A Formal System for Reasoning about Programs Accessing a Relational Database

MARCO A. CASANOVA
Pontificia Universidade Católica do Rio de Janeiro
and
PHILIP A. BERNSTEIN
Harvard University

A formal system for proving properties of programs accessing a database is introduced. Proving that a program preserves consistency of the database is one of the possible applications of the system. The formal system is a variant of dynamic logic and incorporates a data definition language (DDL) for describing relational databases and a data manipulation language (DML) whose programs access data in a database. The DDL is a many-sorted first-order language that accounts for data aggregations. The DML features a many-sorted assignment in place of the usual data manipulation statements, in addition to the normal programming language constructs.

Key Words and Phrases: relational databases, data definition languages, data manipulation languages, aggregation operators, transactions, synchronization, consistency preservation, serializability, program correctness, formal systems, dynamic logic, many-sorted first-order logic
CR Categories: 4.33, 5.21, 5.24

1. INTRODUCTION

A database contains data that model some aspects of the world. The description of a database consists of a set of data structure descriptions and a set of consistency criteria for data values. To say that the data values in a database satisfy the consistency criteria is to say that the data adequately model the world. As a consequence, users expect to observe consistent data and are required to submit updates that will preserve consistency. Such updates are called transactions [17]. In this paper we provide a logic for a data manipulation language

This research was supported in part by the Conselho Nacional de Pesquisas, Brazil, under Grant 1112.1248/76 and in part by the National Science Foundation under Grants MCS-77-05314 and MCS-79-07762.
Authors' addresses: M.A. Casanova, Departmento de Informática, Pontificia Universidade Católica do Rio de Janeiro, Rua Marquês de São Vincente, 225, 22.453, Rio de Janeiro, RJ, Brazil; P.A. Bernstein, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138.

(DML) permitting us to prove, among other properties, that a transaction indeed preserves consistency. Although the concept of transaction is widely used [4, 17, 24, 27, 31, 34], very little work has been done on DML logics [6, 19], and none accounts for the full use of aggregation operators [21].

In Section 2 we explain, in intuitive terms, the database and programming language concepts used throughout the paper. The section ends with the plan used in the rest of the paper to transform a given logic of programs into a DML logic.

Briefly, the plan goes as follows. Section 3 rigorously defines a language to describe relational databases [11] that allows the full use of aggregation operators. Section 4 defines a variation of the regular programs of [28] that fills the role of DML. Section 5 introduces a logic for the DML that is closely related to the regular first-order dynamic logic of [20]. Section 6 briefly discusses how the DML logic can be used to study concurrent transaction systems. Finally, the appendix contains examples of database descriptions, transactions, and derivations in the DML logic.

2. BACKGROUND

2.1 Database Concepts

In this section we give a brief and intuitive description of some database concepts in order to set the stage for the plan described in the following sections.

We loosely characterize a database as an abstraction of a real world enterprise, that is, of a set of objects forming a coherent whole. By a state of an enterprise we mean the instances of all objects at a given time. Objects and states are the real-world counterparts of two complementary technical concepts that dominate the discussion about databases: database schemata and database states. A *database schema* describes the enterprise via a set of data structures abstracting the objects of the enterprise and a set of consistency criteria capturing the logical interconnections between these objects (the format of the data structures may also capture some of the interconnections). A *database state* assigns values to the data structures and corresponds to a state of the enterprise. If a database state satisfies all consistency criteria and thus adequately represents a state of the enterprise, then the state is said to be *consistent*. We collect all possible states in a *database universe*. We may then define a database directly as a database schema and a database universe.

The language used to define databases is called a *data definition language* (DDL). A DDL specifies a notation for writing data structures and consistency criteria, defines what types of concrete objects can be values of data structures (and the other symbols used in consistency criteria), and gives a method for determining when a database state satisfies a consistency criterion. In short, a DDL introduces the tools to define the concepts given in the previous paragraph.

Data retrievals and changes to the database state are expressed in a special kind of programming language, called a *data manipulation language* (DML). Four types of DML statements are commonly identified: a *retrieve* statement extracts data from the current database state; an *update*, *insertion*, or *deletion* statement creates a new database state by modifying, inserting, or deleting data from the current database state, respectively. If the DML expresses all desirable data retrievals and state changes, then it is said to be *complete* [1, 8]. (Chandra and Harel [8] give a precise definition of what we mean by "desirable.")

We are especially interested in DDLs and DMLs that admit *aggregation operators*, such as SUM, MAX, or MIN, which map a subset of a database state into a single data item. For example, if SAL is the set of all salaries in a PAYROLL database, SUM(SAL) returns the summation of all salaries. Aggregation operators add a new dimension to the consistency criteria and DML statements that can be expressed. For example, the criterion SUM(SAL) $\leq$ 100K means that the total payroll cannot exceed 100K. Our interest in aggregation operators is further substantiated because they are supported by some currently existing database systems [7, 21].

A DML program expresses observable changes in the state of the enterprise only if it maps the set of consistent database states into itself, or *preserves consistency* of the database. Such DML programs are called *transactions*. Several tools that facilitate writing transactions exist. For example, the DML might be enhanced by high-level operations that always preserve consistency; the DML program might be modified at compile time so that it preserves consistency [33]; routines, called *triggers* or *demons*, may be invoked at execution time to ensure consistency preservation [16]. Note that the ability to check whether a DML program is a transaction or not, which we investigate in this paper, underlies any of these three approaches.

Example 2.1. To help fix the concepts introduced thus far, we describe in this example an overly simplified database, AIRLINE, of an airline reservation system and two transactions, RESERVE and CANCEL, which access it. We choose tables (or relations) as our data structures and use the language of set theory to write consistency criteria. The DML statements are self-explanatory.

The database schema of AIRLINE has only one table, FLIGHT. Each row (n, s) in FLIGHT represents a flight, where n is the flight number and s is the number of available seats. The schema imposes two consistency criteria: no two rows have the same flight number, and the number of available seats is always nonnegative.

```
database schema AIRLINE
  table FLIGHT with columns NUMBER, #SEATS
  consistency criteria $(\forall (n, s), (n', s') \in \text{FLIGHT})(n = n' \Rightarrow s = s')$
                       $(\forall (n, s) \in \text{FLIGHT})(s \geq 0)$
```

The database universe of AIRLINE consists of all finite binary relations over the natural numbers.

The RESERVE transaction reserves a seat in flight N, if a seat is available, and the CANCEL transaction cancels a seat reservation in flight M.

```
RESERVE (N):
  update (N, $s - 1$) where (N, $s$) $\in$ FLIGHT $\wedge$ $s \geq 1$

CANCEL (M):
  update (M, $s + 1$) where (M, $s$) $\in$ FLIGHT
```

Both transactions certainly preserve consistency of the database. □

2.2 Programming Logic Concepts

In this section we outline some basic concepts of programming logics and indicate their relevance to databases.

Traditionally, a formal system is divided into three components: a syntax for describing objects and properties of objects; a semantics defining what the concrete instances of the objects are and giving a meaning to sentences within the syntax; and a proof theory for reasoning about the objects and their properties. A programming logic is a formal system whose objects are programs.

To investigate transactions, we essentially need a formal system whose objects are programs accessing a database. Another useful formal system would be one whose language is a DDL, since its proof theory could be used to detect redundancies or contradictions in the consistency criteria of a database schema.

The conceptual complexity of a programming logic is directly linked to the richness of the underlying programming language, a fact that had a profound influence on the design of programming languages and the development of programming methodologies. The maxim was to retain only those constructs that had clear and manageable proof rules and to seek programming styles that facilitated correctness proofs. In general, the design of a proof theory almost always forces a healthy purification of the language and its semantics, a stage DDLs and DMLs have not yet reached.

Programming logics differ in their power to express properties of programs. Most of these logics concern themselves with input-output properties of programs, since these are central to the current programming methodologies. But one may also be interested, for example, in program equivalence or in the ongoing behavior of a program. The latter is critical to the investigation of programs that are not supposed to halt, such as operating systems.

Consistency preservation is an input-output property, since it asserts that programs must map the set of consistent database states into itself. In Section 6 we discuss another property of programs accessing databases which is a form of program equivalence.

Much effort has been devoted to mechanizing the proof theory of programming logics, aiming at relieving completely or in part the programmer's burden of verifying the program. The traditional questions of soundness and completeness have also been extensively studied. Soundness requires that every deduced fact is true, which is the minimum one would ask from a proof theory. Completeness requires that every true fact is deducible. The completeness of programming logics presents special problems [12] that were factored out in part by assuming complete subtheories for the data types, among other things. This permitted concentrating on the adequacy of proof rules for the program constructs. But even with such an assumption, it was shown that certain complex programming languages cannot possibly have complete logics [9].

Both proof theory mechanization and metatheoretic investigations are very significant to transaction verification. Since the number of consistency criteria of a database is likely to be large, transaction verification can be tedious and error prone. Therefore, a program verifier, coupled with heuristics trimming down the number of criteria that must be checked, is almost a necessity. Restrictions on the format of consistency criteria may have to be imposed to speed up the process. Soundness and completeness become important because the former means that we will not wrongly declare that a program is a transaction and the latter signifies that we will not fail to prove that a transaction preserves consistency.

2.3 Constructing a Data Manipulation Logic

In this section we describe a concrete strategy for constructing a programming logic for proving the correctness of transactions. We assume that we are given a family of programming logics with the following characteristics:

(1) the logics are based on the notion of a program as a set of operations acting on a (memory) state;
(2) each logic has an *assertion language* to talk about states, which is extended by new formation rules to express properties of programs, including input-output properties;
(3) the logics share a core programming language whose constructs include assignments and are supported by sound proof rules.

We argue that, under certain circumstances, the assertion language can act as a DDL and, with trivial changes, the programming language can become a DML.

More explicitly, the assertion language $\mathscr{L}$ must meet the requirements for DDLs given in Section 2.1. That is, $\mathscr{L}$ must provide a notation for writing database data structures and consistency criteria, define what concrete objects can be values of these structures, and give a method for determining when a database state satisfies a consistency criterion. In the jargon of programming languages, $\mathscr{L}$ must then have a *data type* appropriate for describing database data structures.

Our way of viewing the assertion language as the DDL tends to weight choices differently from past research. Traditionally, DDLs aimed at reflecting the real world, a goal that created room for a plethora of proposals and favored elaborate data structures. However, because the DDL now plays the role of an assertion language, it must be accompanied by a proof theory. Hence a language based on simple data structures with a clear proof theory should be preferred.

We now address the question of transforming a programming language into a DML. The key observation is simply that a program manipulating a database contains among its data structures those listed in the database schema, so that database states become part of the memory states. Then the database state can be changed by an assignment whose left-hand side belongs to the data structures of the database and whose right-hand side describes new data values. Thus updates, insertions, and deletions become just special forms of such assignments. Retrieves follow likewise, except that the left-hand side does not belong to a data structure of the database. Therefore, by assuring that we have assignments of the type chosen for the database data structures, say T, we obtain a DML. The left- and right-hand sides of such assignments can be taken as variables and expressions (or terms) of type T in the assertion language (and hence of what we consider to be the DDL). Thus if we are interested in using aggregation operators in consistency criteria and DML statements, they must be allowed to occur in the expressions of type T in the DDL.

Our method of constructing a DML trivializes the traditional approach of embedding a data sublanguage into a programming language. All four statements, retrieve, insert, delete, update, collapse into forms of special assignments. Although we do not elaborate on this point, the method also discards the completeness question posed in [1] if the core programming language contains some type of **while** loop (see [8]). Moreover, if the proof rule for assignments does not depend on the data types involved, no new proof rule is actually needed, which facilitates recycling program verifiers and metatheory results.

3. A FAMILY OF RELATIONAL DATA DEFINITION LANGUAGES

In this and the next two sections, we describe a programming logic for proving the correctness of transactions, following the strategy outlined in Section 2.3. We begin by defining a family of formal languages whose members meet three conditions:

(1) as DDLs, they provide the syntax and semantics of the data structures and consistency criteria describing a database;
(2) as assertion languages of a programming logic, they must be supported by a proof theory;
(3) as expression languages for the assignments accessing databases, they must have a class of terms adequate to use as right-hand sides of such assignments.

We express our interest in data aggregations by adding a fourth condition:

(4) the term-formation rules must include aggregation operators.

All these conditions reflect the discussion in Section 2.3.

We base our development on the relational model for databases [11], which assumes that data are organized as finite unordered tables or relations. The relational model appeals both to the layman, since tables are, after all, a common method of maintaining data, and to the specialist, since a relation is a simple and familiar mathematical concept. A table t is described by a *relation scheme* giving a name to t and its columns; relation schemes are then the basic data structures of the relational model. A relational database B is in turn described by a *relational database schema* S, consisting of a finite set of relation schemes and a set of consistency criteria. A *relational database state* of S is a set of tables conforming to the relation schemes of S. A database universe is a set of database states, as usual. An aggregation operator in the relational model is a mapping from relations to individuals.

First-order languages have been favored to play the role of relational DDLs [2, 6, 19, 25, 26, 35] since predicate symbols can be used to denote tables and well-formed formulas (wffs) can be used to denote consistency criteria. This choice meets our first three conditions, but not the fourth one. To support our last claim, we observe that there is no symbol in a (one-sorted) first-order language that can be interpreted as a mapping from relations to individuals. In view of this objection, we opted for a family of many-sorted first-order languages with a sort for the individuals and a sort for the n-ary relations, $n > 0$. A function symbol from the sort of n-ary relations to the individual sort can then denote a mapping from n-ary relations to individuals.

The only alternative language known to us that is still first order and yet is powerful enough to express data aggregations is the language of set theory. However, we believe that the language of set theory is in a sense too rich, since we want to talk about individuals and relations over individuals, and not arbitrary sets.

This section is organized as follows. Section 3.1 describes the basic notions of many-sorted languages. Section 3.2 introduces the family of many-sorted languages adopted. Section 3.3 discusses special theories for reasoning about relational databases. Finally, Section 3.4 redefines the basic concepts of the relational model.

Since all formal languages we discuss are first order, we omit this qualification from now on.

3.1 Many-Sorted Languages

In this section we summarize the essential concepts of many-sorted languages. The reader is referred to [15, Ch. 4.3] for a fuller discussion. Assume that we have a nonempty set M of sorts. The symbols of a many-sorted language $\mathscr{L}$ (with sorts from M) are as follows.

Logical Symbols

(1) Parentheses and the usual logical connectives: (,), $\neg$, $\wedge$.
(2) Variables: for each sort i, the symbols $x_1^i, x_2^i, \ldots$.
(3) Equality symbols: for each sort i, there may be the predicate symbol $=_i$ said to be of sort (i, i).

Parameters

(1) Quantifiers: for each sort i there is a universal quantifier symbol $\forall_i$.
(2) Predicate symbols: for each $n > 0$ and each n-tuple of sorts $(i_1, \ldots, i_n)$ there is a (possibly empty) set of n-place predicate symbols of sort $(i_1, \ldots, i_n)$.
(3) Constant symbols: for each sort i there is a (possibly empty) set of constant symbols of sort i.
(4) Function symbols: for each $n > 0$ and each $(n + 1)$ tuple of sorts $(i_1, \ldots, i_n, i_{n+1})$ there is a (possibly empty) set of n-place function symbols of sort $(i_1, \ldots, i_n, i_{n+1})$.

Terms and wffs are defined as in one-sorted first-order languages, except that sort compatibility must be respected when using quantifiers, predicate symbols, and function symbols.

A *many-sorted structure* A for $\mathscr{L}$ is a function from the set of parameters of $\mathscr{L}$ assigning

(1) to the quantifier symbol $\forall_i$, a nonempty set U_i, called the *domain* of A of sort i;
(2) to each predicate symbol p of sort $(i_1, \ldots, i_n)$, a relation

$$p_A \subseteq U_{i_1} \times \cdots \times U_{i_n};$$

(3) to each constant symbol c of sort i, an element c_A of U_i;
(4) to each function symbol f of sort $(i_1, \ldots, i_n, i_{n+1})$, a function

$$f_A : U_{i_1} \times \cdots \times U_{i_n} \to U_{i_{n+1}}.$$

Since there is no equality symbol across sorts, we may always assume that the domains of A are distinct.

A *state* I for $\mathscr{L}$ is a function from the set of parameters and variables of $\mathscr{L}$ such that I restricted to the parameters of $\mathscr{L}$ is a structure A of $\mathscr{L}$ and I assigns to each variable x of $\mathscr{L}$ of sort i an element of the domain of sort i. We also say that *I extends A*, and we continue to use l_I for the value of a parameter or variable l of $\mathscr{L}$ in state I.

The notion of wff P (a closed wff P) of $\mathscr{L}$ being *valid* (*true*) in a structure A is exactly as for one-sorted first-order languages. Let M be a set of variables of $\mathscr{L}$. If v maps each variable of sort i in M into an element of the domain U_i of A, $A \models P(v)$ indicates that P becomes valid in A when each free variable x of P is assigned the value $v(x)$. If I is a state, we simply write $I \models P$ where I gives the values of the variables of $\mathscr{L}$.

The logical axioms and rules for many-sorted first-order languages are those of first-order languages (see, e.g., [30]), again taking into account sorts.

A *many-sorted theory* $\mathscr{T}$ is a formal system such that the language of $\mathscr{T}$ is a many-sorted language $\mathscr{L}$ and the proof theory of $\mathscr{T}$ is an axiom system containing all logical axioms and rules plus a new set D of formulas of $\mathscr{L}$, the *nonlogical axioms* of $\mathscr{T}$. Thus $\mathscr{T}$ is fully specified by a pair $(\mathscr{L}, D)$. When a wff P of $\mathscr{L}$ is derivable from a set F of wffs of $\mathscr{L}$ using the axioms and rules of $\mathscr{T}$, we write $F \vdash_{\mathscr{T}} P$.

Let P be a formula, $\bar{x} = (x_1, \ldots, x_n)$ be a vector of variables, and $\bar{t} = (t_1, \ldots, t_n)$ be a vector of terms. As a final note, $P[\bar{x}]$ indicates that x_i occurs free in P, i in $[1, n]$, and $P[\bar{t}/\bar{x}]$ denotes the formula obtained by replacing each free occurrence of x_i in P by t_i, i in $[1, n]$.

3.2 Special Many-Sorted Languages

In this section we introduce a family of many-sorted languages, adapted in part from [15, Ch. 4.4]. Each member of the family serves well as a relational DDL, can be backed up by a proof theory, has a rich set of terms denoting relations, and can express data aggregations. Thus the family meets all of our four conditions. To fully capture the relational model, we introduce a class of special structures for these languages.

We say that $\mathscr{L}$ is a *special many-sorted language* iff $\mathscr{L}$ is a many-sorted language with sorts: the *individual sort*, abbreviated ind, with lowercase letters as variables and, for each $n > 0$, the *n-place predicate sort*, abbreviated n-pred, with uppercase letters as variables (superscript with n if necessary). We intend the individual domain to be the natural numbers N and the n-pred domain to be the set of all finite n-ary relations over N.

$\mathscr{L}$ must also include the following special parameters, listed with their intended interpretations.

Predicate Symbols

(1) The *equality* (denoted by =) and the *ordering* (denoted by $\leq$) of sort (ind, ind) is intended to denote the corresponding relations over N.

(2) For each $n > 0$ there exists the *membership* e^n of sort (n-pred, ind, ..., ind). The intended interpretation of $e^n(X^n, x_1, \ldots, x_n)$ 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 $X^n(x_1, \ldots, x_n)$.

Function Symbols

(3) The *successor* S of sort (ind, ind) and the *addition* (denoted by +) of sort (ind, ind, ind) are intended to denote the corresponding operations on N.

(4) For each $n > 0$ and $m > 0$ there exists the *union* $\cup^n$, *intersection* $\cap^n$, *difference* $-^n$ of sort (n-pred, n-pred, n-pred), and *Cartesian product* ${}^n\times^m$ of sort (n-pred, m-pred, $(n+m)$-pred). The union $\cup^n$ is intended to denote the union operation over n-ary relations, and similarly for the other operations.

(5) Let P be a tight wff of $\mathscr{L}$ (defined in Section 3.3) and let the free variables of P be classified into two disjoint lists $\bar{x} = (x_1, \ldots, x_m)$ and $\bar{y} = (y_1, \ldots, y_n)$, $m \geq 0$ and $n \geq 0$, such that x_j has sort $i_j \in \{\text{ind}\} \cup \{k\text{-pred}/k \in N\}$ and y_l has sort ind, $1 \leq j \leq m$ and $1 \leq l \leq n$. Then $\mathscr{L}$ contains the function symbol $f_{P[\bar{x},\bar{y}]}$ of sort ($i_1, \ldots, i_m$, n-pred). By analogy with set theory, we write $f_{P[\bar{x},\bar{y}]}$ as $\{\bar{y}/P[\bar{x}, \bar{y}]\}$ and f_{false} as $\emptyset$. $f_{P[\bar{x},\bar{y}]}$ is intended to denote a function mapping a tuple of elements $\bar{a} = (a_1, \ldots, a_m)$, a_j from the domain of sort i_j, into the n-ary relation defined by $P[\bar{a}/\bar{x}, \bar{y}]$. Note that each tight wff P may define several functions, depending on how $\bar{x}$ and $\bar{y}$ are formed.

(6) For each $n > 0$ and $1 \leq i \leq n$ there exists the *maximum* MAX_i^n, *minimum* MIN_i^n, *sum* SUM_i^n, and *cardinality* COUNT_i^n of sort (n-pred, ind). SUM_i^n is intended to denote an aggregation operator mapping an n-ary relation into the sum of all entries in the ith column of the relation, and similarly for the other operators.

Constants

(7) 0 of sort ind is intended to denote the natural number zero. We abbreviate $S(0)$, $S(S(0))$, ... as 1, 2, ...

We stress that we introduced above a family of many-sorted languages. Each member of the family is obtained by taking the language of $(N, 0, S, \leq, +)$ (Presburger arithmetic) and adding the k-pred sort, $k > 0$, set-theoretic and aggregation operations, and other function and predicate symbols depending on the application in question. However, no significance should be assigned to our choice of Presburger arithmetic here other than that it provides the minimum set of functions we need to discuss SUM and COUNT. We require that our languages contain multiplication only in Section 5.2 to obtain a completeness result. Finally, we note that if other types of individuals, besides the natural numbers, are necessary, then the individual sort can be split into several sorts, thus creating other families of languages.

A *special structure* of $\mathscr{L}$ is any structure of $\mathscr{L}$ with the standard domains and assigns to the special parameters of $\mathscr{L}$ their intended interpretations.

A special many-sorted language $\mathscr{L}$ serves as a relational DDL since a constant or a variable of the n-pred sort may denote a table and a wff may describe a consistency criterion. $\mathscr{L}$ has a rich set of terms denoting relations because we included in $\mathscr{L}$ the usual set operations and a restricted set formation rule, formalized as the set of function symbol $f_{P[\bar{x},\bar{y}]}$ described in (5). The set of terms is further enriched by some of the usual aggregation operators; the nature of $\mathscr{L}$ forced the adoption of a different function symbol, e.g., SUM_i^n, for each arity n and column number i. Hence $\mathscr{L}$ can be used as an expression language for assignments to relations and accounts for aggregation operators. A proof theory for $\mathscr{L}$ is discussed in Section 3.3.

As an example of the expressive power of $\mathscr{L}$, we translate the English sentence "the highest payroll p of any department" into $\mathscr{L}$. We assume that $\mathscr{L}$ has a 3-pred variable *EMP* and 1-pred constants *NAME, DEPT, SAL* representing a table *EMP*[*NAME, DEPT, SAL*] and that the various symbols of $\mathscr{L}$ receive their intended interpretations. Then the English sentence becomes

$$p = \mathrm{MAX}_2^2\ (\{(d, t)/\exists n \exists s EMP(n, d, s) \wedge t = \mathrm{SUM}_2^2\ (\{(n, s)/EMP(n, d, s)\})\}).$$

Observe that aggregation operators appear nested inside one another. Thus $\mathscr{L}$ is as expressive as the full QUEL DDL [21] or the SEQUEL-2 DDL [7]. However, unlike these two languages, $\mathscr{L}$ is a first-order language.

3.3 Special Many-Sorted Theories

In this section we clarify the intended interpretation of the special parameters of special many-sorted languages and provide a basis for reasoning about them. More precisely, let $\mathscr{L}$ be a special many-sorted language with no other parameters except the special ones, and let A be a special structure for $\mathscr{L}$. Note that A is unique by definition of special structure and assumption on $\mathscr{L}$. We introduce a many-sorted theory $\mathscr{T}$ whose language is $\mathscr{L}$ and whose nonlogical axioms capture the intended interpretation of the special parameters of $\mathscr{L}$, in the sense that A is a model of $\mathscr{T}$. Hence any theorem of $\mathscr{T}$ is valid in A. The converse question, "Is any formula valid in A a theorem of $\mathscr{T}$?", is much more difficult and is not discussed here.

If the language chosen has other parameters besides the special ones, $\mathscr{T}$ must be augmented with further nonlogical axioms for reasoning about them. We call any such theory a *special many-sorted theory*.

The nonlogical axioms of $\mathscr{T}$ are as follows.

Arithmetical Axioms. An adequate set PA of axioms for Presburger arithmetic constitute the arithmetical axioms.

Finite Relation Axiom

(1) $\forall X^n \exists k(\mathrm{COUNT}^n(X^n) = k)$.

Set Operations Axioms. For each $n > 0$, $m > 0$, and tight wff $P[\bar{x}, \bar{y}]$ of $\mathscr{L}$,

(2) $X \cup^n Y = Z \equiv \forall \bar{x}(Z(\bar{x}) \equiv X(\bar{x}) \vee Y(\bar{x}))$,

(3) $X \cap^n Y = Z \equiv \forall \bar{x}(Z(\bar{x}) \equiv X(\bar{x}) \wedge Y(\bar{x}))$,

(4) $X -^n = Z \equiv \forall \bar{x}(Z(\bar{x}) \equiv X(\bar{x}) \wedge \neg Y(\bar{x}))$,

(5) $X^n \times^m Y = Z \equiv \forall\bar{x}\forall\bar{y}(Z(\bar{x}, \bar{y}) \equiv X(\bar{x}) \wedge Y(\bar{y}))$,

(6) $f_{P[\bar{x}\bar{y}]}(\bar{x}) = X^n \equiv \forall\bar{y}(X^n(\bar{y}) \equiv P[\bar{x}, \bar{y}])$.

Aggregation Operations Axioms. For each $n > 0$ and $1 \le i \le n$,

(7) $\forall x\forall y(\mathrm{MAX}_i^n(X) = y \equiv (\neg\exists\bar{x}X(\bar{x}) \wedge y = 0)$
$\qquad\vee (\exists\bar{x}(X(\bar{x}) \wedge x_i = y) \wedge \forall\bar{x}(X(\bar{x}) \Rightarrow x_i \le y)))$,

(8) $\forall x\forall y(\mathrm{MIN}_i^n(x) = y \equiv (\neg\exists\bar{x}X(\bar{x}) \wedge y = 0)$
$\qquad\vee (\exists\bar{x}(X(\bar{x}) \wedge x_i = y) \wedge \forall\bar{x}(X(\bar{x}) \Rightarrow y \le x_i)))$,

(9) $\forall X(\neg\exists\bar{x}X(\bar{x}) \Rightarrow \mathrm{COUNT}^n(X) = 0)$,

(10) $\forall X\forall\bar{x}(\neg X(\bar{x}) \Rightarrow \mathrm{COUNT}^n(X \cup \{\bar{x}\}) = \mathrm{COUNT}^n(X) + 1)$,

(11) $\forall X(\neg\exists\bar{x}X(\bar{x}) \Rightarrow \mathrm{SUM}_i^n(X) = 0)$,

(12) $\forall X\forall\bar{x}(\neg X(\bar{x}) \Rightarrow \mathrm{SUM}_i^n(X \cup \{\bar{x}\}) = \mathrm{SUM}_i^n(X) + x_i)$,

where, we recall, $X^n(\bar{x})$ abbreviates $e^n(X^n, \bar{x})$, etc.

The special structure A of $\mathscr{L}$ is a model of $\mathscr{T}$ since all nonlogical axioms of $\mathscr{T}$ are clearly valid in A. The only possible doubt concerns Axiom (6), since it involves the notion of a tight formula, which we now discuss. Suppose that we allow any wff P of $\mathscr{L}$ to be used in Axiom (6). Consider, for example, the two instances of Axiom (6) below (where F and G are 1-pred constants, X and Y are 1-pred variables, and R and S are 2-pred constants):

(a) $F = X \equiv \forall t(X(t) \equiv \exists u\exists v(R(u, v) \wedge S(v, t)))$,

(b) $G = Y \equiv \forall t(Y(t) \equiv \exists u\exists v(R(u, v) \wedge t = t))$.

Then, since R_A and S_A are finite relations, by construction of A, so will be F_A. However, the value of G in (b) cannot be a finite relation and hence not an element of the 1-pred domain of A. Hence (b) cannot possibly be true in A. Therefore, asking that A has only finite n-ary relations in the n-pred domain, $n > 0$, and that all instances of Axiom (6) for arbitrary wffs P be true in A cannot both be satisfied.

We were then forced to restrict P in Axiom (6) to formulas such as $\exists u\exists v(R(u, v) \wedge S(v, t))$ that assert the existence of further finite relations in the n-pred domain of A, $n > 0$. We call such formulas *tight*.

To understand the definition of tight formula, it helps to visualize the construction of F_A as follows. Construct first $C = R_A \times S_A$. For each tuple (t, u, v, x) in C, x is in F_A if $u = v$, by (a). Note that only tuples in C need be examined to construct F_A. The definition of tight formula explores exactly this property, following a suggestion in [11].

Definition 3.1. Let $\mathscr{L}$ be a special many-sorted language.

(i) Let Q be a conjunction in $\mathscr{L}$ of the form

$$\bigwedge_{i=1}^{n} Q_i(t_{i1}, \cdots, t_{ik_i}).$$

An individual variable x is *tied* in Q iff

(a) for some t_{ij}, x is t_{ij} and Q_i is a variable or a constant of the k_i-place predicate sort, or
(b) there is a clause in Q of the form $x = t$, t a term, and all variables occurring in t are tied in Q.

(ii) Let P be a wff of $\mathscr{L}$ and let P' be the disjunctive normal form of P. An individual variable is *tied* in P iff x is tied in Q for any disjunct Q of P' where x occurs (free or not).

(iii) A *tight formula* P of $\mathscr{L}$ is a wff of $\mathscr{L}$ such that all individual variables occurring in P are tied in P. □

We now briefly discuss how the models of $\mathscr{T}$ may differ from the special structure A. First, an adequate set PA of axioms for Presburger arithmetic forces the individual domain to have a standard part isomorphic to the natural numbers and a set of so-called Z-chains (see, e.g., [15, Ch. 3]). Second, e^n need not be interpreted as membership at all and the n-pred domain need not be the set of all n-ary relations over individuals. However, if a model M of $\mathscr{T}$ has the same domains as A and assigns to e^n, $n > 0$, the same interpretation as A, then the nonlogical axioms of $\mathscr{T}$ force M to agree with A on the rest of the parameters of $\mathscr{L}$.

The second source of dissatisfaction can be eliminated, in a certain precise sense, by adapting a result from [15, p. 283]. Before stating the result, we observe that we can always assume that the domains of any structure of $\mathscr{L}$ are disjoint, since $\mathscr{L}$ contains no equality between sorts. It also simplifies the discussion to eliminate all special parameters of $\mathscr{L}$, except the arithmetical ones, COUNT^n and SUM$_i^n$. Except for these parameters, all others can be considered as defined symbols of $\mathscr{T}$ (with Axioms (2)–(8) acting as defining axioms).

THEOREM 3.1 *Let M be a model of $\mathscr{T}$ such that the different domains of M are distinct. Then there is a homomorphism h of M onto a model N of $\mathscr{T}$ such that*

(i) h is one-to-one, in fact the identity, on the individual domain (from which it follows that $M \models P(v)$ iff $N \models P(h \circ v)$, for any assignment v of values to the variables of $\mathscr{L}$);

(ii) the n-place predicate domain of N consists of certain n-ary relations over the individual domain, and $(R, a_1, \ldots, a_n) \in e_N^n$ iff $(a_1, \ldots, a_n) \in R$.

PROOF. Since the domains of M are disjoint, we can define h on one domain at a time. On the individual domain D, h is the identity. On n-place predicate domain,

$$h(Q) = \{(a_1, \ldots, a_n) \in D^n/(Q, a_1, \ldots, a_n) \in e_M^n\}.$$

Thus we have

$$(a_1, \ldots, a_n) \in h(Q) \qquad \text{iff} \quad (Q, a_1, \ldots, a_n) \in e_M^n. \tag{1}$$

As e_N^n we simply take the membership relation

$$(R, a_1, \ldots, a_n) \in e_N^n \qquad \text{iff} \quad (a_1, \ldots, a_n) \in R. \tag{2}$$

On the other symbols, N agrees with M. This completes the construction of h and N. Since M is a model of $\mathcal{T}$, by construction, so is N. We now prove that h is a homomorphism from M onto N, that is, for each p-ary predicate symbol P of $\mathcal{L}$ and each q-ary function symbol f of $\mathcal{L}$, $p \geq 0$ and $q \geq 0$, we have

$$(a_1, \ldots, a_p) \in P_M \quad \text{iff} \quad (h(a_1), \ldots, h(a_p)) \in P_N \tag{3}$$

$$h(f_M(a_1, \ldots, a_q)) = f_N(h(a_1), \ldots, h(a_q)). \tag{4}$$

If P and f involve only the individual sort, then there is nothing to prove since h is the identity on the individual domain. Hence we immediately obtain that $=$ and $\leq$ satisfy (3), and S, $+$, and 0 satisfy (4). By (1) and (2), e^n satisfies (3) for each $n > 0$. So we are left with SUM^n and COUNT^n, $n > 0$. We only prove that COUNT^n satisfies (4) (the proof for SUM$_i^n$ is entirely similar). Since M and N have the same individual domain, h is the identity on the individual domain and COUNT$_M^n$ and COUNT$_N^n$ are functions from the n-pred domain into the individual domain, we only have to prove that

$$\text{COUNT}_M^n(R) = \text{COUNT}_N^n(h(R)). \tag{5}$$

We prove (5) by induction on the cardinality of $h(R)$.

Basis. Suppose $h(R) = \varnothing$. Then by (1) we have

$$\neg\exists\bar{a}(e_M^n(R, \bar{a})). \tag{6}$$

Since M and N are models of $\mathcal{T}$, the aggregation axioms are valid in M and N. Then, from (6), we obtain

$$\text{COUNT}_M^n(R) = 0_M \tag{7}$$

and, since $h(R) = \varnothing$, we have

$$\text{COUNT}_N^n(h(R)) = 0_N. \tag{8}$$

Now, by construction of M, $0_M = 0_N$, which implies that

$$\text{COUNT}_M^n(R) = \text{COUNT}_N^n(h(R)). \tag{9}$$

Induction step. Follows similarly.

Finally, the parenthetical remark of (i) follows from the fact that we have equality only for the individual sort, where h is one to one, by analogy with the homomorphism theorem for one-sorted languages [15, p. 91]. □

Theorem 3.1 tells us that, given any model M of $\mathcal{T}$, we can replace M by another model N of $\mathcal{T}$ whose n-pred domain is a set of n-ary relations over individuals and which assigns to e^n the intended interpretation, for each $n > 0$. Moreover, deciding whether a wff P of $\mathcal{L}$ is valid in M can be replaced by the identical problem in N. Hence without loss of generality we can ignore all models that do not have the appropriate n-pred domain and do not assign to e^n the intended interpretation $n > 0$.

3.4 Relational Databases

We now give precision to some relational model concepts within the framework developed. Let $\mathcal{T} = (\mathcal{L}, D)$ be a fixed special many-sorted theory.

Definition 3.2. A triple $\sigma = (\rho, \alpha, \delta)$ is a *relational schema* iff

(i) $\rho = \{R_1, \ldots, R_m\}$ is a set of distinct variables of $\mathcal{L}$, the *database relation names* of σ, where R_i is of the k_i-pred sort, i in $[1, m]$;

(ii) $\alpha = \{A_1, \ldots, A_n\}$ is a set of distinct constants of $\mathcal{L}$, the *attribute names* of σ, all of the 1-pred sort;

(iii) δ is a set of wffs of $\mathcal{L}$ containing the following.

Relation schemes. For each R_i in ρ of the k_i-pred sort, a formula

$$\forall \bar{x}(R_i(\bar{x}) \Rightarrow \bigwedge_{p=1}^{k_i} A_{j_p}(x_p))$$

where $j_1, \ldots, j_{k_i}$, $1 \leq j_1, \ldots, j_{k_i} \leq n$, depend on i.

Consistency criteria. Other wffs. □

An example of a relational schema appears in Section A1 of the appendix.

Let A be a structure for $\mathcal{L}$ and let M be a set of variables of $\mathcal{L}$. Recall from Section 3.1 that v is a valuation of M in A, if v is a function assigning to each variable x in M, x of sort ind or k-pred, an element of the individual or the k-pred domain of A. If v is a valuation for M, $A \models P(v)$ indicates that P becomes valid in A when each free variable x of P in M is valuated as $v(x)$.

Definition 3.3. Let $\sigma = (\rho, \alpha, \delta)$ be a relation schema and let A be a special structure of $\mathcal{L}$. A *database state* of σ for A is a valuation of ρ in A. A *consistent database state* of σ for A is a database state v of σ for A such that $A \models P(v)$, for each P in δ. The *database universe* of σ for A is the set of all database states of σ for A. □

Therefore we assign meaning to a database schema σ by fixing a structure A for $\mathcal{L}$ and constructing the database universe of σ for A. As a consequence, the value of an attribute name of σ is fixed by A, since attribute names are constants of $\mathcal{L}$. By contrast, the value of a database relation name is not fixed by A, but rather by each database state, since database relation names are treated as variables of $\mathcal{L}$. This agrees with the fact that relations are updated during the lifetime of a database, but attributes are not.

This treatment has a consequence we wish to emphasize. Let P be a wff of $\mathcal{L}$ with free variables that also play the role of database relation names, let A be a structure for $\mathcal{L}$, and let v be a database state. Then asking whether P is true in A is not equivalent to asking whether $P(v)$ is true in A. For example, consider P defined as (where *EMP* is a 3-pred variable),

$$\mathrm{SUM}_2^2\ (\{(n, s)/e^3(EMP, n, s, D)\}) < 100\mathrm{K}.$$

Note that *EMP* is the only free variable of P. Then P being valid in A means that the payroll of department D is less than 100K, for any value of *EMP*. However,

$P(v)$ being valid in A, for a given database state v, means that the payroll of department D is less than 100K for the value of EMP given by state v.

With these definitions, we can formalize the following.

(1) P in δ is redundant: $\delta' \vdash_{\mathscr{T}} P$, where δ' is δ without P.
(2) δ is inconsistent: $\delta \vdash_{\mathscr{T}}$ false.
(3) A wff P of $\mathscr{L}$ describes a property of every consistent database state of σ: $\delta \vdash_{\mathscr{T}} P$.
(4) The ith domain of the n-ary relation denoted by R, $1 \le i \le n$, is functionally dependent on the $j_1, \ldots, j_k$ domains, $1 \le j_1, \ldots, j_k \le n$,

$$\forall \bar{x} \forall \bar{y} (R(\bar{x}) \wedge R(\bar{y}) \wedge \bigwedge_{m=1}^{k} x_{j_m} = y_{j_m} \Rightarrow x_i = y_i)$$

where $\bar{x} = (x_1, \ldots, x_n)$ and $\bar{y} = (y_1, \ldots, y_n)$.

4. A FAMILY OF RELATIONAL DATA MANIPULATION LANGUAGES

Given a special many-sorted language $\mathscr{L}$ chosen to act as a DDL, we define a DML appropriate for accessing databases described in $\mathscr{L}$. The DML is constructed from the regular programs of [28], plus a new basic statement taken from [6]. We have chosen regular programs because they are backed up by a programming logic, called regular first-order dynamic logic (abbreviated FDL), as discussed in Section 5.

The DML programs denote binary relations between states of $\mathscr{L}$ (cf. Section 3.1) drawn from a certain universe U. A *universe* U of $\mathscr{L}$ for a special structure A of $\mathscr{L}$ is the set of all states of $\mathscr{L}$ extending A (cf. Section 3.1); A is then the structure *generating* U.

More precisely, given $\mathscr{L}$ and U, we define the class of *many-sorted regular programs* MSRP[$\mathscr{L}$, U] as follows.

Syntax

Statements

(1) If $\bar{x}$ and E are a variable and a term of $\mathscr{L}$, respectively, of sort i, then $x := E$ is in MSRP[$\mathscr{L}$, U] and is called an *assignment of sort i*.
(2) If P is a quantifier-free wff of $\mathscr{L}$, then $P?$ is in MSRP[$\mathscr{L}$, U] and is called a *test*.
(3) If $\bar{x} = (x_1, \ldots, x_n)$ are distinct individual variables of $\mathscr{L}$ and E is a term of $\mathscr{L}$ of the n-pred sort, $n > 0$, then $\bar{x} \leftarrow ?\ E$ is in MSRP[$\mathscr{L}$, U] and is called a random tuple selection.

Formation rules

(4) if s and t are in MSRP[$\mathscr{L}$, U], then $(s \cup t)$, $(s;t)$, and s^* are also in MSRP[$\mathscr{L}$, U] and are called the *union* of s and t, the *composition* of s and t, and the *iteration* of s, respectively.

Semantics. The *meaning* of programs in MSRP[$\mathscr{L}$, U] is given by a function m: MSRP[$\mathscr{L}$, U] $\rightarrow 2^{U^2}$ defined as

(5) $m(x := E) \quad = \{(I, J) \in U^2 / J = [E_I/x]I\}$,

(6) $m(P?) \quad = \{(I, I) \in U^2 / I \models P\}$,

(7) $m(\bar{x} \leftarrow ?\, E) = \{(I, J) \in U^2/(\exists\, \bar{a} \in E_I)(J = [\bar{a}/\bar{x}]I)\}$,

(8) $m(s \cup t) \quad = m(s) \cup m(t)$ (union of both binary relations),

(9) $m(s;t) \quad = m(s) \circ m(t)$ (composition of both binary relations),

(10) $m(s^*) \quad = (m(s))^*$ (reflexive and transitive closure of $m(s)$),

where, following [20], we use $[\bar{v}/\bar{s}]I$ to denote the state J of $\mathscr{L}$ differing from I at most on the value of a tuple of symbols $\bar{s}$ of $\mathscr{L}$ (or a single symbol), which is $\bar{v}$ in J.

Examples of assignments are

(11) $x := x + 1$,

(12) $EMP := EMP \cup \{(\text{KENNEDY, WHITEHOUSE, 100K})\}$,

(13) $d := \text{MAX}_2^2\ (\{(d, t)/\exists n \exists s\, EMP(n, d, s)$

$\wedge\, t = \text{SUM}_2^2\ (\{(s, n)/EMP(n, d, s)\})\})$.

Assignments of the n-place predicate sort are then appropriate for manipulating the database relations or, more generally, for constructing new relations out of old ones. The usual relational operations [7, 10, 21] are defined as follows:

(14) **retrieve** $R(\bar{x})$ **where** $P[\bar{x}] \quad = R := \{\bar{x}/P[\bar{x}]\}$,

(15) **insert** $R(\bar{x})$ **where** $P[\bar{x}] \quad = R := R \cup \{\bar{x}/P[\bar{x}]\}$,

(16) **delete** $R(\bar{x})$ **where** $P[\bar{x}] \quad = R := R - \{\bar{x}/P[\bar{x}]\}$,

(17) **update** $R(\bar{f}(\bar{x}))$ **where** $P[\bar{x}] = R := \{\bar{x}/R(\bar{x}) \wedge \neg P[\bar{x}]\}$

$$\cup\ \{\bar{y}/\exists \bar{x}(R(\bar{x}) \wedge P[\bar{x}] \wedge \bigwedge_{i=1}^{n} y_i = f_i(\bar{x}))\},$$

where R is an n-pred variable of $\mathscr{L}$, $\bar{x} = (x_1, \ldots, x_n)$ are distinct individual variables occurring free in P, a tight wff of $\mathscr{L}$, and $\bar{f} = (f_1, \ldots, f_n)$ are function symbols of the sort (ind, ind, ..., ind).

A test P? indicates whether the computation should continue or not, depending on the truth of P. The union s $\cup$ t of s and t indicates that s or t should be executed next. The iteration s* of s means that s must be repeatedly executed an unspecified number of times. Taken together, they permit defining the following Algol-like constructs:

(18) **if** P **then** r **else** s = (P?; r) $\cup$ ($\neg P$?; s),

(19) **if** P **then** r = (P?; r) $\cup$ ($\neg P$?),

(20) **while** P **do** r = (P?; r)*; $\neg P$?.

We treat **begin** and **end** just as left and right parentheses, respectively. It is also possible to define the nondeterministic IF and DO of [14]:

$$(21)\quad \textbf{if}\ B_1 \rightarrow r // \cdots // B_n \rightarrow r_n\ \textbf{fi} \quad = (B_1\,?\,;\, r_1 \cup \cdots B_n\,?\,;\, r_n),$$

$$(22)\quad \textbf{do}\ B_1 \rightarrow r_1 // \cdots // B_n \rightarrow r_n\ \textbf{od} = (B_1\,?\,;\, r_1 \cup \cdots \cup B_n\,?\,;\, r_n)^*;\ \bigwedge_{i=r}^{n} \neg B_i\,?.$$

A random tuple selection $\bar{x} \leftarrow ?\ E$ assigns to $\bar{x}$ an arbitrary tuple of the relation denoted by E. Using the random tuple selection, we define the **for-each** construct of [19], which scans a relation tuple by tuple in an arbitrary order:

```
(23) for each $R(\bar{x})$ where $P[\bar{x}]$ key $K$ do s =
         $R_0 := \emptyset$; $R_T := \{\bar{y}/R(\bar{y}) \wedge P[\bar{y}/\bar{x}]\}$;
     while $R_T \neq \emptyset$ do
       ($\bar{x} \leftarrow ?\ R_T$;
       $R_0 := R_0 \cup \{\bar{x}\}$;
       s;
       $R_T := \{\bar{y}/R(\bar{y}) \wedge P[\bar{y}/\bar{x}] \wedge \neg\exists\bar{z}(R_0(\bar{z}) \wedge \bigwedge_{i \in K} z_i = y_i)\}$),
```

where R, R_0 and R_T are variables of $\mathscr{L}$ of the n-pred sort, $\bar{x} = (x_1, \ldots, x_n)$ are distinct individual variables occurring free in P, a tight wff of $\mathscr{L}$, and $K \subset [1, n]$ is a nonminimal key [11] of R. The **for-each** construct then scans the relation denoted by R in an arbitrary order; all tuples already scanned are kept in R_0; the last line of code in (23) guarantees that any tuple whose key has not been altered by s will not be scanned twice.

Thus the statements and constructs of MSRP[$\mathscr{L}$, U] permit defining the Algol fragment of [22], the basic constructs of [14], the relational operations, **insert**, **delete**, **update**, and **retrieve**, of [7, 10, 21], and the **for each** construct of [19] in an economical way. Such economy does not impair the elegance of transactions and will pay off when studying the metatheory of our programming logic.

Examples of programs in MSRP[$\mathscr{L}$, U] appear in Section A3 of the appendix.

We close this section with a very brief remark on whether or not our DML captures the "desirable" queries of [1, 8]. Following [1], the desirable queries are those that can be expressed by the relational operators originally introduced by Codd [11] (Cartesian product, set union, set difference, selection or restriction, and projection), plus a new least fixed point operator. Intuitively, this last operator permits defining a relation R inductively. We claim that the desirable queries can indeed be expressed in our DML. Each of the original relational operations has its direct counterpart in the special many-sorted languages used to write the right-hand side of assignments of our DML. Hence any query involving only these operators can be expressed as an assignment in our DML. Now any query involving the least fixed point operator can be expressed procedurally using **while** loops, as suggested in [1]. Therefore any desirable query can be expressed by a program in our DML.

5. REASONING ABOUT DATA MANIPULATION

In this section we offer a variant of dynamic logic (DL) to reason about many-sorted regular programs accessing a database. DL is not essential to our work and could be replaced by any other programming logic that accounts for input-output properties of programs and caters to bounded nondeterminism, in the sense of [14, Ch. 9]. However, DL has some attractive characteristics, explored in the next section.

Section 5.1 presents the language and an axiom system for our variant of DL, which is almost identical to regular first-order DL [20] (abbreviated here to FDL). The similarities between the two logics are explored in Section 5.2 to outline consistency and completeness results for our logic. Section 5.2 is optional and directed to those familiar with the results for FDL.

5.1 Regular Many-Sorted Dynamic Logic

Let $\mathscr{L}$ be a special many-sorted language, chosen as the DDL, and let MSRP[$\mathscr{L}$, U] be the set of many-sorted regular programs acting as a DML, where U is a universe of $\mathscr{L}$. Recall that $\mathscr{L}$ contains the addition symbol + and the individual constants 0 and 1, which receive the usual interpretations in U. We define the *regular many-sorted dynamic logic over* $\mathscr{L}$ *and* U, MDL[$\mathscr{L}$, U], as a formal system as follows.

Language. The [ ]-*extension* $\mathscr{LB}$ of $\mathscr{L}$, defined as follows.

Syntax. The syntax is the same as that of $\mathscr{L}$, with the additional formation rule:

(1) if P is a wff of $\mathscr{L}$ or $\mathscr{LB}$ and r is a program in MSRP[$\mathscr{L}$, U], then [r]P is a wff of $\mathscr{LB}$ (read "box of r, P").

Semantics. The notion of validity is extended to [r]P as follows:

(2) $I \models [\mathrm{r}]P$ iff $\forall J((I, J) \in m(r) \Rightarrow J \models P)$ or, in words, [r]P is valid in I iff either r does not halt starting in I (that is, for no J in U, (I, J) is in m(r)) or in any state J that can be reached from I via r, P is valid.

Proof theory. The axiom system AS follows (P and Q are wffs of $\mathscr{LB}$, except where noted):

(3) all tautologies of propositional calculus;

(4) all wffs of $\mathscr{L}$ valid in U;

(5) $[x := E]Q \equiv Q[E/x]$, $\quad Q$ a wff of $\mathscr{L}$;

(6) $[P?]Q \equiv P \Rightarrow Q$, $\quad P$ a quantifier-free wff of $\mathscr{L}$;

(7) $[\bar{x} \leftarrow ?\ E]Q \equiv \forall \bar{x}(e^n(E, \bar{x}) \Rightarrow Q)$, $\quad Q$ a wff of $\mathscr{L}$;

(8) $[\mathrm{s} \cup \mathrm{t}]Q \equiv [\mathrm{s}]Q \wedge [t]Q$;

(9) $[\mathrm{s};\ \mathrm{t}]Q \equiv [\mathrm{s}][\mathrm{t}]Q$;

(10) modus ponens:

$$\frac{P,\ P \Rightarrow Q}{Q};$$

(11) $\exists_i$-introduction:

$$\frac{P \Rightarrow Q}{\exists_i xP \Rightarrow \exists_i xQ};$$

for each sort i of $\mathscr{L}$;

(12) necessitation:

$$\frac{P \Rightarrow Q}{[\mathrm{s}]P \Rightarrow [\mathrm{s}]Q};$$

(13) invariance:

$$\frac{P \Rightarrow [\mathrm{s}]P}{P \Rightarrow [\mathrm{s}]^* P};$$

(14) convergence:

$$\frac{P[n+1/n] \Rightarrow \langle \mathrm{s} \rangle P[n]}{P[n] \Rightarrow \langle \mathrm{s}^* \rangle P[0/n]},$$

where P is a wff of $\mathscr{L}$ with a free variable n of the individual sort not occurring in s.

We also add $\langle \mathrm{r} \rangle P$ (in words, "diamond of r, P") as an abbreviation for $\neg[\mathrm{r}]\neg P$; $I \models \langle \mathrm{r} \rangle P$ means that there is a state J that can be reached from I via r and in which P is valid.

The language of MDL[$\mathscr{L}$, U] can express, for example, the following properties of programs accessing a database described by the schema $\sigma = (\rho, \alpha, \delta)$ with $\rho = \{R_1, \ldots, R_m\}$.

(15) r is a transaction of σ (r preserves consistency for σ):

$$\models_U \bar{P} \Rightarrow [\mathrm{r}]\bar{P}, \qquad \text{where} \quad \bar{P} = \bigwedge_{P \in \delta} P;$$

(16) r performs the same changes in the database relations as does s:

$$\models_U \forall \rho' (\langle \mathrm{r} \rangle \rho = \rho' \equiv \langle \mathrm{s} \rangle \rho = \rho'),$$

where $\rho' = \{R'_1, \ldots, R'_m\}$, R'_i with the same arity as R_i $(1 \le i \le m)$, and $\rho = \rho'$ abbreviates $\bigwedge_{i=1}^{m} R_i = R'_i$.

Therefore, MDL[$\mathscr{L}$, U] permits us to investigate not only consistency preservation, but also equivalence with respect to σ.

In Section A4 of the appendix we exemplify how to prove properties of programs using the axiom system AS.

5.2 Soundness and Completeness Results

In this section we briefly investigate the soundness and completeness of the axiom system AS of MDL[$\mathscr{L}$, U]. Both results follow from the metatheory of regular first-order dynamic logic (FDL) developed in [20] and are described in more detail in [5]. In adapting results for FDL to MDL, one should keep in mind that their languages differ in the assertion language, which is a many-sorted language in MDL instead of a one-sorted one, and in the programming language, which allows random tuple selections. Moreover, the axiom system for MDL is exactly that given in [20] for FDL, with one extra axiom for random tuple selections.

Let $\mathscr{L}$ be a special many-sorted language, U be a universe for $\mathscr{L}$, $\mathscr{LB}$ be the [ ]-extension of $\mathscr{L}$, and B be an axiom system for $\mathscr{LB}$. B is *U-sound* iff, for any wff P of $\mathscr{LB}$, $\vdash_B P$ implies $\models_U P$. Thus B is U-sound iff every axiom of B is U-valid and every inference rule of B preserves U-validity. B is *U-complete* iff, for every wff P of $\mathscr{LB}$, $\models_U P$ implies $\vdash_B P$.

The U-soundness of the axiom system AS follows because the axiom system proposed in [20] for FDL remains U-sound when we take a many-sorted language as assertion language. This implies that all rules of AS preserve U-validity and Axioms (5), (6), (8), and (9) are U-valid. The U-validity of Axiom (7) is proved in Theorem 5.1.

THEOREM 5.1 $\models_U([\bar{x} \leftarrow ?\ E]Q \equiv \forall \bar{x}(e^n(E, \bar{x}) \Rightarrow Q))$, *$Q$ a wff of $\mathscr{L}$.*

PROOF. Let I be in U, $s = \bar{x} \leftarrow ?\ E$ and Q be a wff of $\mathscr{L}$. Then we have

$$\begin{array}{ll}
I \models [s]Q \text{ iff } (\forall J \in U)((I, J) \in m(s) \Rightarrow J \models Q) & (\text{definition of } I \models [s]Q), \\
\quad \text{iff } (\forall J \in U)(J = [\bar{a}/\bar{x}]I \wedge \bar{a} \in E_I \Rightarrow & (\text{definition of } m), \\
\qquad J \models Q) & \\
\quad \text{iff } (\forall\ \bar{a} \in D)(\bar{a} \in E_i \Rightarrow I \models Q[\bar{a}/\bar{x}]), & (\text{definition of } U), \\
\qquad \text{where } D \text{ is the domain of the sort} & \\
\qquad \text{of } E & \\
\quad \text{iff } I \models \forall \bar{x}(e^n(E, \bar{x}) \Rightarrow Q) & (\text{definition of } I \models P). \quad \square
\end{array}$$

The U-completeness of AS again follows directly from results for FDL, except that the adoption of a special many-sorted language does matter now. We restate here Harel's theorem of completeness [20, Th. 3.1] (specialized to our case), which helps one to understand the differences between FDL and MDL. We say that $\mathscr{L}$ is *U-expressive for $\mathscr{LB}$* iff, for any wff P of $\mathscr{LB}$, there is a wff Q of $\mathscr{L}$ such that $\models_U(P \equiv Q)$. An axiom system B for $\mathscr{LB}$ is *propositionally complete* iff all instances of tautologies of propositional calculus are theorems of B and modus ponens is an inference rule.

THEOREM 5.2 [20, Th. 3.1] *For any universe U of $\mathscr{L}$, a U-sound axiom system B for $\mathscr{LB}$ is U-complete if*

(*i*) *B is propositionally complete*;
(*ii*) *$\mathscr{L}$ is U-expressive for $\mathscr{LB}$*;
(*iii*) *for any program r, variable x of sort i and wffs P and Q of $\mathscr{LB}$,*
  (*a*) *if* $\vdash_B(P \Rightarrow Q)$, *then* $\vdash_B([r]\ P \Rightarrow [r]Q)$,
  (*b*) *if* $\vdash_B(P \Rightarrow Q)$, *then* $\vdash_B(\exists_i xP \Rightarrow \exists_i xQ)$;
(*iv*) *for any program r and any wffs P and Q of $\mathscr{L}$*
  (*a*) *if* $\models_U P$ *then* $\vdash_B P$,
  (*b*) *if* $\models_U(P \Rightarrow [r]Q)$ *then* $\vdash_B(P \Rightarrow [r]Q)$,
  (*c*) *if* $\models_U(P \Rightarrow \langle r \rangle Q)$ *then* $\vdash_B(P \Rightarrow \langle r \rangle Q)$.

Theorem 5.2 should be understood in the light of Cook's pioneering paper

[12]. Cook first observed that, in general, we cannot obtain a U-complete axiom system B for $\mathscr{LB}$. Indeed, mimicking the argument in [12, p. 85], the formulas $[\mathrm{r}]P$ provable in B are recursively enumerable, since B is an axiom system. However, the formula [r]false is true in U iff r fails to halt for all initial states in U. Therefore the true formulas cannot be recursively enumerable in the case in which the halting problem for our programming language is recursively unsolvable. But this is the case here, since we assume that $\mathscr{L}$ contains =, +, 0, and 1, which receive their usual interpretation in U ([12, Th. 2, p. 85]). Cook's suggestion was to prove the completeness of B assuming a complete system $B(\mathscr{L})$ for $\mathscr{L}$ and assuming the U-expressiveness of $\mathscr{L}$, or what he called *relative completeness*. These assumptions correspond to conditions (ii) and (iv-a) of Theorem 5.2. The axioms and rules involving [ ] and $\langle\,\rangle$ should then be viewed as mechanisms for translating a wff of P of $\mathscr{LB}$ into an equivalent set of wffs of $\mathscr{L}$, which can then be proved in $B(\mathscr{L})$ (see [20, Sec. 3.4.1]).

We now briefly discuss how to meet the conditions of Theorem 5.2. Conditions (i), (iii), and (iv-a) of Theorem 5.2 express exactly the same requirements as (3) and (10), (11) and (12), and (4) of the axiom system AS, respectively. Conditions (iv-b) and (iv-c) are achieved by induction on the structure of the program r exactly as for FDL ([20, Th. 3.9 and 3.11]), using Axiom (7) of AS to cope with random tuple selection. Condition (ii) of Theorem 5.2 needs a more detailed discussion, though.

As for FDL, we prove that if $\mathscr{L}$ contains arithmetic and U is an arithmetical universe, then $\mathscr{L}$ is U-expressive for $\mathscr{LB}$. We say that *$\mathscr{L}$ contains arithmetic* iff $\mathscr{L}$ has the function symbols + and . of sort (ind, ind, ind), the individual constants 0 and 1, and the predicate symbol = of sort (ind, ind). A universe U of $\mathscr{L}$, generated by a special structure A of $\mathscr{L}$, is an *arithmetical universe* iff A assigns the usual interpretations to +, ., 0, 1, = (by definition of special structure, the individual domain of A is the set of natural numbers $\mathbb{N}$) and effective interpretations to the other function and predicate symbols of $\mathscr{L}$.

Let $R_r(\bar{m}, \bar{n})$ be the relation each program r computes, where $\bar{m}$ and $\bar{n}$ represent the initial and final values of the variables $\bar{x}$ that r modifies. Note that R_r is not necessarily the graph of a function because r is nondeterministic.

LEMMA 5.1 *Assume that programs use only individual variables. Then, if $\mathscr{L}$ contains arithmetic and U is an arithmetical universe, $\mathscr{L}$ is U-expressive for $\mathscr{LB}$.*

SKETCH OF PROOF. We give in essence the argument in [20] for FDL. Let r be a program that uses only individual variables. By assumption on U, R_r is a recursively enumerable relation over $\mathbb{N}$. Since $\mathscr{L}$ contains arithmetic, U is an arithmetical universe and R_r is recursively enumerable; R_r is definable in $\mathscr{L}$. That is, there is a wff $F_r[\bar{x}, \bar{y}]$ of $\mathscr{L}$ with free variables $\bar{x}$ and $\bar{y}$ such that $\models_U F_r[\bar{m}/\bar{x}, \bar{n}, \bar{y}]$ iff $R_r(\bar{m}, \bar{n})$. Now, given a wff Q of $\mathscr{L}$, $\langle \mathrm{r}\rangle Q$ is U-valid iff $\exists\bar{y}(F_r[\bar{x}, \bar{y}] \wedge Q[\bar{y}/\bar{x}])$ is U-valid, by definition of $\models_U\langle \mathrm{r}\rangle Q$ and construction of F_r. Given any wff P of $\mathscr{LB}$, we then proceed by induction to eliminate all occurrences of boxes [] and diamonds $\langle\,\rangle$ in P, using the equivalent in U of $\langle \mathrm{r}\rangle Q$ to obtain a wff $P_{\mathscr{L}}$ of $\mathscr{L}$ equivalent to P in U. □

THEOREM 5.3 *If $\mathscr{L}$ contains arithmetic and U is an arithmetical universe, then $\mathscr{L}$ is U-expressive for $\mathscr{LB}$.*

SKETCH OF PROOF. We indicate that the argument in Lemma 5.1 carries on here. The only doubt lies in that a program r may now modify variables that are valued as relations over N. Thus, at first sight, we may not capture what r computes by a relation R_r over N and claim that R_r is recursively enumerable. However, all relations in the n-pred domain, $n > 0$, are finite and hence can be mapped into natural numbers (given a finite relation S over N with $|S| = m$, we can use sequence numbers [30, Ch. 6.4] to encode each tuple t_i in S as n_i in N; the natural number corresponding to S will be the sequence number encoding $(n_1, \ldots, n_m)$). Using such a mapping and the fact that P in $\{\bar{x}/P[\bar{x}]\}$ is a tight wff of $\mathscr{L}$, we can still claim that r computes a recursively enumerable relation over N, and the argument in the proof of Lemma 5.1 can be repeated here. □

Finally, we state the completeness theorem for AS.

THEOREM 5.4 *If $\mathscr{L}$ contains arithmetic and U is an arithmetical universe, then AS is U-complete.*

SKETCH OF PROOF. The proof follows from Theorems 5.2 and 5.3. Conditions (i), (iii), and (iv-a) of Theorem 5.2 are equivalent to Axioms (3) and (10), (11) and (12), and (4) of the axiom system AS, respectively. Conditions (iv-b) and (iv-c) of Theorem 5.2 are obtained by induction on the structure of a program as for FDL ([20, Th. 3.9 and 3.11]). Finally, condition (ii) of Theorem 5.2 corresponds to Theorem 5.3. □

6. CORRECTNESS OF DATABASE SYSTEMS

In this section we outline how regular many-sorted dynamic logic can be applied to database management systems (DBMSs) supporting concurrent transaction. Correctness criteria for DBMSs guarantee, among other things, that each transaction is correctly executed, queries read consistent data, or consistency is preserved. To study the correctness of DBMSs, we distinguish two types of systems.

An *open* DBMS, such as IMS [23], supports any transaction mix accessing a database, acquiring information about transactions as they are submitted. Correct concurrent execution is guaranteed by *schedulers*, subsystems designed to intercept and reorder all access requests or synchronization calls (such as look requests). Schedulers work with imperfect information, since transactions are not known in advance, and they must be efficient, because they operate on-line. Hence their study tends to center around the design of efficient algorithms, rather than around program correctness. A survey of some scheduler designs appears in [3] and their correctness criteria appear in [5].

A *closed* DBMS, such as an airline or hotel reservation system, is characterized by a known set of transactions accessing a fixed database. Schedulers can also be used here to guarantee correct concurrent execution, but special code is likely to achieve better performance, particularly when transactions interfere heavily and yet a fast response time is required. We can model a closed DBMS as a concurrent program, whose components are the transactions, and study its correctness using an appropriate logic.

We now briefly apply regular many-sorted dynamic logic to closed DBMSs, concentrating on criteria guaranteeing that transactions are correctly executed and consistency is preserved. Let $\mathscr{L}$ be a special many-sorted language, let U be a universe for $\mathscr{L}$, let $\sigma = \langle \rho, \alpha, \delta \rangle$, with $\rho = \{R_1, \ldots, R_m\}$, be a database schema, and let $t = \{t_1, \ldots, t_n\}$ be a set of concurrent transactions for σ. Following [18], we model t by a nondeterministic **do** loop r, as exemplified in Section A5 of the appendix.

Our previous notion of consistency preservation also applies to r, since r is in MSRP[$\mathscr{T}$, U]. So we only discuss transaction execution. Informally, an execution of r mapping a state I into I' correctly executes each transaction iff I' can be obtained from I by executing one transaction after the other in some arbitrary order. More precisely, s is a *serialization* of r iff s is a program of the form $t_{i_1}; \ldots; t_{i_n}$, where $i_1 \ldots i_n$ is a permutation of $1 \ldots n$. Let SER(r) be the (finite) set of all serializations of r. We say that r is *serializable* iff $\models_U \forall V'((\langle \mathrm{r} \rangle V = V') \equiv (\langle \bigcup_{s \in SER(r)} s \rangle V = V'))$, where $V = (v_1, \ldots, v_n)$ is an ordering of all variables of $\mathscr{L}$ modified by r and $V' = (v'_1, \ldots, v'_n)$ is another vector of variables of $\mathscr{L}$ such that v'_i does not occur in r and has the same sort as v_i, i in $[1, n]$. Serializability is then a case of program equivalence and represents a version of the usual notion of serializability [4, 17] for interpreted transactions. We may generalize serializability by taking V to be any set of variables. Finally, we note that the language of DL permitted defining serializability concisely; this elegance would be lost if we adopted a logic accounting only for input-output properties of programs such as Hoare's logic [22].

By the results given in Section 5.2, the axiom system AS is theoretically adequate to prove consistency preservation and serializability for closed DBMSs. In Section A5 of the appendix we outline a serializability proof. However, constructing consistency preservation or serializability proofs can be quite difficult, even for very simple systems, so it pays to supplement AS with special heuristics [5]. In fact, considerable work must still be done in the area of closed DBMSs to harness concurrency and bring down to a manageable size the task of proving correctness of these systems.

7. CONCLUSIONS

We transferred a considerable amount of programming logics theory to databases by considering the database data structures as part of the program, so that database accesses reduce to assignments. Choosing dynamic logic as the underlying programming logic permitted us to study not only consistency preservation, but also transaction equivalence and serializability. However, other logics capturing the on-going properties of programs are needed to study further properties, such as reliability, that require the database state to be always consistent (perhaps after some rollback).

APPENDIX

A1 Example of a Relational Schema

In this appendix we exemplify how to write database schemas and programs, and we prove facts about them. The DDL will be a special many-sorted language $\mathscr{L}$ with a 2-pred variable ACC, 1-pred constants NO and BAL, and function symbols $+$ and $-$ of sort (ind, ind, ind). $\mathscr{L}$ is equipped with a special structure A assigning to NO and BAL the values $[1, p]$ and $[0, \infty)$, respectively, and to $+$ and $-$ the usual interpretations. U denotes the universe of $\mathscr{L}$ generated by A. The DML will be the set of many-sorted regular programs over $\mathscr{L}$ and U, MSRP[$\mathscr{L}$, U].

We define a relational database with just one table containing account numbers and balances. Each entry (n, b) in the table is uniquely identified by the account number n and we assume that n ranges from 1 to p and that b is nonnegative. The corresponding database schema goes as follows:

(1) BANK = ({ACC}, {NO, BAL}, {P1, P2, P3}), where
(2) P1 = $\forall n \forall b(\mathrm{ACC}(n, b) \Rightarrow \mathrm{NO}(n) \wedge \mathrm{BAL}(b))$ (the relational scheme of ACC);
(3) P2 = $\forall n \forall b \forall b'(\mathrm{ACC}(n, b) \wedge \mathrm{ACC}(n, b') \Rightarrow b = b')$;
(4) P3 = $\forall n(\mathrm{NO}(n) \equiv 1 \leq n \leq p) \wedge \forall b(\mathrm{BAL}(b) \equiv 0 \leq b)$.

Note. We frequently use syntactical constants, such as P1, ranging over wffs of $\mathscr{L}$ and programs of MSRP[$\mathscr{L}$, U].

A2 Example of a Derivation in a Special Many-Sorted Theory

We exemplify in this section how to use the special many-sorted theory $\mathscr{T}$ of $\mathscr{L}$. Suppose that we want to talk about the affluent people of the bank, defined as those with balances greater than 10K. A convenient approach consists of extending $\mathscr{T}$ to a new theory $\mathscr{T}' = (\mathscr{L}', D')$ by introducing by definition a 2-pred constant AFF with defining axiom

(1) $\mathrm{AFF} = \{(m, c)/\mathrm{ACC}(m, c) \wedge c \geq 10\mathrm{K}\}$;

that is, AFF is a *view* of BANK [13]. Let P4 be a wff of $\mathscr{L}'$ expressing that tuples in the value of AFF are still uniquely identified by account numbers:

(2) P4 = $\forall n \forall b \forall b'(\mathrm{AFF}(n, b) \wedge \mathrm{AFF}(n, b') \Rightarrow b = b')$.

We now show that P2 $\vdash_{\mathscr{T}'}$ P4. From Axioms (1) and (2), P4 is equivalent to

(3) $\forall n \forall b \forall b'(e^2(\{(m, c)/\mathrm{ACC}(m, c) \wedge c \geq 10\mathrm{K}\}, n, b)$
$\wedge\, e^2(\{m, c)/\mathrm{ACC}(m, c) \wedge c \geq 10\mathrm{K}\}, n, b') \Rightarrow b = b')$.

Using the set operations axioms of $\mathscr{T}'$, P4 is then equivalent to

(4) $\forall n \forall b \forall b'(\exists X \exists Y((X(n, b) \wedge Y(n, b') \Rightarrow b = b')$
$\wedge\, \forall m \forall c(X(m, c) \equiv \mathrm{ACC}(m, c) \wedge c \geq 10\mathrm{K})$
$\wedge \forall m \forall c(Y(m, c) \equiv \mathrm{ACC}(m, c) \wedge c \geq 10\mathrm{K})))$.

Simplifying (4), we obtain

(5) $\forall n \forall b \forall b'(\mathrm{ACC}(n, b) \wedge b \geq 10\mathrm{K} \wedge \mathrm{ACC}(n, b') \wedge b' \geq 10\mathrm{K} \Rightarrow b = b')$,

which now follows from P2.

A3 Examples of Regular Many-Sorted Programs

We describe two programs, AUDIT and TRANSFER, which access BANK. AUDIT scans the value of ACC tuple by tuple, in descending order of account number, and computes the bank's assets. TRANSFER transfers D dollars from account N to account M, if N has enough funds.

```
AUDIT
begin comment n—current account number being scanned (n = 0 when ACC has been
                    scanned because, by definition, $MAX_i^n(\emptyset) = 0$)
                 b—unique balance of account n
                 s—sum of the balances of all accounts up to n

  $s := 0$;
  $n := MAX_1^2(ACC)$;
  while $n \neq 0$ do
  begin $b \leftarrow ? \{c/ACC(n, c)\}$;
         $s := s + b$;
         $n := MAX_1^1(\{m/\exists c(ACC(m, c) \wedge m < n)\})$
  end
end
```

```
TRANSFER
begin $b \leftarrow ? \{c/ACC(N, c)\}$;
  if $b > D$
    then begin update $ACC(m, c+D)$ where $m = M$;
               update $ACC(m, c-D)$ where $m = N$;
         end
end
```

or, eliminating all defined constructs:

$$\text{TRANSFER} = \text{T1}; ((b > \text{D}?; T2; T3) \cup (\neg b > \text{D}?)),$$

where

$$\text{T1} = b \leftarrow ? \{c/\text{ACC}(\text{N}, c)\}$$

$$\text{T2} = \text{ACC} := \text{ACC1}$$

$$\text{T3} = \text{ACC} := \text{ACC2}$$

$$\begin{aligned}\text{ACC1} &= \{(m, c)/\text{ACC}(m, c) \wedge m \neq \text{M}\}\\ &\quad \cup \{(m, c)/\exists d(\text{ACC}(m, d) \wedge m = \text{M} \wedge c{=}d{-}\text{D}\}\\ \text{ACC2} &= \{(m, c)/\text{ACC}(m, c) \wedge m \neq \text{N}\}\\ &\quad \cup \{(m, c)/\exists d(\text{ACC}(m, d) \wedge m = \text{M} \wedge c = d{+}\text{D}\}.\end{aligned}$$

Note. ACC1 and ACC2 are syntactical constants standing for the right-hand side of T1 and T2, respectively.

A4 Examples of Proofs

We outline a proof that TRANSFER preserves the bank's assets and AUDIT computes the bank's assets, assuming a consistent initial state in both cases.

More precisely, for TRANSFER we prove that

(1) $s = \mathrm{SUM}^2_2(\mathrm{ACC}) \wedge \mathrm{P1} \wedge \mathrm{P2} \wedge \mathrm{P3} \Rightarrow [\mathrm{TRANSFER}]s = \mathrm{SUM}^2_2(\mathrm{ACC})$

where s is an individual variable of $\mathscr{L}$ (not occurring in TRANSFER). Let $P = (s = \mathrm{SUM}^2_2(\mathrm{ACC}))$. To prove (1), we first obtain a wff Q of $\mathscr{L}$ equivalent to $[\mathrm{TRANSFER}]P$:

(2)	$[\mathrm{TRANSFER}]P \equiv [\mathrm{T1}]([b > \mathrm{D}?][\mathrm{T2}][\mathrm{T3}]P \wedge [\neg b > \mathrm{D}?]P)$	[axioms for composition and union, rule of necessitation, and definition of TRANSFER]
(3)	$[\mathrm{T3}]P \equiv \mathrm{P4}$, with $\mathrm{P4} = P[\mathrm{ACC2}/\mathrm{ACC}]$	[axiom for assignments and definition of T3]
(4)	$[\mathrm{T2}][\mathrm{T3}]P \equiv [\mathrm{T2}]\mathrm{P4}$	[Axiom (3), necessitation]
(5)	$[\mathrm{T2}]\mathrm{P4} \equiv \mathrm{P5}$, with $\mathrm{P5} = \mathrm{P4}[\mathrm{ACC1}/\mathrm{ACC}]$	[axiom for assignments and definition of T2]
(6)	$[\mathrm{T2}][\mathrm{T3}]P \equiv \mathrm{P5}$	[Axioms (4), (5)]
(7)	$[b > \mathrm{D}/][\mathrm{T2}][\mathrm{T3}]P \equiv [b > \mathrm{D}?]\mathrm{P5}$	[Axiom (6), necessitation]
(8)	$[b > \mathrm{D}?]\mathrm{P5} \equiv \mathrm{P6}$, with $\mathrm{P6} = (b > \mathrm{D}? \Rightarrow \mathrm{P5})$	[axiom for tests]
(9)	$[b > \mathrm{D}?][\mathrm{T2}][\mathrm{T3}]P \equiv \mathrm{P6}$	[Axioms (7), (8)]
(10)	$[\neg b > \mathrm{D}?]P \equiv \mathrm{P7}$, with $\mathrm{P7} = (\neg b > \mathrm{D}? \Rightarrow P)$	[axiom for tests]
(11)	$[\mathrm{T1}]([b > \mathrm{D}?][\mathrm{T2}][\mathrm{T3}]P \wedge [\neg b > \mathrm{D}?]P) \equiv [\mathrm{T1}](\mathrm{P6} \wedge \mathrm{P7})$	[Axioms (9), (10), propositional reasoning, necessitation]
(12)	$[\mathrm{T1}](\mathrm{P6} \wedge \mathrm{P7}) \equiv Q$, with $Q \equiv \forall d(e^1(\{c/\mathrm{ACC}(n, c)\}, d) \Rightarrow \mathrm{P6}[d/b] \wedge \mathrm{P7}[d/b])$	[axiom for random tuple selections, definition of T1]
(13)	$[\mathrm{TRANSFER}]P \equiv Q$	[Axioms (2), (12)]

Then (1) is equivalent to $P \wedge \mathrm{P1} \wedge \mathrm{P2} \wedge \mathrm{P3} \Rightarrow Q$, which does not involve any program. It is not difficult to convince oneself that this formula is valid in U. Hence it is an axiom of AS (clause (4) of Section 5.1) and the proof is completed.

For AUDIT, we want to prove that

(14) $\mathrm{P1} \wedge \mathrm{P2} \wedge \mathrm{P3} \Rightarrow [\mathrm{AUDIT}]s \equiv \mathrm{SUM}^2_2(\mathrm{ACC})$.

The proof of (14) is based on a derived rule for **while** constructs [20, 22], defined as

$$\textit{while rule}: \frac{P \wedge B \Rightarrow [\mathrm{r}]P}{P \Rightarrow [\textbf{while } B \textbf{ do } \mathrm{r}]P \wedge \neg B}.$$

The proof proceeds by taking P as

(15) $P = (s = \mathrm{SUM}^2_2(\{(m, c)/\mathrm{ACC}(m, c) \wedge m > n\}) \wedge \mathrm{P3})$.

(P2 is needed to derive that b is the unique balance associated with n.)

A5 A Serializability Proof

In this section we outline a proof that AUDIT and TRANSFER, when synchronized, form a serializable set of transactions with respect to the assets s computed by AUDIT. We model $t = \{\text{AUDIT, TRANSFER}\}$ by a nondeterministic **do** loop r, in the manner of [18]:

```
r = $c_1 := 0$; $c_2 := 0$;
    do $c_1 = 0 \rightarrow s := 0; c_1 := 1$
    // $c_1 = 1 \rightarrow n := \text{MAX}_1^2(\text{ACC}): c_1 := 2$
    // $c_1 = 2 \wedge n \neq 0 \rightarrow c_1 := 3$
    // $c_1 = 3 \rightarrow b \leftarrow ?\{c/\text{ACC}(n, c)\}; c_1 := 4$
    // $c_1 = 4 \rightarrow s := s + b; c_1 := 5$
    // $c_1 = 5 \rightarrow n := \text{MAX}_1^1(\{m/\exists c(\text{ACC}(m, c) \wedge m < n)\}); c_1 := 2$
    // $c_1 = 2 \wedge n = 0 \rightarrow c_1 := 6$
    // $c_2 = 0 \wedge \neg(\text{M} \leq n \leq \text{N}) \wedge \neg(\text{N} \leq n \leq \text{M}) \rightarrow \text{TRANSFER}; c_2 := 1$
    od
```

The variables c_1 and c_2 act as program counters and each line of the **do** loop corresponds to an atomic action. Thus we model a concurrent execution of t by the nondeterministic interleaving of the atomic actions of AUDIT and TRANSFER. In fact, we consider TRANSFER a single atomic action that synchronizes with AUDIT via the condition $\neg(\text{M} \leq n < \text{N}) \wedge \neg(\text{N} \leq n < \text{M})$. That is, TRANSFER cannot move money from one account that AUDIT has already scanned to another one that AUDIT will still scan (otherwise AUDIT would sum the transferred dollars twice). By requiring that r be serializable with respect to the assets s, we guarantee that AUDIT runs as if alone and thus correctly sums up the bank's assets. More precisely, we require that

$$\forall x((\langle r \rangle x = s) \equiv (\langle \textstyle\bigcup_{p \in \text{SER(r)}} p \rangle \; x = s)). \tag{A1}$$

We now outline how (A1) could be proved. First, observe that the right-hand side always implies the left-hand side, that is, a serial execution of the transactions is just a special case of a concurrent execution. Assuming that $\mathscr{L}$ is expressive for its [ ]-extension, we can find a wff P of $\mathscr{L}$ equivalent to the right-hand side of the equivalence. In our case, this task is relatively simple, since the assets computed by AUDIT remain the same before and after TRANSFER is executed. Thus P is simply $x = \text{SUM}_2^2(\text{ACC})$. In view of these observations, (A1) is equivalent to

$$\forall x((\langle r \rangle x = s) \Rightarrow P) \tag{A2}$$

which is in turn equivalent to (using $\langle \text{r} \rangle x = s \equiv \neg[\text{r}]x \neq s$)

$$\forall x(\neg P \Rightarrow [\text{r}]x \neq s). \tag{A3}$$

But proving (A3) reduces to the familiar problem of synthesizing an invariant for r in order to apply the invariance rule of AS. In our case, an appropriate invariant is

$$\begin{aligned} I = x \neq \mathrm{SUM}_2^2(\mathrm{ACC}) &\wedge \forall n \forall b(\mathrm{ACC}(n, b) \Rightarrow n > 0) \\ &\wedge (c_1 = 1 \Rightarrow s = 0) \\ &\wedge (c_1 = 2 \vee c_1 = 3 \vee c_1 = 4 \\ &\quad \Rightarrow s = \mathrm{SUM}_2^2(\{(m, c)/\mathrm{ACC}(m, c) \wedge m > n\})) \\ &\wedge (c_1 = 4 \Rightarrow \mathrm{ACC}(n, b)) \\ &\wedge (c_1 = 5 \Rightarrow s = \mathrm{SUM}_2^2(\{(m, c)/\mathrm{ACC}(m, c) \wedge m > n\})) \\ &\wedge (c_1 \neq 1 \wedge \cdots \wedge c_1 \neq 5 \Rightarrow s = \mathrm{SUM}_2^2(\{(m, c)/\mathrm{ACC}(m, c)\})). \end{aligned} \tag{A4}$$

Note that I captures the execution of AUDIT and does not depend on TRANSFER.

REFERENCES

1. AHO, A.V., AND ULLMAN, J.B. Universality of data retrieval languages. In Proc. 6th Annu. ACM Symp. Principles of Programming Languages, Jan. 1979, pp. 110–120.
2. BANCILHON, F. On the completeness of query languages for relational data bases. In Proc. 7th Symp. Math. Foundations of Computer Science, Zakopane, Poland (*Lecture Notes in Computer Science*, Springer-Verlag, New York, Sept. 1978).
3. BERNSTEIN, P.A., ET AL. A formal model of concurrency control mechanisms for database systems. *IEEE Trans. Softw. Eng.* (May 1979).
4. BERNSTEIN, P.A., SHIPMAN, D.W., AND ROTHNIE, J.B., JR. Concurrency control in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (March 1980), 18–51.
5. CASANOVA, M.A. The concurrency control problem for database systems. Ph.D. dissertation, Harvard Univ., Cambridge, Mass., Nov. 1979.
6. CASANOVA, M.A., AND BERNSTEIN, P.A. The logic of a relational data manipulation language. In Proc. 6th ACM Symp. Principles of Programming Languages, Jan. 1979, pp. 101–109.
7. CHAMBERLIN, D.D., ET AL. SEQUEL-2: A unified approach to data definition manipulation and control. Tech. Rep. RJ 1798, IBM, New York, June 1976.
8. CHANDRA, A.K., AND HAREL, D. Computable queries for relational databases. In Proc. 11th ACM Symp. Theory of Computing, May 1979.
9. CLARKE, E.M., JR. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *J. ACM* 26, 1 (Jan. 1979), 129–147.
10. CODD, E.F. A database sublanguage founded on the relational calculus. In Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control.
11. CODD, E.F. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June 1970), 377–387.
12. COOK, S.A. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput. 7*, 1 (Feb. 1978).
13. DAYAL, U. Schema mapping problems in database systems. Ph.D. dissertation, Harvard Univ., Cambridge, Mass., Aug. 1979.
14. DIJKSTRA, E.W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.
15. ENDERTON, H.B. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.
16. ESWARAN, K.P. Specifications, implementations and interactions of a trigger subsystem in an integrated database system. Tech. Rep. RJ1820, IBM, New York, Aug. 1976.
17. ESWARAN, K.P., ET AL. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.
18. FLON, L., AND SUZUKI, N. Nondeterminism and the correctness of parallel programs. Presented at the IFIP Conf. Working Group on Formal Specifications of Programming Languages, Aug. 1977.

19. GARDARIN, G., AND MELKANOFF, M. Proving consistency of database transactions. In Proc. 1979 Int. Conf. Very Large Data Bases, Oct. 1979, pp. 291-298.
20. HAREL, D. First-order dynamic logic. In *Lecture notes in Computer Science*, vol. 68, Springer-Verlag, New York, 1979.
21. HELD, G. D., STONEBRAKER, M.R., AND WONG, E. INGRES—A relational database system. In *Proc.* 1975 *AFIPS NCC*, AFIPS Press, Arlington, Va., pp. 409-416.
22. HOARE, C.A.R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576-580.
23. Information management system virtual storage (IMS/VS) general information manual. IBM no. GH20-1260, IBM, New York.
24. LAMPORT, L. Towards a theory of correctness for multi-user data base system. Tech. Rep. TR-CA-7610-0712, Massachusetts Computer Associates, Oct. 1976.
25. MINKER, J. Search strategy and selection function for an inferential relational system *ACM Trans. Database Syst. 3*, 1 (Mar. 1978), 1-31.
26. NICOLAS, J.M. First-order logic formalization for functional, multivalued and mutual dependencies. In Proc. 1978 ACM-SIGMOD Int. Conf. Management of Data, May 1978, pp. 40-46.
27. PAPADIMITRIOU, C.H., BERNSTEIN, P.A., AND ROTHNIE, J.B. Some computational problems related to database concurrency control. In Proc. Conf. Theoretical Computer Science, Aug. 1977, pp. 275-282.
28. PRATT, V.R. Semantical considerations on Floyd-Hoare logic. In Proc. 17th IEEE Symp. Foundations of Computer Science, Oct. 1976, pp. 109-120.
29. SCHMIDT, J.W. Some high-level language constructs for data of type relation. *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 247-261.
30. SHOENFIELD, J.R. *Mathematical Logic.* Addison-Wesley, Reading, Mass., 1967.
31. STEARNS, R.E., ET AL. Concurrency control for database systems. In Proc. 17th IEEE Symp. Foundations of Computer Science, Oct. 1976, pp. 19-32.
32. STONEBRAKER, M. Implementation of integrity constraints and views by query modification. In Proc. ACM-SIGMOD Int. Conf. Management of Data, May 1975.
33. STONEBRAKER, M., WONG, E., AND KREPS, P. The design and implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 189-222.
34. THOMAS, R.H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4*, 2 (June 1979), 180-209.
35. VAN EMDEN, M.H. Computation and deductive information retrieval. Presented at the IFIP Conf. Working Group on Formal Specifications of Programming Languages, Aug. 1977.

Received March 1979; revised January and April 1980; accepted April 1980