenient strategy to enforce integrity constraints, quite in the spirit of the encapsulation strategy from the abstract data type area [29]. In the context of databases, built-in operations were considered, for example, in [73].

This paper is based largely on four formalisms. As far as database concepts are concerned, the relational model of data [14] is adopted. The schema design properties considered are formalized within First-Order Logic, if they do not involve updates, or Dynamic Logic [10, 11, 26], otherwise. Modal Logic [13] is a

* On leave from the Universidade Federal do Rio Grande do Sul
used when dynamic consistency criteria are considered. A survey of the relationship between these formalisms (except Modal Logic) appears in [22].

The literature on schema design theory is reasonably extensive, especially for the relational model. Therefore, other references to related work are delayed to Section 2 when each specific property is described.

The paper is organized as follows. Section 2 contains an informal presentation of the schema design properties discussed. Section 3 formalizes the notions of conceptual and external schemas and their basic properties. Only concepts from First-Order Logic are used in this section. Section 4 repeats the discussion in Section 3 for schemas containing a predefined set of updates. It depends largely on Dynamic Logic. Section 5 formalizes the concept of dynamic consistency criteria using Modal Logic. Finally, Section 6 contains conclusions and directions for future research.

2. AN OVERVIEW OF SCHEMA DESIGN CONCEPTS

In this section we informally describe the basic schema design concepts and properties that will be formalized later. For ease of reference, properties will be numbered and grouped in Figure 2.1 at the end of the section.

2.1 - Basic Concepts

We discuss in this section how a database is described.

A conceptual schema describes the database as a whole. It consists of a set of data structures describing how data is organized in the database and a set of consistency criteria specifying the allowed data values. In the relational model of data, the basic data structures are flat tables, and the consistency criteria can be described as first-order formulas. A language used to describe schemas is called a data definition language.

A set of data values satisfying the consistency criteria is called a consistent or valid database state.

An external schema describes the aspects of the database that are meaningful to a group of users. Just like a conceptual schema, it contains data structure descriptions and consistency criteria. We may also talk here of consistent or valid database states.

We now state a basic property any schema (conceptual or external) must satisfy:

P1: Consistency: The set of consistency criteria must not contain contradictions or, equivalently, at least one consistent state must exist.

Property P1 is, in general, very hard to check. However, there are important special cases where this property is trivially satisfied. In the relational model, almost all classes of consistency criteria considered in the literature [1, 5, 14, 18, 19, 23, 30, 32, 41, 42, 46, 47] are special cases of the extended embedded implicational dependencies (XEDs) [19]. But a set of XEDs is trivially satisfied by the "initial" state, that is, by the state where all relations are empty. Therefore, if all consistency criteria are XEDs, then Property P1 is trivially satisfied.

The following property would also be desirable (although not necessary):

P2: Logical Independence: no consistency criterion is logically implied by the others.

Again, checking this property becomes easy in certain special cases. For example, if all criteria are functional or multivalued dependencies [10, 47], there is an efficient algorithm to check logical independence [38] (for surveys about other aspects of the design of relational databases that have also received considerable attention see [4, 13, 44]).

We conclude this section by observing that the schema definition may also include dynamic consistency criteria imposing restrictions directly on the state transition that can be brought about by updates. Examples are: (1) employee's salaries never decrease; (2) once an employee is hired, he can never be dismissed. Obviously, dynamic consistency criteria, such as those above, would be enforced with the help of auxiliary structures (i.e., names of former employees) summarizing information about past states of the database.

Regarding dynamic consistency criteria, we introduce the following property:

P3: Dynamic Consistency: no state transition (resulting from a sequence of one or more updates) violates any dynamic consistency criteria.

2.2 - Dynamic Aspects of Database Design

By contrast with Section 2.1, we concentrate in this section on the problem of defining how the database can be modified. We assume that we are given a data manipulation language (DML) which can be used to write programs, or updates. For this purpose, we will be concerned with two problems:

(1) What updates are allowed?
(2) Is the DML chosen in some sense adequate?

As for the first problem, we consider that an update u is valid, or preserves consistency, iff u maps the set of consistent database states into itself. This basic requirement on updates was studied in [10, 11, 20, 24].

The second problem requires a longer discussion. We first observe that the database designer may want to specify that certain updates should be available to the users. The availability of updates may be expressed as pairs of formulas (P, Q) that we call update specifications (P and Q are called pre- and post-conditions, respectively).

Update specifications impose a restriction on the DML, which we state as a new property. We say that a program p achieves (P, Q) iff p maps every state satisfying P into some state satisfying Q.

P4: Adequacy of the Schema DML: every update specification must be achievable by some valid update (written in the DML chosen).

Property P4 should be understood as follows. During this stage, the designer anticipates that certain changes in the database will have to be made. He then describes these changes non-procedurally using pairs of pre- and post-conditions. Later, he chooses a schema DML. Finally, he checks if the DML chosen is powerful enough to achieve every anticipated state transition.

Pairs of formulas may also be used to specify how a (static) consistency criterion is to be enforced. For example, consider constraint "every employee must work for some department" and the update "delete department d". Then, the update can be implemented either by deleting d and all employees working in d, or by deleting only the employee works in d. Note that both methods satisfy the criterion, but they are very different. The structural model of [45] indeed has two different types of constraints corresponding to the two interpretations above.

Following the spirit of Property P4, we have two other properties. First, it may be desirable to reach any consistent database state from any other consistent database state. This property is related to the notion of completeness in [33].
P5. Completeness of the Schema DML: given any pair (A,B) of consistent database states, there must be some valid update (of the DML), which maps A into B.

Second, we may identify some particular consistent state A as the "empty" or initial state. Then, we may require that:

P6. Reachability: given any consistent database state A, there must be some valid update that maps A into A.

We close this section with a few comments on the concepts introduced here. First, the notion of valid update amounts to forcing a program to satisfy certain pre- and post-conditions. Thus, the techniques developed for constructing correct software can be used here. However, Properties P4, P5, and P6 impose restrictions not only on programs, but on the DML itself, since it is possible to generate updates with certain characteristics. Hence, they may be very difficult to check, unless the DML was designed so that it trivially satisfies these properties [39].

2.3 - Schemas with Built-in Operations

The scheme DML is normally understood as the basic DML, that is, the DML of the data model or the DML supported by the DBMS. However, this approach makes it difficult to assure that every update is valid and, hence, it compromises the quality of data. To circumvent this problem, one may: (i) provide the schema (conceptual and external) with built-in operations to modify the database (written in the basic DML) that preserve consistency; (ii) force users to utilize the built-in operations when they modify the database. This strategy then obviously guarantees that all updates preserve consistency. By the restricted scheme DML, we mean the set of programs (of the basic DML) that modify the database only through the built-in operations.

In addition to the properties of Section 2.2, we may list certain other restrictions on a restricted scheme DML. Given a schema with built-in operations, we require:

P7. Consistency preservation: Each built-in operation is valid, i.e., preserves consistency.

P8. Operation independence: No built-in operation can be eliminated without affecting the set of all possible database state transitions.

P9. Operation applicability: For each built-in operation there is at least one consistent state where the operation can be successfully applied to modify the database.

We close with some observations about the concepts introduced here. First, although built-in operations are an excellent mechanism to assure that every update is valid, they may create some serious problems when one considers properties P4, P5, and P6. For example, it may not be clear if, given a group of built-in operations and a pair (A,B) of consistent database states, it is possible to write a program in the restricted DML that takes A to B. Second, some of the properties discussed in Sections 2.2 and 2.3 may not make sense when we consider external schemas with built-in operations, although they all apply in principle to conceptual and external schemas. For example, it is quite possible that the set of all consistent database states of an external schema E may be a subset of the consistent database states of the initial state by repeatedly applying built-in operations of E. This follows because, for security reasons, more consistent state A of E may not be reached by applying only updates available through E, but A can indeed be reached indirectly as a result of the combined update activity of other external schemas. Finally, we observe that sometimes external users may not be able to observe completely the effect of an update. Consider the case where an external user sees only information about employees whose salaries do not exceed a certain amount, but he is able to raise salaries beyond this limit. The effect of raising someone's salary beyond the limit will look to the external user as a delete operation, since the employee will disappear from his external schema. It is also interesting to note that if the external user had among his update operations a delete employee update, then deleting an employee or raising his salary beyond the limit would have the same external effect.

4.4 - The Interaction between the Conceptual and External Schemas

One important fact concerning external schemas is that an external schema is a virtual object with no independent existence. The data an external user wants to see is materialized from a conceptual database schema via a function which is part of the external schema definition. Likewise, each operation on an external schema is always translated into a program in the conceptual DML.

With this in mind, two other properties are of great importance:

P10. Translation correctness: the translation t of an operation u is correct in the sense that the result of u coincides with the external state constructed from the result of t.

This property is the minimum we require from t. Further requirements are discussed in [3, 8, 16, 21, 33, 35], which concentrate specifically on the view update problem.

P11. Consistency preservation: the translation of each operation maps the set of consistent database states into itself.

If we allow the translation of each operation to modify data in the database only through the database schema built-in operations then Property P7 implies Property P11.

Hence, this restriction should be adopted as much as possible since it greatly simplifies database design.

All other properties we consider in this section depend on the design strategy one has in mind. Suppose first that we adopt the position that the conceptual schema must contain a complete description of the application and that each external schema is just a window through which a group of users sees the database. This implies that each external schema does not contain any restriction that may not be deduced from those in the conceptual schema. Note that the presence of consistency criteria in the external schemas is still necessary so that each user gets an accurate description of the data he sees or manipulates.

More precisely, each external schema must also satisfy the following property, which was studied in [9, 36].

P12. Logical independence: Each external consistency criterion is a logical consequence of the conceptual consistency criteria in the sense that any external state constructed from a consistent database state is also consistent (with respect to the conceptual consistency criteria).

Therefore, if all external schemas satisfy P11 and P12, then no operation on one external schema can produce as a side-effect an inconsistent external state of another schema.

Failure to satisfy property P12 can have two interrelated consequences. First, an inconsistent external state may be generated from a consistent database state. Second, an update to an external schema may produce a side-effect an inconsistent external state of another external schema (this is really a consequence of the previous remark).

However, Property P12 may impose excessive restrictions on external schemas, so we may have to consider alternative strategies.
adopt a different design strategy. We may consider the conceptual schema as a
concurrency description of the application, incorporating just those properties that
are stable and meaningful to the users, such as legal restrictions or long-range,
global policies. Each external schema inherits these properties from the conceptual
schema, but it also reflects the peculiarities of a group of users and short-term
policies that are subject to change. Hence, Property P12 would be too strong in
this case.

This second design strategy requires a change of attitude towards the role of the
conceptual schema. Instead of considering its as the central concept, we take the
active schema, defined as the conceptual schema augmented with all consistency
criteria derived (in a precise sense to be described in Section 3.2) from the
definition of the external schema. When deriving the active schema it may be easier
to proceed iteratively, starting from the conceptual schema and then adding the ex-
ternal schemas one by one. Similarly, during the lifetime of a database, new
external schemas can be added or existing ones dropped. The separate study of a single
external schema together with the conceptual schema may also be useful to isolate
consistency characteristics of a group of users and global policies em-
bodied in the conceptual consistency criteria.

All concepts and properties developed previously for conceptual schemas also apply
to active schemas. In particular, a database state is now valid or consistent iff it
satisfies all constraints of the active schema; likewise, an update is now valid iff it.maps
the set of consistent database states (in the above sense) into itself.
Furthermore, the constraints of the active schema logically imply all
external consistency criteria. Hence, each consistent database state of the active
schema generates consistent external states and Property P12 is obtained by a fiat.

The two design alternatives compare as follows. The first design alternative
certainly facilitates the definition of external schemas since all restrictions of
the application are already embodied in the conceptual schema. It also simplifies
checking consistency preservation, as compared to the second design alternative,
which requires checking preservation of the potentially redundant criteria of
the active schema. On the other hand, a conceptual schema designed according to the
first strategy is less stable and, hence, more difficult to maintain. Or, in differ-
ent words, if we follow the second design alternative, moderate evolutions in
the application will impact just the external schemas and, hence, the active schema,
while keeping the conceptual schema stable. Evolvability is enhanced if the con-
ceptual schema DML has the completeness property (P5), since no built-in operation
of an external schema will be left untranslated by lack of flexibility of the
schema DML.

To achieve full evolvability one can change the consistency criteria of an external
schema E by just changing the corresponding active schema E again, or even the
constraints of some other external schema, or even the constraints of the con-
ceptual schema.

2.5 - The Interaction between External Schemas

It is worth investigating not only the interaction between an external schema and
the underlying conceptual schema, but also the interplay between external schemas
over the same conceptual schema. We already mentioned in Section 2.4 that, if
Properties P11 and P12 are met, then no operation of the conceptual schema can
produce as side-effect an inconsistent external state of another schema. But this
does not say that there is no side-effect. So, it might be of interest to detect
which, and how, operations of an external schema affect other external schemas.

Given two external schemas E and E', we may classify the effects of an operation of
E on E' as either an influence on queries or an influence on operations. By an

Influence on queries, we mean that data items visible through E' are changed as a
result of applying an operation of E; by an influence on operations, we mean that there is an
operation of E' whose result depends on whether or not E was applied.

Let us consider first the problem of detecting if an operation Ω of E influences
some query of E'. Note that Ω and E' have no direct relationship and, in fact, dif-
f erent languages are in principle used to describe Ω and E'. But both Ω and E' are
virtual objects in the sense that Ω is implemented in terms of a program τ at the
conceptual schema level and E' is likewise defined in terms of the conceptual
schema. Hence, the problem of checking if Ω influences some query of E' reduces to
testing if τ affects the definition of E' (since this definition can be thought as the
translation of a query that accesses all data in the external schema).

The problem of detecting if an operation Ω of E influences an operation τ of E'
is again treated by considering the translation of Ω and the translation of τ of Ω.
Note that both τ and τ are programs at the conceptual schema level, which
facilitates their comparison.

After detecting which influences exist, the next step would be to verify if some of
the detected influences are undesirable. But this step involves an independent
description of the set of allowed influences. Two bit matrices, Ω and Ω', can be
used for this purpose, where Ω has as many rows as there are built-in external
schema operations and as many columns as there are external schemas and Ω' = 1
indicates that the ith built-in operation can influence the jth schema; Ω has as
many rows and columns as there are built-in operations and Ω' = 1 indicates
that the ith operation can influence the jth operation.

The above discussion then leads to our last property.

P13. Influence consistency: no undesirable influences occur.

Naturally, if the database description in question does not satisfy Property P13,
some external schemas or even the conceptual schema would have to be redefined
and the design process iterated.

2.6 - Summary

Figure 2.1 lists all properties defined in this section. Not all these properties are
compulsory and, in fact, some of them make sense only if built-in operations are
used, whereas others reflect the design strategy adopted. Specifically,
properties P1 and P2 must be satisfied by all schemas; if the schema definition
includes built-in operations, then properties P7, P10, P11 and P13 must also be
satisfied. (Note that if the translation of each external built-in operation uses
only the conceptual schema built-in operations to modify the database, then P7
implies P11). Independently of the use of built-in operations, property P12 is
required if the conceptual schema completely describes the enterprise, but it does
not apply if the strategy of introducing certain constraints only through the exter-
nal schemas is adopted. Properties P14 and P12 are all desirable, but their
violations do not create invalid situations from the point of view of the
applications users. Property P6 is also desirable, especially when conceptual schema
are considered. Property P4 is compulsory in so far as the database design includes
update specifications. Finally, we observe that property P5 can be enforced only if
all transitions between valid states are valid, that is, if property P3 is vacuous
satisfied due to the absence of any dynamic consistency criterion.

The list of properties now follows.
Properties of Conceptual and External Database Schemas

3. SCHEMAS WITHOUT BUILT-IN OPERATIONS

In the rest of the paper, we formalize the properties listed in Section 2, but following an order dictated by the formalism used. We begin by treating in this section all properties that can be formalized within first-order logic.

3.1 - Special Many-Sorted Languages

Concepts pertaining to the relational model have been formalized in first-order logic by treating relations as predicate symbols [11, 13, 29]. However, this approach is not adequate when we have to consider functions from relations into relations [10], or when we have to quantify over variables ranging over relations, which is the case in this paper (see Section 4.2). Hence, we are forced to adopt a certain class of many-sorted first-order languages that emulates second-order languages [17 pp. 281, 10] to formalize the relational model.

We say that \( L \) is a special many-sorted (first-order) language if \( L \) is a many-sorted first-order language with sorts: the \( \text{individual sort} \), abbreviated \( \text{ind} \), with lowercase letters as variables and, for each \( n > 0 \), the \( n \)-place predicate sort, abbreviated \( \text{pred} \), with uppercase letters as variables (superscripted with \( n \) if necessary). We intend the \( n \)-pred domain to be a set of \( n \)-ary relations over individuals.

\( L \) must also include the following special parameters, listed with their intended interpretations:

1. the equality \( = \) of sort \( \text{ind} \), and, for each \( n > 0 \), the \( n \)-pred equality \( =_n \) of sort \( \text{pred} \), \( n \)-pred;

2. for each \( n > 0 \), the membership \( \in^\text{sort} \) of sort \( \text{pred} \), \( \text{ind} , \ldots , \text{ind} \). The intended interpretation of \( \in^\text{sort}(X^n, x_1, \ldots , x_n) \) is that the tuple denoted by \( (x_1, \ldots , x_n) \) is in the \( n \)-ary relation denoted by \( X^n \); hence, whenever possible, we abbreviate \( e^n(X^n, x_1, \ldots , x_n) \) as \( X^n(x_1, \ldots , x_n) \).

A general structure \( A \) is any structure of \( L \) such that:

1. \( A \) has intended domains and assigns to the special parameters their intended interpretations;
2. Let \( P \) be a well-formed formula of \( L \) and let the free variables of \( P \) be classified into two disjoint lists \( X = (x_1, \ldots , x_n) \) and \( Y = (y_1, \ldots , y_m) \), such that \( x_i \) has sort \( \text{ind} \) or \( \text{pred} \) and \( y_j \) has sort \( \text{ind} \) or \( \text{sort} \). Then, the following closed comprehension sentence is true in \( A \):

\[ \forall x_1 \ldots x_n \exists y_1 \ldots y_m \psi(x_1 \ldots x_n, y_1 \ldots y_m) \equiv P(x_1 \ldots x_n, y_1 \ldots y_m) \]

It is through the concept of general structure that we can truly say that \( L \) emulates a second-order language. This point is further discussed in [17 pp 281, 10]. We write \( P \in A \) to indicate that \( P \) is valid in a general structure \( A \) of \( L \) and \( P \in A \) to indicate that \( P \) is valid in any general structure (not necessarily in any structure) that satisfies all wffs in \( P \).

The second condition on general structures permits us to add new function symbols to \( L \) by definition. Let \( P(X,Y) \) be a well-formed formula in \( L \) with free variables \( X, Y \) (we follow the conventions adopted in the description of general structures). Then, a function symbol \( f \) of sort \( (x_1, \ldots , x_m, \text{pred}) \) can be added to \( L \) by definition with defining axiom

\[ f(x_1, \ldots , x_m) = \psi(x_1 \ldots x_n, y_1 \ldots y_m) \equiv P(x_1 \ldots x_n, y_1 \ldots y_m) \]

Note that, by definition of general structure, the uniqueness and existence conditions [40, p.59] for \( f \) are satisfied.

3.2 - Conceptual and External Schemas

Before defining what we mean by a schema, it is worth noting that we classify all symbols of a schema into two sets. The first set contains all symbols, such as \( \text{ind} \), whose intended interpretation is fixed. The second set includes all symbols whose meaning varies over time, which will be the relation names in the case of the relational model. Their meaning at a given point in time \( t \) comprises what is called the database state at \( t \) (however, for simplicity, our definition of database state also includes the meaning of all other symbols).

DEFINITION 3.1:

(a) A pair \( \sigma = (L, P) \) is a relational schema iff

1. \( L \) is a special many-sorted language with a distinguished set of constant \( r_1, \ldots , r_t \) of the \( k \)-pred sort (\( k \)-sort), the relation names of \( L \);
2. \( P \) is a set of wffs of \( L \), the consistency criteria of \( \sigma \);

(b) A database state of \( \sigma \) is a general structure of \( L \).
(c) A consistent database state of $\sigma$ is a database state $A$ of $\sigma$ such that $A^P$, for $P \in \mathcal{P}$.

(d) A database universe $U$ of $\sigma$ is a set of database states such that: (i) all database states differ only on the values of the relation names; (ii) for every $A \in U$, for every $n$-ary relation name $r$ and every $n$-ary relation $R$ (of the common $n$-pred domain), there is $B \in U$ such that $B = [R/xA]$.

Thus, we view a schema $\sigma = (L, P)$ as a first-order theory and a consistent database state $\sigma$ as a model of $\sigma$. The notion that the meaning of all symbols, except the relation names, is fixed is embodied in the definition of database universe.

We can now readily formalize the basic schema properties listed in Section 2. Let $\sigma = (L, P)$ be a schema.

1. consistency: there is at least one consistent database state of $\sigma$ or, equivalently, $P \not\models \bot$.

2. logical independence: $P \vdash \psi \equiv P \vdash \psi$, for each $P \in \mathcal{P}$.

Note: the formalization of Property P2 is deferred to Section 5.

An external schema $\pi$ of a schema $\sigma$ is just another schema whose language is basically the same as $\sigma$, but which may have its own relation names. As a convenience, we also require that no relation name of $\sigma$ is used in $\pi$. In addition, $\pi$ contains a function defining its relation names in terms of those of $\sigma$.

**Definition 3.2:**

Let $\sigma = (L, P)$ be a schema.

A triple $\pi = (M, \mathcal{Q}, J)$ is an external schema of $\sigma$ if and only if:

1. $\pi = (M, \mathcal{Q})$ is a schema such that all symbols of $M$, except the relation names, are also symbols of $L$ and no relation name of $L$ is a symbol of $M$.

2. $J$ is a function assigning to each $n$-ary relation name $r$ of $L$ a term $\beta^L_r$ of $L$ of the same sort as $r$.

We can extend $J$ to an interpretation of $M$ into $\sigma$ (17, pp 156) as follows: $J$ considers the domains of $M$ and $L$ to be identical; $J$ is the identity on each symbol of $M$, except the relation names; and $J$ assigns to each relation name $r$ of $M$ the wff $\beta^M_r = \beta^L_r$ (this is just to conform with the definition of interpretation).

The function $J$ can be used for two different purposes. First, given a wff $P$ of $M$, we can construct a wff $P^J$ of $L$ by replacing each relation name $r$ of $M$ by its definition $\beta^L_r$, as explained in (17, pp 160).

Second, given a general structure $A$ of $L$, we can construct a structure $A^J$ for $M$ as follows. The domain of $A^J$ are those of $A$; for each relation name $r$ of $M$, $A^J(r)$ is the relation $A(r^J)$ of $A^J$, where $A^J(r^J)$ agrees with $A$ on every other symbol. $A^J$ is called the structure of $M$ induced from $A$ by $J$.

We can relate the two constructions by the following lemma.

**Lemma 3.1:** For any structure $A$ of $L$ and any wff $P$ of $M$, $\vdash A^J P$ if and only if $\vdash A^J P$.

**Lemma 3.1** has several interesting consequences. We begin by showing that Lemma 3.1 implies that the induced structure of $M$ is indeed a general structure of $M$.

**Corollary 3.1:** If $A$ is a general structure of $L$, then $A^J$ is a general structure of $M$.

**Proof:**

Let $A$ be a general structure of $L$. Since $A$ has the intended domain and assigns to the special parameters their intended interpretations, so does $A^J$. Let $P$ be a closed comprehension sentence of $M$. Then, $P^J$ is a closed comprehension sentence of $L$ and, hence, true in $A$. By Lemma 3.1, $P$ is then true in $A^J$. Hence, $A^J$ also satisfies the second condition of general structures.

We now use Lemma 3.1 to formalize Property P12. Let $\pi = (M, \mathcal{Q}, J)$ be an external schema of $\sigma = (L, P)$, $\pi$ and $\sigma$ satisfy Property P12 iff, for any model $A$ of $P$, for any $Q \in \mathcal{Q}$, $A^J$ satisfies $Q$. But, by Lemma 3.1, $A^J$ satisfies $Q$ iff $A^J$ satisfies $Q^J$. Hence, Property P12 is equivalent to saying that $\pi^J$ is a logical consequence of $P$, which is a much more convenient characterization of Property P12.

**Property P12:** Logical dependence: $P \vdash Q^J$, for each $Q \in \mathcal{Q}$.

We now define what we mean by active schema. Given a conceptual schema $\sigma = (L, P)$ with a set of external schemas $\pi_1, \pi_2, \ldots, \pi_n$, we say that $\sigma = (L, P)$ is the corresponding active schema iff $P = \bigcup_{i=1}^{n} \pi_i^J$. As already discussed, this concept is important for the second design strategy mentioned in Section 2 since $\sigma$ incorporates all consistency criteria defined for the application, all the language of the conceptual scheme.

This concludes the list of properties that we formalize using first-order logic.

4. SCHEMES WITH BUILT-IN OPERATIONS

In this section, we discuss schemes with built-in operations. The formalism we use is a variant of Dynamic Logic (DL), quite similar to the one described in (10, 11). We first briefly describe the DL variant and then address scheme properties.

4.1 Regular Many-Sorted Dynamic Logic

Before defining the DL variant, we need a new concept. Let $L$ be a special many-sorted language with a set of distinguished constants, henceforth called program variables. A universe $U$ for $L$ is a set of general structures of $L$ satisfying two conditions:

(i) any two structures in $U$ differ only on the values of the program variables;

(ii) for any $A \in U$, any program variable $x$ and any element $e$ of the appropriate domain, there is $B \in U$ such that $B = [e/x]A$. These conditions guarantee that, for example, if $x$ is assigned $e$ as value, the resulting structure is in $U$. That is, the universe is closed under assignment, so to speak.

The programming language the DL variant uses is the set of regular programs over $L$ (RP), defined inductively as follows:

**Syntax:**

1. (1) for any program variable $x$ of $L$ and any term $t$ of $L$ such that $x$ and $t$ are of the same sort, $x^t$ is in RP and is called an assignment;

2. (2) for any wff $P$ of $L$, $P^t$ is in RP(L) and is called a test;

3. (3) for any $a, b \in$ RP(L), $a^b$ and $a^b$ are also in RP(L) and are called the union of $a$ and $b$, the composition of $a$ and $b$ and the iteration of $a$, respectively.

**Semantics:** for a fixed universe $U$ of $L$, the meaning of programs in RP(L) is given by a function $\mu$ assigning to each $a \in$ RP(L) a binary relation $\mu(a) \subseteq U^2$ as follows:
(4) \( m(x; t) = \{(a, B) : B \rightarrow A(t)/x\} A \)
(5) \( m(P) = \{(a, A) : A \rightarrow P\} \)
(6) \( m(ab) = m(a) \cup m(b) \) (union of both binary relations)
(7) \( m(ab) = m(a) \circ m(b) \) (composition of both binary relations)
(8) \( m(a) = m(a) \circ m(a) \) (reflexive and transitive closure of \( m(a) \))

The language DL of the DL variant (based on L) is defined as follows:

**syntax:** The syntax of DL is the same as that of L, with one additional formation rule:

(1) If P is a wff of L or DL and \( m(P) \) is a wff of DL (read "box of \( b, P\)").

**semantics:** For a fixed universe \( U \) of L, the notion of validity is extended to \( [b, P] \) as follows:

(2) \( [b, P] = \{b\} \) if \( b \) is valid in A iff \( b \) does not halt starting in A (that is, for no \( B \) in \( U \), \( A(B) \subseteq B \)) or, for any state \( B \) that can be reached from \( A \) via \( b \), \( P \) is valid in \( B \).

We also introduce by definition \( \neg P \) (read "diamond of \( b, P\)"), as \( \gamma(b, P) \). Hence, we have

(3) \( \neg P \) is valid in A iff \( b \) does not halt starting in A and \( b \) is valid in A.

The language of the DL variant permits us to express three properties of programs that are central to formalizing certain schema properties. Let \( U \) be a universe of L, \( P \) be a wff of L, and \( t_1, \ldots, t_n \) be terms of L. Let \( a \) and \( b \) be two programs in \( HL(U) \) and \( x_1, \ldots, x_n \) be variables of \( L \) not occurring in \( b \), where \( x_i \) is of the same sort as \( t_i, 1 \leq i \leq n \). Then, we say that:

(1) \( P \) is an invariant of \( b \) iff \( [b, P] = [b] P \).

(2) The values of \( t_1, \ldots, t_n \) are unaffected by \( b \) iff \( b \) is valid in A.

(3) Programs \( a \) and \( b \) are equivalent for \( t_1, \ldots, t_n \) iff \( a \) and \( b \) are equivalent for \( t_1, \ldots, t_n \) in \( U \) if \( \forall x_1 \ldots x_n : Q = \{b\} Q \), where \( Q = \bigwedge_{i=1}^{n} x_i = t_i \).

4.2 - Conceptual and External Schemas Revised

We are now ready to define schemas with built-in operations.

**DEFINITION 4.1:**

(a) A schema with built-in operations is a triple \( (L, P, \sigma) \) such that:

(1) \((L, P)\) is a schema such that \( L \) contains a distinguished set of constants called program variables, which include the relation names;

(2) \( \sigma \) is a set of programs in \( HP[L] \).

(b) An external schema with operations of \( \sigma \) is a four-tuple \( (M, Q, N) \) such that:

(1) \((M, Q, N)\) is a schema with operations such that all symbols of \( M \), except the relation names, are also symbols of \( M \) and no relation name of \( M \) is a symbol of \( N \).

(2) \( J \) is a function assigning to each \( n \)-ary relation name \( r \) of \( M \) term \( J \) of \( L \) of the same sort as \( r \) and to each operation \( a \) a program \( a^* \) of \( HP[L] \).

The notions of database state and consistent database state are as in Definition 3.1. However, we adopt the concept of universe defined in Section 4.1, rather than that of Definition 3.1. This follows because not only the relation names, but certain other constants (i.e., the other program variables) may change value from state to state.

We now discuss the role of the set \( \sigma \) of built-in operations of a schema \( \sigma \). Recall from Section 2.3 that we considered the DML of the set of regular programs that used only operations in \( \sigma \) to update the database. In the abstract data type jargon, this strategy is called "ancapulation".

More precisely, we define the set \( RP_{\sigma}[L] \) of restricted regular programs for a schema \( \sigma \) with a set of built-in operations as follows:

(1) If \( a \sigma \), then \( a \sigma L \).

(2) If \( b \) is an assignment of \( RP[L] \) such that the left-hand side is not a relation name of \( \sigma \), then \( b \sigma L \).

(3) If \( P \) is a wff of \( L \), then \( P \) is \( RL \).

(4) If \( a, b \sigma L \), then \( a, b \sigma L \).

The relevancy of ancapulation by means of built-in operations lies in the fact that we can prove that, if each built-in operation preserves consistency, then so does each program in \( RP[L] \). Therefore, users cannot violate the consistency of the database. To prove this result, we start with a basic lemma.

**LEMMA 4.1:** Let \( \sigma = (L, P, \sigma) \) be a schema with a set of \( \sigma = \{\sigma_1, \ldots, \sigma_n\} \) of built-in operations, and \( b \sigma L \). Construct the regular program \( p = (a_1 \ldots a_n o_1 \ldots o_m) \). Then, for any \( (A, B) \subseteq b \), there is \( (A, C) \subseteq p \) such that \( B \) and \( C \) are equal on the relation names of \( \sigma \).

**Proof:** Let \( b \sigma L \) and \( (A, B) \subseteq b \). Since \( b \) is a built-in operation in \( \sigma \) to modify the database, there is a sequence \( q = (a_1 \ldots a_n o_1 \ldots o_m) \) of operations in \( \sigma \), generated by the execution of \( b \) that takes \( A \) to \( B \), such that \( q \) takes \( A \) to \( C \) and \( B \) and \( C \) are equal on the relation names of \( \sigma \). But, by definition of \( p \), \( (a_1 \ldots a_n o_1 \ldots o_m) \) is also a possible execution of \( p \). Hence, \( (A, C) \subseteq p \). Therefore, we conclude that \( \sigma \) is consistent.

**COROLLARY 4.2:** Let \( \sigma = (L, P, \sigma) \) be a schema and assume that no program variable, other than the database relation names of \( \sigma \), occurs in wffs of \( P \). Then, if each operation in \( \sigma \) preserves consistency, then every program in \( RP[L] \) preserves consistency.

**Proof:** Let \( b \sigma L \). Let \( (A, B) \subseteq b \) and assume that \( A \) is consistent. By Lemma 4.1, there is \( C \subseteq D \) such that \( (A, C) \subseteq p \), where \( p = (a_1 \ldots a_n o_1 \ldots o_m) \), and \( B \) and \( C \) are equal on the relation names of \( \sigma \). But, by assumption, each \( \sigma_i, i = 1, \ldots, m \), preserves consistency. So, \( p \) also preserves consistency. Therefore, since \( A \) is consistent, \( C \) is also consistent. Thus, by the assumption on \( P \), \( B \) is also consistent. Hence, we conclude
that every \textbf{build}(L) preserves consistency. D

To summarize, in the case of encapsulation through built-in operations, users cannot violate consistency of the database since every possible DML program preserves consistency.

4.3. Formalization of Properties Related to Built-in Operations.

Within the framework developed here, we can completely formalize all properties related to built-in operations. Properties P7, P9, and P11 follow directly using the concepts introduced at the end of Section 4.1. Let \( S = (l, P, O) \) be a scheme and \( U \) be a universe for \( L \).

P7. Consistency preservation: for each be \( O \), for each PeP, P is an invariant of \( b \) (for \( U \)).

P9. Operation applicability: for each be \( O \), it is not the case that all relation names of \( O \) are unaffected by \( b \).

Let \( \sigma \) and \( U \) be as above and \( \pi = (M, O, J, N) \) be an external schema of \( \sigma \).

P11. Conceptual consistency preservation: for each be \( N \), for each PeP, \( P \) is an invariant of \( b^* \) (for \( U \)).

Consider now Property P8, operation independence. Which requires that no operation \( O \) is superfluous. To formalize Property P8, we recall from Lemma 4.1 that the set of all database transactions that can be performed by finite sequences of operations coincides with the set of database transitions that can be brought about by the regular program \( \mathcal{P} = (O, \mathcal{U}, \mathcal{G}, \mathcal{C}) \). Let \( \mathcal{P} = (O_1, \ldots, O_n \quad \mathcal{G}) \). Then, \( \mathcal{P} \) captures all database transitions that can be executed without the help of \( S \).

Therefore, we can concretely state Property P8 as follows:

P8. Operation independence: for any \( I, I_1, I_2 \), \( I_1 \) is not equivalent to \( P_1 \) for the relation names of \( S \).

Property P10, translation correctness, is not so easy to formalize, though, so let us then try to precisely define it first. Let \( \mathcal{S} = (M, O, J, N) \) be an external schema of \( \sigma = (l, P, O) \). Assume that \( r_1, \ldots, r_n \) are the relation names of \( \pi \). Let \( U \) be the universe of \( L \) in question. Then, Property P10 says the following. Let \( A \mathcal{C} U \). Apply \( b \) to \( A \), obtaining some state \( \mathcal{B} \mathcal{U} \). Now apply \( b \) to \( A \mathcal{B} \mathcal{C} \), obtaining \( C \). Then, \( A \mathcal{B} \mathcal{C} \) must agree on \( r_1, \ldots, r_n \) (informally \( b = c \)). Conversely, let \( A \mathcal{B} \mathcal{C} \), obtaining some state \( C \) of \( \pi \). Then, there must be some state \( A \mathcal{B} \mathcal{C} \) attainable by \( b \) from \( A \) such that \( C \) and \( b \) agree on \( r_1, \ldots, r_n \). This can be expressed concisely by the diagram in Figure 4.1.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4_1.png}
\caption{Figure 4.1}
\end{figure}

Now we observe that \( J \) can be emulated by a program \( q \) in the set of regular programs \( \mathcal{R}(\mathcal{L}) \) if we adopt the universe \( V \) of \( \mathcal{L} \) constructed from \( U \) by using \( J \) to extend each \( \mathcal{A}U \) to the relation names of \( \pi \). Let \( r_1, \ldots, r_m \) be the relation names of \( \pi \) and recall from Section 4.2 that \( r_1^J \) is a term of the same sort as \( r_1 \). Then, we have:

\[ q = r_1^J \Rightarrow \ldots \Rightarrow r_m^J \Rightarrow \ldots \]

Combining all these observations with the notion of program equivalence, P10 can be formalized as follows:

(10) Translation correctness: \( b^* \) correctly translates \( b \). If \( A \mathcal{B} \mathcal{C} \) are equivalent for \( r_1, \ldots , r_m \).

To investigate Property P13, influence consistency, we first have to formalize the notion of operation influence. Let \( \pi = (M, O, J, N) \) and \( \pi' = (M', O', J', N') \) be two external schemes of \( \sigma = (l, P, O) \). An operation \( b \) does not influence any query of \( r \) iff, given any two states \( A \) and \( B \) of \( \pi \) such that \( b \) maps \( A \) into \( B \), then \( A \) and \( B \) agree on the relation names \( r_1, \ldots, r_m \) of \( \pi \). Using the concepts at the end of Section 4.1, we then say that \( b \) does not influence any query of \( \mathcal{E} \). The diagram of Figure 4.2(a) may help understand this definition.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4_2.png}
\caption{Figure 4.2}
\end{figure}

Likewise, we say that \( a \) does not influence \( b \) iff \( a \) and \( b \) are equivalent for \( r_1, \ldots, r_m \) (c.f. the diagram in Figure 4.2(b)).

Now the formalization of Property P13 follows:

P13. Influence Consistency: all influences detected (using the definitions above) are allowed.

4.4. Properties Related to the DML

We now discuss P4, P5, and P6. All these properties are related to the existence of programs satisfying certain restrictions. But this cannot be expressed in (First-Order) DL since we cannot quantify over programs. For example, Property P4 would have to be expressed as follows:

\[ \]
P4. Adequacy of the Schema DNL: \[ \exists p (\exists q (\forall x (x \in q \rightarrow q(x)) \land \exists x (x \in p \land \neg q(x)))) \] for each update specification \( \langle y, 0 \rangle \), where \( U \) is the universe in question and \( F \) is the conjunction of all consistency criteria.

Note that we used a variable \( b \) ranging over the set of allowed programs. Hence, the \( \forall b \) above is phrased within a schema of First-Order Dynamic Logic.

However, when we consider a schema \( \gamma \) with a set \( \{a_1, \ldots, a_n\} \) of built-in operations, we can formalize Properties P2 and P6. We assume that we have no program variable, other than the relation names of \( \gamma \), occurs in any consistency criteria of \( \gamma \). We also need additional notation. If \( F \) is a consistency criteria of \( \gamma \), let \( FL(F) \) denote the \( FL \) obtained by replacing each relation name \( x_i \) by a variable \( x_i \) in the same database state as \( F \).

Consider now Property P5, which says that given any pair \((A, B)\) of consistent database states, there must be some valid update that maps \( A \) into \( B \). By Lemma 4, this is equivalent to requiring that \( p = \langle 0, A, \ldots, x, \rangle \) is a consistent database state in any other consistent database states.

P5: Consistency of the Schema DNL:
\[ \forall x \forall y \forall \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exist
THEOREM 5.1: For any $AQ$, for any AQD, $AQD \iff AQD$.

Proof:
It suffices to prove that
(1) $AQD \iff AQD$.  
(2) $AQD \iff AQD$.

since, using (1) and (2), we can prove by induction on the structure of AQD that

$AQD \iff AQD$.  

We first prove that
(3) $R \subseteq R$.

Let $p \equiv (u, v, \ldots, w, x)$, then $q(p) \equiv R$. But $p(q) \equiv (u, v, \ldots, w, x)$, by definition of $R$.

Now, by adapting the proof of Lemma 4.1, we can prove that

(4) $W(AQD) \subseteq (AQD \subseteq \subseteq AQD)$.

where $W(AQD)$ indicates that $A$ and $A'$ agree on the relation names of $A$.

Now, it is easy to prove that (3) implies that
(5) $AQD \subseteq AQD$.

(6) $AQD \subseteq AQD$ implies $AQD \subseteq AQD$.

and that (4), together with our assumption about $aq$ in $D$, implies the converse.
(5) and the converse of (6).

Theorem 5.1 has two important consequences. First, consider the task of designing built-in operations so that no sequence of programs that when executed modify the database violates the dynamic criteria. By Theorem 5.1, it is sufficient to design the operations so that all dynamic criteria are valid in $I \equiv (U, R')$, where $R' = (m(A), u, v, \ldots, w, x)$ and $R^* = (m(A), u, v, \ldots, w, x)$ are built-in operations. But since $I$ is defined in terms of sequences of built-in operations (and not sequences of programs), the second task is much simpler than the first. This was the reason for introducing $I$.

Second, by the proof of Theorem 5.1, we have $R \equiv (m(A), u, v, \ldots, w, x)$ is a regular program. Therefore, given $U \in (U, R')$, $U \subseteq U$ iff $U \subseteq U$, and $U \subseteq U$ iff $U \subseteq U$. But this implies that, for any $W(D)$ of $U$, we can find a system $W(D)$ such that $W(D) \subseteq W(D)$. Since, in addition, we have $U \subseteq U$, we can state all dynamic consistency criteria of a schema with built-in updates, $O_1, \ldots, O_n$ as $U \subseteq U$. (The underlying assumption that no program variables other than the relation names are used in dynamic consistency criteria should not be forgotten).

To summarize, dynamic consistency criteria can be discussed within the framework of Modal Logic. However, when schemes with built-in operations are considered, we have the option to use Dynamic Logic to express these constraints. Naturally, auxiliary data structures summarizing information about the past states of the databases (e.g., names of former employees) would probably be needed to construct the built-in operations.

Finally, we observe that the schema are sufficiently abstract to apply to most databases. For example, the schema for a two years after admission, any graduate student must have completed all credits, "an employee can only have his salary raised six months after being hired", and "at a specified date, a fine is added to the amount in debt, if the debt is not paid". Such sentences are quite naturally interpreted in the schema when one considers the optimal scheduling of tasks, which are capable of initiating action rather than merely acting in response to user requests.

Although these sentences are somewhat similar to dynamic consistency criteria; they are best discussed using formalisms such as the one described in (12, 92).

5. CONCLUSIONS

Taken together, the properties defined constitute our notion of correctness of database design, with respect to conceptual and external schemes.

We have found that, although there has been considerable research on the subject, it has been rather fragmentary. Moreover, often too much effort is spent on problems that are not relevant to properties that we consider fundamental. For example, most research on dependency theory would help only in some cases where the dependency is logically independent from the others. For example, we have seen that the properties we have listed:

By taking the point of view of one who is developing a database design methodology, we were aware of the need for breadth and comprehensiveness. In some cases the formal language used to characterize a property was part of a formal system that also provided tools to verify if a given database design satisfied the property. In other situations, we relied on the expression power of a natural language only to convey our ideas more precisely.

We used first-order logic to formalize the basic design concepts and properties, and adapted the language when we considered dynamic built-in operations. Strictly speaking, this is an essential move, since we can describe properties involving built-in operations in first-order logic (33). However, we believe that dynamic logic, which gives special attention to programs, provides a much more natural notation to describe these properties. Similar remarks apply to modal logic. In fact, we have shown that one of Section 5 that dynamic constraints can be formulated within dynamic logic when built-in operations are used. Again, this observation does not render modal logic superfluous because the latter provides a much more convenient notation to describe these constraints.

At a second point the importance of design strategies involving built-in operations is clear. In particular, it is much simpler to characterize how the various schema interact if, at both the conceptual and external levels, only built-in operations are considered and, in addition, the external operations are defined using the conceptual ones.

The concept of active schema, defined from the conceptual and external schemes, is a powerful tool to verify very useful. Consider, a design strategy where the conceptual schema contains only stable consistency criteria, whereas the external schemas contain criteria that describe less permanent restrictions. This strategy builds upon the description, that currently holds, of the role of enterprise is provided by the active schema.

Finally, we observe that, by listing basic design properties, we tried to map the area so that future research may proceed in an orderly fashion, attacking each property in turn. For example, tools may be developed to help synthesize built-in operations from non-procedural specifications, that take into account the consistency criteria. This is the area of translation of built-in operations of external schemas may be partly automated, perhaps requiring the intervention of some administrator to resolve translation ambiguities.
ACKNOWLEDGEMENT

This research was supported in part by FINEP and by CNPq grants 402090/80 and 102451/78.

REFERENCES


