Chapter 4

T h e T h i r d M a n i f e s t o

RM Prescriptions
RM Proscriptions
OO Prescriptions
OO Proscriptions
RM Very Strong Suggestions
OO Very Strong Suggestions
Recent *Manifesto* changes

RM PRESCRIPTIONS

1. A **scalar data type** (**scalar type** for short) is a named, finite set of scalar values (**scalars** for short). Given an arbitrary pair of distinct scalar types named *T1* and *T2,* respectively, with corresponding sets of scalar values *S1* and *S2,* respectively, the names *T1* and *T2* shall be distinct and the sets *S1* and *S2* shall be disjoint; in other words, two scalar types shall be equal——i.e., the same type——if and only if they have the same name (and therefore the same set of values). **D** shall provide facilities for users to define their own scalar types (*user-defined* scalar types); other scalar types shall be provided by the system (*built-in* or *system-defined* scalar types). **D** shall also provide facilities for users to destroy user-defined scalar types. The system-defined scalar types shall include type **boolean** (containing just two values, here denoted TRUE and FALSE), and **D** shall support all four monadic and 16 dyadic logical operators, directly or indirectly, for this type.
2. All scalar values shall be **typed**——i.e., such values shall always carry with them, at least conceptually, some identification of the type to which they belong.
3. A **scalar operator** is an operator that, when invoked, returns a scalar value (the **result** of that invocation). **D** shall provide facilities for users to define and destroy their own scalar operators (*user-defined* scalar operators). Other scalar operators shall be provided by the system (*built-in* or *system-defined* scalar operators). Let *Op* be a scalar operator. Then:
   a. *Op* shall be **read-only,** in the sense that invoking it shall cause no variables to be updated other than ones that are purely local to *Op*.
   b. Every invocation of *Op* shall denote a value ("produce a result") of the same type, the **result type**——also called the **declared type**——of *Op*. The definition of *Op* shall include a

Date and Darwen, DATABASES TYPES & THE RELATIONAL MODEL, pp. 81-92,

specification of the declared type of *Op*. That type shall be nonempty.

c. The definition of *Op* shall include a specification of the type of each parameter to *Op,* the **declared type** of that parameter. That type shall be nonempty. If parameter *P* is of declared type *T,* then, in every invocation of *Op,* the argument *A* that corresponds to *P* in that invocation shall also be of type *T,* and that argument *A* shall be **effectively assigned** to *P*. *Note:* The prescriptions of this paragraph c. shall also apply if *Op* is an update operator instead of a read-only operator (see below).

It is convenient to deal with update operators here as well, despite the fact that such operators are not scalar (nor are they nonscalar——in fact, they are not typed at all). An **update operator** is an operator that, when invoked, is allowed to update at least one variable that is not purely local to that operator. Let *V* be such a variable. If the operator accesses *V* via some parameter *P,* then that parameter *P* is **subject to update**. **D** shall provide facilities for users to define and destroy their own update operators (*user-defined* update operators). Other update operators shall be provided by the system (*built-in* or *system-defined* update operators). Let *Op* be an update operator. Then:

d. No invocation of *Op* shall denote a value ("produce a result").

e. The definition of *Op* shall include a specification of which parameters to *Op* are subject to update. If parameter *P* is subject to update, then, in every invocation of *Op,* the argument *A* that corresponds to *P* in that invocation shall be a variable specifically, and, on completion of the execution of *Op* caused by that invocation, the final value assigned to *P* during that execution shall be **effectively assigned** to *A*.

4. Let *T* be a nonempty scalar type, and let *v* be an appearance in some context of some value of type *T*. By definition, *v* has exactly one **physical representation** and one or more **possible representations** (at least one, because there is obviously always one that is the same as the physical representation). Physical representations for values of type *T* shall be specified by means of some kind of *storage structure definition language* and shall not be visible in **D**. As for possible representations:

   a. If *T* is user-defined, then at least one possible representation for values of type *T* shall be declared and thus made visible in **D**. For each possible representation *PR* for values of type *T* that is visible in **D,** a **selector** operator *S,* of declared type *T,* shall be provided with the following properties:

1. There shall be a one-to-one correspondence between the parameters of *S* and the components of *PR* (see RM Prescription 5). For definiteness, assume the parameters of *S* and the components of *PR* each constitute an ordered list of *n* elements ($n \geq 0$), such that the *i*th element in the list of parameters corresponds to the *i*th element in the list of components; then the declared types of the *i*th elements in the two lists shall be the same ($i = 1, 2, ..., n$).
2. Every value of type *T* shall be produced by some invocation of *S* in which every argument is a literal.
3. Every successful invocation of *S* shall produce some value of type *T*.

b. If *T* is system-defined, then zero or more possible representations for values of type *T* shall be declared and thus made visible in **D**. A possible representation *PR* for values of type *T* that is visible in **D** shall behave in all respects as if *T* were user-defined and *PR* were a declared possible representation for values of type *T*. If no possible representation for values of type *T* is visible in **D,** then at least one **selector** operator *S,* of declared type *T,* shall be provided with the following properties:

1. Every argument to every invocation of *S* shall be a literal.
2. Every value of type *T* shall be produced by some invocation of *S*.
3. Every successful invocation of *S* shall produce some value of type *T*.

5. Let some declared possible representation *PR* for values of scalar type *T* be defined in terms of components *C1, C2, ..., Cn* ($n \geq 0$), each of which has a name and a declared type. Let *v* be a value of type *T,* and let *PR*(*v*) denote the possible representation corresponding to *PR* for that value *v*. Then *PR*(*v*) shall be **exposed**——i.e., a set of read-only and update operators shall be provided such that:

a. For all such values *v* and for all *i* ($i = 1, 2, ..., n$), it shall be possible to "retrieve" (i.e., read the value of) the *Ci* component of *PR*(*v*). The read-only operator that provides this functionality shall have declared type the same as that of *Ci*.

b. For all variables *V* of declared type *T* and for all *i* ($i = 1, 2, ..., n$), it shall be possible to update *V* in such a way that if the values of *V* before and after the update are *v* and *v′* respectively, then the possible representations

Date and Darwen, DATABASES TYPES & THE RELATIONAL MODEL, pp. 81-92,

corresponding to *PR* for *v* and *v'* (i.e., *PR*(*v*) and *PR*(*v'*), respectively) differ in their *Ci* components.

Such a set of operators shall be provided for each possible representation declared for values of type *T*.

6. **D** shall support the **TUPLE** type generator. That is, given some heading {*H*} (see RM Prescription 9), **D** shall support use of the **generated type** TUPLE{*H*} as a basis for defining (or, in the case of values, selecting):

   a. Values of that type (see RM Prescription 9)

   b. Variables of that type (see RM Prescription 12)

   c. Attributes of that type (see RM Prescriptions 9 and 10)

   d. Components of that type within declared possible representations (see RM Prescription 5)

   e. Read-only operators of that type (see RM Prescription 20)

   f. Parameters of that type to user-defined operators (see RM Prescriptions 3 and 20)

   The generated type TUPLE{*H*} shall be referred to as a **tuple type,** and the name of that type shall be, precisely, TUPLE{*H*}. The terminology of *degree, attributes,* and *heading* introduced in RM Prescription 9 shall apply, *mutatis mutandis,* to that type, as well as to values and variables of that type (see RM Prescription 12). Tuple types TUPLE{*H1*} and TUPLE{*H2*} shall be equal if and only if {*H1*} = {*H2*}. The applicable operators shall include operators analogous to the RENAME, *project,* EXTEND, and JOIN operators of the relational algebra (see RM Prescription 18), together with tuple assignment (see RM Prescription 21) and tuple comparisons (see RM Prescription 22); they shall also include (a) a tuple selector operator (see RM Prescription 9), (b) an operator for extracting a specified attribute value from a specified tuple (the tuple in question might be required to be of degree one——see RM Prescription 9), and (c) operators for performing tuple "nesting" and "unnesting."

7. **D** shall support the **RELATION** type generator. That is, given some heading {*H*} (see RM Prescription 9), **D** shall support use of the **generated type** RELATION{*H*} as the basis for defining (or, in the case of values, selecting):

   a. Values of that type (see RM Prescription 10)

   b. Variables of that type (see RM Prescription 13)

   c. Attributes of that type (see RM Prescriptions 9 and 10)

   d. Components of that type within declared possible representations (see RM Prescription 5)

   e. Read-only operators of that type (see RM Prescription 20)

f. Parameters of that type to user-defined operators (see RM Prescriptions 3 and 20)

The generated type RELATION{*H*} shall be referred to as a **relation type,** and the name of that type shall be, precisely, RELATION{*H*}. The terminology of *degree, attributes,* and *heading* introduced in RM Prescription 9 shall apply, *mutatis mutandis,* to that type, as well as to values and variables of that type (see RM Prescription 13). Relation types RELATION{*H1*} and RELATION{*H2*} shall be equal if and only if {*H1*} = {*H2*}. The applicable operators shall include the usual operators of the relational algebra (see RM Prescription 18), together with relational assignment (see RM Prescription 21) and relational comparisons (see RM Prescription 22); they shall also include (a) a relation selector operator (see RM Prescription 10), (b) an operator for extracting the sole tuple from a specified relation of cardinality one (see RM Prescription 10), and (c) operators for performing relational "nesting" and "unnesting."

8. **D** shall support the **equality** comparison operator "=" for every type *T*. Let *Op* be an operator with a parameter *P,* let *P* be such that the argument corresponding to *P* in some invocation of *Op* is allowed to be of type *T,* and let *v1* and *v2* be values of type *T*. Then *v1* = *v2* shall evaluate to TRUE if and only if, for all such operators *Op,* two successful invocations of *Op* that are identical in all respects except that the argument corresponding to *P* is *v1* in one invocation and *v2* in the other are indistinguishable in their effect.

9. A **heading** {*H*} is a set of ordered pairs or **attributes** of the form <*A,T*>, where:

   a. *A* is the name of an **attribute** of {*H*}. No two distinct pairs in {*H*} shall have the same attribute name.

   b. *T* is the name of the **declared type** of attribute *A* of {*H*}.

   The number of pairs in {*H*}——equivalently, the number of attributes of {*H*}——is the **degree** of {*H*}.

   Now let *t* be a set of ordered triples <*A,T,v*>, obtained from {*H*} by extending each ordered pair <*A,T*> to include an arbitrary value *v* of type *T,* called the **attribute value** for attribute *A* of *t*. Then *t* is a **tuple value** (**tuple** for short) that **conforms** to heading {*H*}; equivalently, *t* is of the corresponding tuple type (see RM Prescription 6). The degree of that heading {*H*} shall be the **degree** of *t,* and the attributes and corresponding types of that heading {*H*} shall be the **attributes** and corresponding **declared attribute types** of *t*. Given a heading {*H*}, a *selector* operator, of type TUPLE{*H*}, shall be available for selecting an arbitrary tuple conforming to {*H*}; every such tuple shall be produced by some invocation

of that selector in which every argument is a literal, and every successful invocation of that selector shall produce some such tuple.

10. A **relation value** *r* (**relation** for short) consists of a *heading* and a *body,* where:

    a. The **heading** of *r* shall be a heading {*H*} as defined in RM Prescription 9; *r* **conforms** to that heading (equivalently, *r* is of the corresponding relation type——see RM Prescription 7). The degree of that heading {*H*} shall be the **degree** of *r,* and the attributes and corresponding types of that heading {*H*} shall be the **attributes** and corresponding **declared attribute types** of *r*.

    b. The **body** of *r* shall be a set *B* of tuples, all having that same heading {*H*}. The cardinality of that body shall be the **cardinality** of *r*.

    Given a heading {*H*}, a *selector* operator, of type RELATION{*H*}, shall be available for selecting an arbitrary relation conforming to {*H*}; every such relation shall be produced by some invocation of that selector in which every argument is a literal, and every successful invocation of that selector shall produce some such relation.

11. **D** shall provide facilities for users to define **scalar variables**. Each scalar variable shall be named and shall have a specified nonempty (scalar) **declared type**. Let scalar variable *V* be of declared type *T;* for so long as variable *V* exists, it shall have a value that is of type *T*. Defining *V* shall have the effect of initializing *V* to some value——either a value specified explicitly as part of the operation that defines *V,* or some implementation-defined value if no such explicit value is specified.

12. **D** shall provide facilities for users to define **tuple variables**. Each tuple variable shall be named and shall have a specified nonempty **declared type** of the form TUPLE{*H*} for some heading {*H*}. Let variable *V* be of declared type TUPLE{*H*}; then the degree of that heading {*H*} shall be the **degree** of *V,* and the attributes and corresponding types of that heading {*H*} shall be the **attributes** and corresponding **declared attribute types** of *V*. For so long as variable *V* exists, it shall have a value that is of type TUPLE{*H*}. Defining *V* shall have the effect of initializing *V* to some value——either a value specified explicitly as part of the operation that defines *V,* or some implementation-defined value if no such explicit value is specified.

13. **D** shall provide facilities for users to define **relation variables** (**relvars** for short)——both database relvars (i.e., relvars that are part of some database) and application relvars

(i.e., relvars that are local to some application). **D** shall also provide facilities for users to destroy database relvars. Each relvar shall be named and shall have a specified **declared type** of the form RELATION{*H*} for some heading {*H*}. Let variable *V* be of declared type RELATION{*H*}; then the degree of that heading {*H*} shall be the **degree** of *V,* and the attributes and corresponding types of that heading {*H*} shall be the **attributes** and corresponding **declared attribute types** of *V*. For so long as variable *V* exists, it shall have a value that is of type RELATION{*H*}.

14. Database relvars shall be either *real* or *virtual*. A **virtual relvar** *V* shall be a database relvar whose value at any given time is the result of evaluating a certain relational expression at that time; the relational expression in question shall be specified when *V* is defined and shall mention at least one database relvar. A **real relvar** shall be a database relvar that is not virtual. Defining a real relvar *V* shall have the effect of initializing *V* to some value——either a value specified explicitly as part of the operation that defines *V,* or an empty relation if no such explicit value is specified.

    Application relvars shall be either *public* or *private*. A **public relvar** shall be an application relvar that constitutes the perception of the application in question of some portion of some database. A **private relvar** shall be an application relvar that is completely private to the application in question and is not part of any database. Defining a private relvar *V* shall have the effect of initializing *V* to some value——either a value specified explicitly as part of the operation that defines *V,* or an empty relation if no such explicit value is specified.

15. By definition, every relvar shall have at least one **candidate key**. At least one such key shall be defined, either explicitly or implicitly, at the time the relvar in question is defined, and it shall not be possible to destroy all of the candidate keys of a given relvar (other than by destroying the relvar itself).

16. A **database** shall be a named container for relvars; the content of a given database at any given time shall be a set of database relvars. The necessary operators for defining and destroying databases shall not be part of **D** (in other words, defining and destroying databases shall be done "outside the **D** environment").

17. Each **transaction** shall interact with exactly one database. However, distinct transactions shall be able to interact with distinct databases, and distinct databases shall not necessarily be disjoint. Also, **D** shall provide facilities for

a transaction to define new relvars, or destroy existing ones, within its associated database (see RM Prescription 13).

18. **D** shall support the usual operators of the **relational algebra** (or some logical equivalent thereof). Specifically, it shall support, directly or indirectly, at least the operators RENAME, *restrict* (WHERE), *project,* JOIN, UNION, INTERSECT, MINUS, DIVIDEBY, EXTEND, SUMMARIZE, GROUP, and UNGROUP. All such operators shall be expressible without excessive circumlocution. **D** shall support **type inference** for relation types, whereby the type of the result of evaluating an arbitrary relational expression shall be well defined and known to both the system and the user.

19. **Relvar names** and **relation selector invocations** shall both be valid relational expressions. **Recursion** shall be permitted in relational expressions.

20. **D** shall provide facilities for users to define and destroy their own **tuple operators** (*user-defined* tuple operators) and **relational operators** (*user-defined* relational operators). Paragraphs a.-c. from RM Prescription 3 shall apply, *mutatis mutandis*.

21. **D** shall support the **assignment** operator ":=" for every type *T*. The assignment shall be referred to as a scalar, tuple, or relation (or relational) assignment according as *T* is a scalar, tuple, or relation type. Let *V* and *v* be a variable and a value, respectively, of the same type. After assignment of *v* to *V,* the equality comparison *V* = *v* shall evaluate to TRUE (see RM Prescription 8). Furthermore, all variables other than *V* shall remain unchanged, apart possibly from variables defined in terms of *V* or variables in terms of which *V* is defined or both.

    D shall also support a **multiple** form of assignment, in which several individual assignments shall be performed as a single operation. Let *MA* be the multiple assignment

    A1 , *A2* , ... , *An* ;

    (where *A1, A2, ..., An* are individual assignments, each assigning to exactly one target variable, and the semicolon marks the overall end of the operation). Then the semantics of *MA* shall be defined by the following pseudocode (Steps a.-d.):

    a. For *i* := 1 to *n,* expand any syntactic shorthands involved in *Ai*. After all such expansions, let *MA* take the form

        V1 := *X1* , *V2* := *X2* , ... , *Vz* := *Xz* ;

        for some *z* ≥ *n,* where *Vi* is the name of some variable not defined in terms of any others and *Xi* is an expression of declared type the same as that of *Vi*.

b. Let *p* and *q* ($1 \leq p < q \leq z$) be such that *Vp* and *Vq* are identical and there is no *r* (*r* < *p* or *p* < *r* < *q*) such that *Vp* and *Vr* are identical. Replace *Aq* in *MA* by an assignment of the form

Vq := WITH *Xp* AS *Vq* : *Xq*

and remove *Ap* from *MA*. Repeat this process until no such pair *p* and *q* remains. Let *MA* now consist of the sequence

U1 := *Y1* , *U2* := *Y2* , ... , *Um* := *Ym* ;

where each *Ui* is some *Vj* ($1 \leq i \leq j \leq m \leq z$).

c. For *i* := 1 to *m,* evaluate *Yi*. Let the result be *yi*.

d. For *i* := 1 to *m,* assign *yi* to *Ui*.

Note: Step b. of the foregoing pseudocode makes use of the WITH construct of **Tutorial D**. For further explanation, see Chapter 5.

22. **D** shall support certain **comparison operators,** as follows:

a. The operators for comparing scalars shall include "=", "≠", and (for ordinal types) "<", ">", etc.

b. The operators for comparing tuples shall include "=" and "≠" and shall not include "<", ">", etc.

c. The operators for comparing relations shall include "=", "≠", "⊆" ("is a subset of"), and "⊇" ("is a superset of") and shall not include "<", ">", etc.

d. The operator "∈" for testing membership of a tuple in a relation shall be supported.

In every case mentioned except "∈" the comparands shall be of the same type; in the case of "∈" they shall have the same heading. *Note:* Support for "=" for every type is in fact required by RM Prescription 8.

23. **D** shall provide facilities for defining and destroying **integrity constraints** (**constraints** for short). Let *C* be a constraint; *C* can be thought of as a boolean expression (though it might not be explicitly formulated as such), and it shall be **satisfied** if and only if that boolean expression evaluates to TRUE. No user shall ever see a state of affairs in which *C* is not satisfied. There shall be two kinds of constraints:

a. A **type** constraint shall specify the set of values that constitute a given type.

b. A **database** constraint shall specify that values of a given set of database relvars taken in combination shall be such that a given boolean expression (which shall mention no variables other than the database relvars in question)

Date and Darwen, DATABASES TYPES & THE RELATIONAL MODEL, pp. 81-92,

evaluates to TRUE. Insofar as feasible, **D** shall support **constraint inference** for database constraints, whereby the constraints that apply to the result of evaluating an arbitrary relational expression shall be well defined and known to both the system and the user.

24. Let *DB* be a database, let *DBC1, DBC2, ..., DBCn* be all of the database constraints defined for *DB* (see RM Prescription 23), and let *DBC* be any boolean expression that is logically equivalent to

    (*DBC1*) AND (*DBC2*) AND ... AND (*DBCn*) AND TRUE

    Then *DBC* is **the total database constraint** for *DB*.

25. Every database shall include a set of database relvars that constitute the **catalog** for that database. **D** shall provide facilities for assigning to relvars in the catalog.

26. **D** shall be constructed according to well-established principles of **good language design**.

RM PROSCRIPTIONS

1. **D** shall include no concept of a "relation" whose attributes are distinguishable by ordinal position. Instead, for every relation *r* expressible in **D,** the attributes of *r* shall be distinguishable by *name*.
2. **D** shall include no concept of a "relation" whose tuples are distinguishable by ordinal position. Instead, for every relation *r* expressible in **D,** the tuples of *r* shall be distinguishable by *value*.
3. **D** shall include no concept of a "relation" containing two distinct tuples *t1* and *t2* such that the comparison "*t1 = t2*" evaluates to TRUE. It follows that (as already stated in RM Proscription 2), for every relation *r* expressible in **D,** the tuples of *r* shall be distinguishable by *value*.
4. **D** shall include no concept of a "relation" in which some "tuple" includes some "attribute" that does not have a value.
5. **D** shall not forget that relations with no attributes are respectable and interesting, nor that candidate keys with no components are likewise respectable and interesting.
6. **D** shall include no constructs that relate to, or are logically affected by, the "physical" or "storage" or "internal" levels of the system.
7. **D** shall support no tuple-at-a-time operations on relvars or relations.
8. **D** shall not include any specific support for "composite" or "compound" attributes, since such functionality can more

cleanly be achieved, if desired, through the type support already prescribed.

9. **D** shall include no "domain check override" operators, since such operators are both *ad hoc* and unnecessary.
10. **D** shall not be called SQL.

OO PRESCRIPTIONS

1. **D** shall permit **compile-time type checking**.
2. If **D** supports **type inheritance,** then such support shall conform to the inheritance model defined in Part IV of this book.
3. **D** shall be **computationally complete**. That is, **D** may support, but shall not require, invocation from so-called "host programs" written in languages other than **D**. Similarly, **D** may support, but shall not require, the use of other languages for implementation of user-defined operators.
4. Transaction initiation shall be performed only by means of an explicit **"begin transaction"** operator. Transaction termination shall be performed only by means of a **"commit"** or **"rollback"** operator; commit must always be explicit, but rollback can be implicit (if and only if the transaction fails through no fault of its own). If transaction *TX* terminates with commit ("normal termination"), changes made by *TX* to the applicable database shall be committed. If transaction *TX* terminates with rollback ("abnormal termination"), changes made by *TX* to the applicable database shall be rolled back.
5. **D** shall support **nested transactions**——i.e., it shall permit a *parent* transaction *TX* to initiate a *child* transaction *TX'* before *TX* itself has terminated, in which case:
   a. *TX* and *TX'* shall interact with the same database (as is in fact required by RM Prescription 17).
   b. Whether *TX* shall be required to suspend execution while *TX'* executes shall be implementation-defined. However, *TX* shall not be allowed to terminate before *TX'* terminates; in other words, *TX'* shall be wholly contained within *TX*.
   c. Rollback of *TX* shall include the rolling back of *TX'* even if *TX'* has terminated with commit. In other words, "commit" is always interpreted within the parent context (if such exists) and is subject to override by the parent transaction (again, if such exists).
6. Let *AggOp* be an **aggregate** operator, such as SUM. If the argument to *AggOp* happens to be empty, then:
   a. If *AggOp* is essentially just shorthand for some iterated scalar dyadic operator *Op* (the dyadic operator is "+" in the case of SUM), and if an identity value exists for *Op* (the

identity value is 0 in the case of "+"), then the result of that invocation of *AggOp* shall be that identity value.

b. Otherwise, the result of that invocation of *AggOp* shall be undefined.

OO PROSCRIPTIONS

1. Relvars are not domains.
2. No database relvar shall include an attribute of type *pointer*.

RM VERY STRONG SUGGESTIONS

1. **D** should provide a mechanism according to which values of some specified candidate key (or certain components thereof) for some specified relvar are **supplied by the system**. It should also provide a mechanism according to which an arbitrary relation can be extended to include an attribute whose values (a) are unique within that relation (or within certain partitions of that relation), and (b) are once again **supplied by the system**.
2. **D** should include some declarative shorthand for expressing **referential constraints** (also known as **foreign key** constraints).
3. Let *RX* be a relational expression. By definition, *RX* can be thought of as designating a relvar, *R* say——either a user-defined relvar (if *RX* is just a relvar name) or a system-defined relvar (otherwise). It is desirable, though not always entirely feasible, for the system to be able to **infer the candidate keys** of *R,* such that (among other things):

   a. If *RX* constitutes the defining expression for some virtual relvar *R',* then those inferred candidate keys can be checked for consistency with the candidate keys explicitly defined for *R'* and——assuming no conflict——become candidate keys for *R'.*

   b. Those inferred candidate keys can be included in the information about *R* that is made available (in response to a "metaquery") to a user of **D**.

   D should provide such functionality, but without any guarantee (a) that such inferred candidate keys are not proper supersets of actual candidate keys, or (b) that such an inferred candidate key is discovered for every actual candidate key.
4. **D** should support **transition constraints**——i.e., constraints on the transitions that a given database can make from one value to another.

Date and Darwen, DATABASES TYPES & THE RELATIONAL MODEL, pp. 81-92,

5. **D** should provide some shorthand for expressing **quota queries**. It should not be necessary to convert the relation concerned into (e.g.) an array in order to formulate such a query.
6. **D** should provide some shorthand for expressing the **generalized transitive closure** operation, including the ability to specify generalized *concatenate* and *aggregate* operations.
7. **D** should provide some means for users to define their own generic **operators,** including in particular generic **relational** operators.
8. **SQL** should be implementable in **D**——not because such implementation is desirable in itself, but so that a painless migration route might be available for current SQL users. To this same end, existing SQL databases should be convertible to a form that **D** programs can operate on without error.

OO VERY STRONG SUGGESTIONS

1. Some level of **type inheritance** should be supported (in which case, see OO Prescription 2).
2. Operator definitions should be **logically distinct** from the definitions of the types of their parameters and results, not "bundled in" with those latter definitions (though the operators required by RM Prescriptions 4, 5, 8, and 21 might be exceptions in this regard).
3. **D** should support the concept of **single-level storage**.

RECENT *MANIFESTO* CHANGES

There are a number of differences between the *Manifesto* as defined in the present chapter and the version documented in this book's predecessor (reference [83]). For the benefit of readers who might be familiar with that earlier version, we summarize the main differences here.

- RM Prescription 1 has been simplified and corrected. In particular, (a) the requirement that values and variables of type *T* be operable upon solely by means of operators defined for type *T* has been deleted, since it was tautologous; (b) the references to RM Prescriptions 4 and 5 have been deleted, since they were redundant.
- Type **truth value** has been renamed type **boolean,** and the truth values *true* and *false* have been renamed TRUE and FALSE, respectively.
- RM Prescription 3 has been restructured to make it clear that scalar operators are read-only by definition, while update operators have no type at all but can update variables (arguments in particular) of any type. Paragraph a. outlaws

side effects on the part of read-only operators. Paragraph b. requires result types to be nonempty. Paragraph c. requires parameter types to be nonempty. Paragraph e. explains the semantics of parameters that are subject to update in terms of effective assignment instead of (as previously) in terms of passing by reference *vs*. passing by value.

- RM Prescription 4 now refers explicitly to nonempty types. It allows empty possreps. Also, "actual" representations are now called physical representations, and the requirement that declared possreps be defined as part of the pertinent type definition (instead of, possibly, elsewhere) has been deleted.
- Several omissions to do with selectors have been rectified: Result types (and in fact the complete semantics) of tuple and relation selectors are specified; scalar selector parameters are explained; and possreps and selectors for system-defined scalar types are specified. Also, result types are specified for the read-only operators that access possrep components.
- The phrase "at most only" has been deleted from paragraph b. of RM Prescription 5. (As it stood, that phrase rendered the paragraph vacuous, while deleting "at most" but keeping "only" would result in a prescription that could be awkward to satisfy.)
- RM Prescriptions 6 and 7 have been extended to include tuple and relation types as possible declared types for parameters and read-only operators.
- RM Prescription 11 now defines tuples in terms of tuple types instead of *vice versa*.
- Headings are now denoted {*H*} instead of *H*.
- The fact that (except for relvars) variables must have a nonempty declared type is now stated explicitly, as is the fact that they always have a value (there is no such thing as an unitialized variable). Real relvars can now be explicitly initialized.
- RM Prescription 14 has been expanded to include details of application relvars.
- Candidate key specifications can now be implicit (for virtual relvars in particular; previously we required them always to be explicit, but that was just an oversight).
- RM Prescription 18 now mentions GROUP and UNGROUP.
- RM Prescription 20 has been generalized.
- RM Prescription 21 has been clarified (and, in the case of multiple assignment, corrected).

Date and Darwen, DATABASES TYPES & THE RELATIONAL MODEL, pp. 81-92,

- RM Prescription 23 now explicitly spells out the semantics of type and database constraints (attribute and relvar constraints as such are no longer mentioned).
- RM Prescription 24 has been revised and simplified.
- RM Very Strong Suggestion 8 has been deleted (and RM Very Strong Suggestion 9 has been renumbered accordingly); very strongly suggesting the "special values" approach to missing information is (we now feel) to promote that approach more than it merits.
- OO Very Strong Suggestion 1 has been abbreviated.
- OO Very Strong Suggestions 3 and 4 have been deleted (and OO Very Strong Suggestion 5 has been renumbered accordingly); we no longer believe there are any strong arguments in favor of supporting additional "collection" type generators, over and above RELATION.

In addition to all of the foregoing, almost all of the prescriptions, proscriptions, and very strong suggestions have been reworded (in some cases extensively). However, those revisions in themselves are not intended to induce any changes in what is being described.

***** End of Chapter 4 *****

Date and Darwen, DATABASES TYPES & THE RELATIONAL MODEL, pp. 81-92,