Information Systems Vol. 15, No. 2, pp. 207–232, 1990
Printed in Great Britain. All rights reserved

0306-4379/90 $3.00 + 0.00

SCHEMA EVOLUTION AND THE RELATIONAL ALGEBRA

EDWIN MCKENZIE[1] and RICHARD SNODGRASS[2]
[1]Airforce Acquisitions Center, Bedford, Mass., U.S.A.
[2]Department of Computer Science, University of Arizona, Tucson, AZ 85721, U.S.A.

(*Received in revised form* 24 *May* 1989; *received for publication* 16 *November* 1989)

Abstract—In this paper we discuss extensions to the conventional relational algebra to support both aspects of transaction time, evolution of a database's *contents* and evolution of a database's *schema*. We define a relation's schema to be the relation's temporal *signature*, a function mapping the relation's attribute names onto their value domains, and *class*, indicating the extent of support for time. We also introduce commands to change a relation, now defined as a triple consisting of a sequence of classes, a sequence of signatures, and a sequence of states. A semantic type of system is required to identify semantically incorrect expressions and to enforce consistency constraints among a relation's class, signature, and state following update. We show that these extensions are applicable, without change, to historical algebras that support valid time, yielding an algebraic language for the query and update of temporal databases. The additions preserve the useful properties of the conventional algebra.

1. INTRODUCTION

A database's *schema* describes the *structure* of the database; the *contents* of the database must adhere to that structure [1, 2]. *Schema evolution* refers to changes to the database's schema over time. Conventional databases allow only one schema to be in force at a time, requiring *restructuring* (also termed *logical reorganization* [3]) when the schema is modified. With the advent of databases storing past states [4], it becomes desirable to accommodate multiple schemas, each in effect for an interval in the past. Schema *versioning* refers to retention of past schemas resulting from schema evolution.

In an earlier paper [5] we proposed extensions to the conventional relational algebra [6] that model the evolution of a database's contents. We did not, however, consider the evolution of a database's schema. In this paper, we provide further extensions to the conventional relational algebra that model the evolution of a database's schema. The extensions that support evolution of a database's contents are repeated here for completeness and because the extensions supporting schema evolution are best explained in concert with those earlier extensions.

1.1. Approach

Languages for database query and update exist at no less than three levels of database abstraction. At the user-interface level, calculus-based languages such as SQL are available for expressing query and update operations. At the algebraic level, the relational algebra is the formal, abstract language for expressing these same operations. Finally, at the physical level, query and update operations can be defined in terms of data structures and access strategies. In this paper we focus on language definition at the algebraic level. Our goal here is to define formally an algebraic language for database query and update that supports evolution (and versioning) of a database's schema, as well as its contents. To do so, we extend the relational algebra to handle one aspect of time in databases.

Time must be added to the underlying data model before it can be added to the relational algebra. In previous papers, we identified three orthogonal aspects of time that a database management system (DBMS) needs to support: valid time, transaction time, and user-defined time [7, 8]. *Valid time* concerns modeling time-varying reality. The valid time of, say, an event is the clock time when the event occurred in the real world, independent of the recording of that event in some database. *Transaction time*, on the other hand, concerns the storage of information in the database. The transaction time of an event is the transaction number (an integer) of the transaction that stored the information about the event in the database. *User-defined time* is an uninterpreted domain for which the DBMS supports the operations of input, output, and perhaps comparison. As its name implies, the semantics of user-defined time is provided by the user or application program. These three types of time are orthogonal in the support required of the DBMS.

In these same papers, we defined four classes of relations depending on their support for valid time and transaction time: snapshot relations, rollback relations, historical relations, and temporal relations. User-defined time, unlike valid time and transaction time, is already supported by the relational algebra, in that it is simply another domain, such as integer or character string, provided by the DBMS [9–11]. *Snapshot relations* support neither valid time nor transaction time. They model an enterprise at one particular point in time. As a snapshot relation is changed to reflect changes in the enterprise being

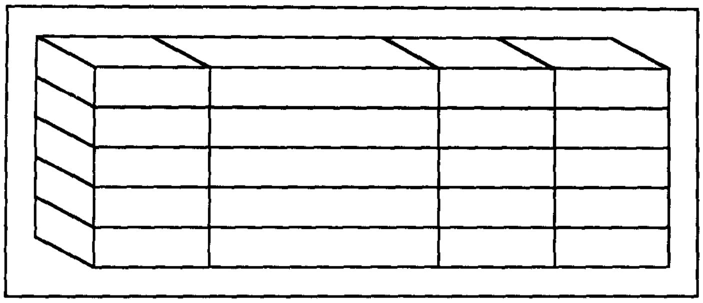
Fig. 1. Snapshot relation.

modeled, past states of the relation, representing past states of the enterprise, are discarded. A snapshot relation consists of a set of *tuples* with the same set of *attributes*, and is usually represented as a two-dimensional table with attributes as columns and tuples as rows, as shown in Fig. 1. Note that snapshot relations are exactly those relations supported by the relational algebra. Hence, for clarity, we will refer to the relational algebra hereafter as the snapshot algebra. *Rollback relations* support transaction time but do not support valid time. They may be represented as a sequence of snapshot states indexed by transaction time, as shown in Fig. 2. (Here, the last transaction deleted one tuple and appended another.) Because they record the history of database activity, rollback relations can be rolled back to one of their past snapshot states for querying.

Historical relations support valid time but do not support transaction time. They model the history, as it is best known, of an enterprise. When an historical relation is changed, however, its past state, like that of a snapshot relation, is discarded. An historical relation may be represented as a three-dimensional solid, as shown in Fig. 3. Because they record the history of the enterprise being modeled, historical relations support historical queries. They do not, however, support rollback operations. *Temporal relations* support both valid time and transaction time. They may be represented as a sequence of historical states indexed by transaction time, as shown in Fig. 4. Because they record both the history of the enterprise being modeled and the history of database activities, temporal relations support both historical queries and rollback operations.

Data models that support these four classes of relations have several important properties. First, a relation's schema can no longer be defined in terms of the relation's attributes alone; it must also include the relation's *class* (i.e. snapshot, rollback, historical, or temporal). Second, rollback and temporal re-

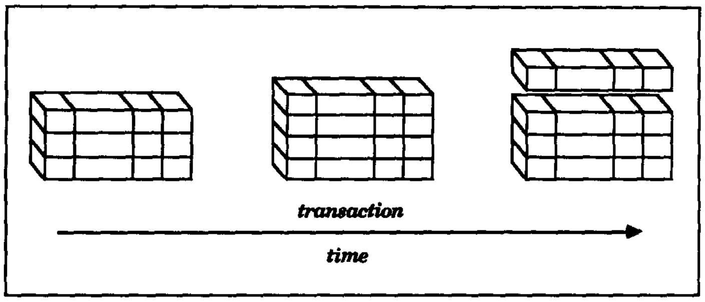

Fig. 2. Rollback relation.

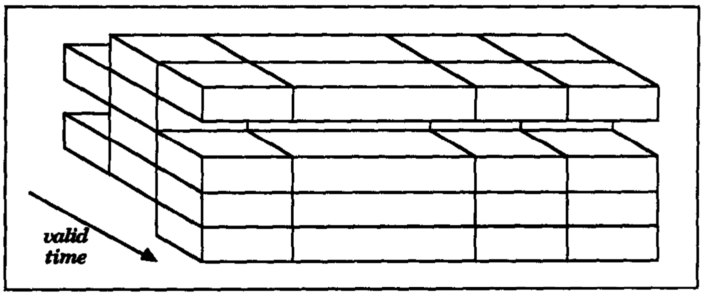

Fig. 3. Historical relation.

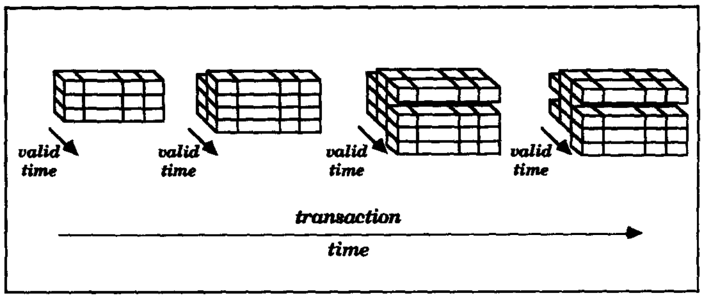

Fig. 4. Temporal relation.

lations, unlike snapshot and historical relations, are *append-only* relations. Information, once added to a rollback or temporal relation, cannot be deleted; otherwise, rollback operations could not be supported. Third, valid time and transaction time are orthogonal aspects of time. A relation may support either valid time or transaction time without supporting both. Also, the time when an enterprise changes (i.e. valid time) need not be, and usually will not be, the same as the time when the database is updated (i.e. transaction time) to reflect that change. Finally, the same measures of time need not be used for valid and transaction time. For example, a temporal relation will have a variable granularity, which changes with each update, for transaction time but could have a fixed granularity (e.g. second) for valid time.

Fortunately, since valid time and transaction time are orthogonal, they may be studied in isolation. There have already been several proposals for adding valid time to the snapshot algebra [12–21], so we will not consider valid time in detail. We focus here on extensions to support transaction time.

In a previous paper [5] we discussed extensions to the snapshot algebra to enable it to handle one aspect of transaction time: evolution of a database's contents. To handle evolution of the contents of a database containing both snapshot and rollback relations, we defined a relation to be a sequence of snapshot states, ordered by transaction number. Snapshot relations were modeled as single-element sequences while rollback relations were modeled as multi-element, append-only sequences. We also defined a database to be an ordered pair whose first component was a function from identifiers (i.e. relation names) to relations and whose second component was the transaction number of the most recently committed transaction on the database. We then augmented the algebra with a rollback operator to make past states of rollback relations available in the algebra and encapsulated this extended algebra in a language of commands for database update. Finally, we showed that the same approach could be used to extend an arbitrary historical algebra (i.e. an algebra supporting valid time) to handle evolution of the contents of a database containing both historical and temporal relations.

We now extend the relational algebra to handle the other aspect of transaction time: evolution of a database's schema. Schema evolution is associated solely with transaction time, since it defines how reality is modeled by the database. For example, a person's marital status is a (time-varying) aspect of reality, but the decision whether to record marital status, encoded in the schema, is a (time-varying) aspect of the database. Hence, we add the schema to the domain of the database states. The schema and the contents of the database combine to define the database's state.

A relation's structure can no longer be defined in terms of the relation's attributes alone; it must also be defined in terms of the relation's class. Hence, we define a relation's schema to be a pair consisting of the relation's class and a function, which we refer to as the relation's *signature*, that maps the relation's attribute names onto their value domains. (If the identification of primary keys is desirable, this would also properly go into the signature.) The relation's contents, which we refer to as the relation's *state*, always must be consistent with both the relation's class and the relation's signature.

Our model of transaction time is predicated on two assumptions. First, we assume that a database may contain snapshot, rollback, historical, and temporal relations. Second, we assume that the class and signature, as well as the contents, of each relation in the database may change over time. For example, a relation defined initially as a snapshot relation could be changed to be a historical, rollback, or temporal relation. Later, it could be changed to be a snapshot relation once again.

A model of transaction time in a database containing relations of all four classes must maintain, for each relation, its current class, signature, and state. The model also must retain, for each relation, its signature and state for those intervals during which its class was either rollback or temporal. Hence, we define a relation to be a triple consisting of a sequence of classes, a sequence of signatures, and a sequence of states, all ordered by transaction number. The class sequence records the relation's current class and intervals when the relation's class was either rollback or temporal. Similarly, the signature and state sequences record the relation's current signature and state and all changes in signature and state during intervals when the relation's class was either rollback or temporal. We also define a database state to be a function from identifiers (i.e. relation names) to relations. Finally, we define a database to be an ordered pair whose first component is a database state and whose second component is the transaction number of the most recently committed transaction on the database.

We define four commands for database update: define_relation, modify_relation, destroy, and rename_relation. The define_relation command assigns a new class and signature, along with the empty snapshot or historical state, to an undefined relation. The modify_relation command changes the current class, signature, and state of a defined relation. The destroy command is the counterpart of the define_relation command. It either physically or logically deletes from the database the current class, signature, and state of a relation, depending on the relation's class when the command is executed. The rename_relation command binds the current class, signature, and state of a relation to a new identifier. We assume that these commands execute in the context of a single, previously created database. Hence, no commands are necessary to create or delete the database. Since we are considering modeling transaction time from a functional, rather than from a performance, viewpoint, commands affecting access methods, storage mechanisms, or index maintenance are also not relevant.

Allowing a database's schema, as well as its contents, to change increases the complexity of the language. If we allow the database's schema to change, an algebraic expression that is semantically correct for the database's schema when one command executes may not be semantically correct for the database's schema when another command executes. We need a mechanism for identifying semantically incorrect algebraic expressions relative to the database's schema when each command executes and a way of ensuring that the schema and contents of the database state resulting from the command's execution are compatible. To identify semantically incorrect expressions, we introduce a *semantic type system* and augment all commands to do type-checking.

Finally, we encapsulate commands within a system of transactions to provide for both single-command and multiple-command transactions. A multiple-command transaction, like a single-command transaction, is treated as an atomic update operation, whether it changes one relation or several relations. Transactions are specified by the keywords begin_transaction and either commit_transaction or abort_transaction, the latter depending on whether the transaction commits or aborts.

This language was designed to satisfy several other objectives as well. First, the language subsumes the expressive power of the snapshot algebra. For every expression in the snapshot algebra, there is an equivalent expression in the language. Second, the language subsumes the expressive power of any arbitrary historical algebra. For every expression in an historical algebra, there is an equivalent expression in the language. Third, the language ensures that all data stored in a relation when its class was either rollback or temporal are retained permanently and are accessible via a rollback operator, even after the relation is logically deleted from the database. Fourth, commands change only a relation's class, signature, and state current at the start of a transaction. Past data, once saved, that are retained to support rollback operations, are never changed. Hence, the language accommodates implementations that use write-once-read-many (WORM) optical disks to store non-current class, signature, and state information.

We employ denotational semantics to define the semantics of the language, due to its success in formalizing operations involving side-effects, such as assignment, in programming languages [22, 23]. In defining the semantics of commands and algebraic operators, we have favored simplicity of semantics at the expense of efficient direct implementation. The language would be inefficient, in terms of storage space and execution time, if mapped directly into an implementation. However, the semantics do not preclude more efficient implementations using optimization strategies for both storage and retrieval of information. Elsewhere, we review briefly some of the techniques for efficient implementation, compatible with our semantics, that have been proposed by others [18]. We also, without loss of generality, assume that transactions are executed sequentially in a single-user environment. Our approach applies equally to environments that permit the concurrent execution of transactions as long as their concurrency control mechanisms induce a serialization of transactions.

Our language for supporting the above extensions will be the topic of the next section. Additional aspects of the rollback operators are discussed briefly in Section 3. Section 4 will review related work and compare our approach with those of others.

2. THE LANGUAGE

In this section we provide the syntax and denotational semantics of our language for database query and update. In denotational semantics, a language is described by assigning to each language construct a *denotation*—an abstract entity that models its meaning [22–25]. We chose denotational semantics to define the language because denotational semantics combines a powerful descriptive notation with rigorous mathematical theory [22], permitting the precise definition of database state. First, we define the syntax of the language. Then we define the language's semantic domains and a semantic type system for expressions. Finally, we define the semantic functions that map the language constructs onto their denotations.

2.1. Syntax

The language has three basic types of constructs: programs, commands, and expressions. A program is a sequence of one or more transactions. Both single-command and multi-command transactions are allowed. Commands occur within transactions; they change relations (e.g. define a relation, modify a relation, delete a relation). Expressions occur within commands and denote a single snapshot or historical state. We represent these three types of constructs by the syntactic categories:

PROGRAM	Category of programs
COMMAND	Category of commands
EXPRESSION	Category of expressions

We use Backus–Naur Form to specify here the syntax of programs, commands, and expressions in terms of their *immediate constituents* (i.e. the highest-level constructs that make up programs, commands, and expressions). The complete syntax of the language, including definitions of the lower-level constituents such as identifiers and snapshots states, is given elsewhere [18].

$P ::=$ begin_transaction C commit_transaction
$\quad |$ begin_transaction C abort_transaction
$\quad | P_1; P_2$

$C ::=$ define_relation$(I, Y, Z) |$
$\quad$ modify_relation(I, Y', Z', E)
$\quad |$ destroy$(I) |$ rename_relation$(I_1, I_2) | C_1, C_2$

$E ::=$ [snapshot, Z, S] $|$ [historical, Z, H] $| I$
$\quad | E_1 \cup E_2 | E_1 - E_2 | E_1 \times E_2 | \pi X(E) | \sigma F(E)$
$\quad | E_1 \hat{\cup} E_2 | E_1 \hat{-} E_2 | E_1 \hat{\times} E_2 | \hat{\pi} X(E) | \hat{\sigma} F(E)$
$\quad | \rho(I, N) | \hat{\rho}(I, N) | (E)$

$Y' ::= Y | *$

$Y ::=$ snapshot $|$ rollback $|$ historical $|$ temporal

$Z' ::= Z | *$

$Z ::= (I_{1,1} : I_{1,2}, \ldots, I_{m,1} : I_{m,2})$

where C, C_1, and C_2 range over the category *COMMAND*; E, E_1, and E_2 range over the category *EXPRESSION*; F ranges over the category *SIGMA EXPRESSION* of Boolean expressions of elements from the categories *IDENTIFIER* and *STRING* (i.e. the category of strings in an alphabet), the relational operators, and the logical operators; H ranges over the category *H-STATE* of alphanumeric representations of historical states in an arbitrary historical algebra; $I, I_{1,1}, \ldots, I_{m,2}$ range over the category *IDENTIFIER* of alphanumeric identifiers; N ranges over the category *NUMERAL* of decimal numerals; P, P_1, and P_2 range over the category *PROGRAM*; S ranges over the category *S-STATE* of alphanumeric representations of snapshot states; X ranges over the category *IDENTIFIER LIST*; Y ranges over the category *CLASS* of character strings denoting relation classes; and Z ranges over the category *SIGNATURE* of alphanumeric representations of signatures.

An expression, which evaluates to either a snapshot or historical state, may be a constant (i.e. an ordered triple consisting of a relation class, signature, and state); an identifier I, representing the current state of the relation denoted by I; or an algebraic operator on either one or two other expressions. The allowable operators include the five operators that serve to define the snapshot algebra and the operators that serve to define an arbitrary historical algebra [26]. Any operator in a given historical algebra may be included in the language as long as expressions involving that operator evaluate to a single historical state. Because historical algebras have different sets of operators, we show here only the historical counterparts to the conventional algebraic operators, simply for illustration. Each is represented as $\widehat{op}$ to distinguish it from its snapshot algebra counterpart *op*. We also have included two additional operators, a rollback operator ρ and its historical counterpart $\hat{\rho}$.

The rollback operator ρ takes two arguments, an identifier I and a transaction number N, and retrieves from the relation denoted by I the snapshot state current at the time of transaction N. Similarly, the rollback operator $\hat{\rho}$ retrieves from the relation denoted by I the historical state current at the time of transaction N.

Examples. The following are two examples of syntactically correct expressions in the language. The first is a constant and the second is an expression involving both a rollback operator and a constant. Their semantics will be specified in Sections 2.3 and 2.4. Because the historical algebras all define historical relations differently, we show in this paper only examples involving snapshot and rollback relations. Each example, however, has, for a given arbitrary historical algebra, an analogue involving historical and temporal relations.

```
[snapshot, (sname:string, class:string), (sname:"Phil", class:"junior"),
                                          (sname:"Linda", class:"senior"),
                                          (sname:"Ralph", class:"senior")]

π(sname) ( ρ(R1, 4)) × [snapshot, (course:string), (course:"English")]
```

Note that the alphanumeric representation of a signature includes both the names of attributes and the names of the attributes' value domains.□

There are four commands in the language. We present here a brief description of each command, with some examples. The semantics of commands will be defined formally in Section 2.5.

The define_relation command binds a class, a signature, and an empty relation state to an identifier *I*.

Example.

```
define_relation(R1, snapshot, (sname:string, class:string))
```

Here, the identifier R1 is defined to denote a snapshot relation with two attributes, sname and class. The contents of the relation is, by default, the empty set.□

The modify_relation command may change the current class, signature, or state of a relation. Command parameters specify the new class, signature, and state. The special symbol "*" represents, depending on context, either the current class or the current signature of a relation. It may appear as a parameter in a modify_relation command to indicate that a relation's new class (or signature) is simply the relation's current class (signature), unchanged.

Examples.

```
modify_relation(R1, *, *, [snapshot, (sname:string, class:string),
                           (sname:"Phil", class:"junior"),
                           (sname:"Linda", class:"senior"),
                           (sname:"Ralph", class:"senior")])

modify_relation(R1, *, (sname:string, course:string),
                π(sname) (R1) × [snapshot, (course:string), (course:"English")])

modify_relation(R1, rollback, *, R1)
```

The first command changes the state of the relation denoted by R1 but leaves the relation's class and signature unchanged. The second command changes the relation's signature and state, but not its class. The third command changes only the relation's class, as the expression R1 evaluates to the current state of the relation.□

The destroy command deletes, either physically or logically, the current class, signature, and state of a relation, depending on the relation's class when the command is executed. The rename_relation command renames a relation by binding its current class, signature, and state to a new identifier.

Examples.

```
destroy(R1)

rename_relation(R2, R1)
```

Here we first delete the relation denoted by R1 and then rename the relation denoted by R2 as R1.□

Programs in the language contain two types of transactions, committed transactions and aborted transactions. Committed transactions are transactions which the user initiates that eventually commit. Aborted transactions are transactions which the user initiates that for some reason, dictated either by the user or by the system, abort rather than commit. The semantics of programs will be defined formally in Section 2.6.

2.2. Semantic domains

A program denotes the database resulting from the execution of one or more transactions, in order, on an empty database. By defining the database that results from the execution of an arbitrary sequence of transactions, we specify the semantics of that transaction sequence, and hence the semantics of the language. In this section, we will define formally the flat domain (i.e. a domain with a trivial partial ordering [27]) of databases; later sections will provide the connection between the syntactic category of programs and the semantic domain of databases. All domains introduced are flat domains and the notation $\{\dots\}$ is used to represent flat domains.

Assume that we are given the domain $D = \{D_1, \dots, D_e\}$, where each domain D_u, $1 \leqslant u \leqslant e$, is an arbitrary, non-empty, finite or denumerable set. Then, we can define the following semantic domains:

$$TRANSACTION\ NUMBER = \{0, 1, \dots\}$$

A transaction number is a non-negative integer that identifies a transaction that changes the database. The transaction number assigned to a transaction can be viewed as that transaction's time-stamp.

$$RELATION\ CLASS = \{\text{UNDEFINED, SNAPSHOT, ROLLBACK, HISTORICAL, TEMPORAL}\}$$

A relation is either undefined or defined to be a snapshot, rollback, historical, or temporal relation.

$$RELATION\ SIGNATURE = IDENTIFIER \rightarrow [D + \{\text{UNBOUND}\}]$$

where the notation "+" on domains means the disjoint union of domains. A relation's signature is a function that maps identifiers either onto a domain

D_u, $1 \leqslant u \leqslant e$ or onto UNBOUND. If a signature maps an identifier onto UNBOUND, then the identifier is unbound in that signature (i.e. it is associated with no domain). If, however, a signature maps an identifier onto a domain, then that mapping defines an attribute.

SNAPSHOT STATE = Domain of all semantically correct snapshot states (sets of m-tuples), as defined in the snapshot algebra [28], for elements of the domain *RELATION SIGNATURE* and the domain $\{D_1 + \cdots + D_e\}$, where $\varnothing$ is the empty snapshot state. Hence, a snapshot state s on a relation signature z is a finite set of mappings from $\{I | z(I) \neq \text{UNBOUND}\}$ to D, with the restriction that for each mapping $st \in s$, $st(I) \in z(I)$.

HISTORICAL STATE = Domain of all semantically correct historical states as defined in an arbitrary historical algebra (e.g. as defined in [12–21]).

$$\boxed{\begin{aligned} \mathit{RELATION} = {} & [\mathit{RELATION\ CLASS} \times \mathit{TRANSACTION\ NUMBER} \times \\ & \qquad [\mathit{TRANSACTION\ NUMBER} + \{-\}]]^{+} \times \\ & [\mathit{RELATION\ SIGNATURE} \times \mathit{TRANSACTION\ NUMBER}]^{*} \times \\ & [[\mathit{SNAPSHOT\ STATE} \times \mathit{TRANSACTION\ NUMBER}] + \\ & \ [\mathit{HISTORICAL\ STATE} \times \mathit{TRANSACTION\ NUMBER}]]^{*} \end{aligned}}$$

where the special element "–" stands for the *present* time. A relation is thus an ordered triple consisting of (1) a sequence of (relation class, transaction number, transaction number or "–") triples, (2) a sequence of (relation signature, transaction number) pairs, and (3) a sequence of (relation state, transaction number) pairs.

Relations are dynamic objects whose class, signature, and state are all allowed to change over time. For example, a relation defined initially as a snapshot relation could be modified to be a historical, rollback, or temporal relation. Later, the relation could be modified to be a snapshot relation once again. Every relation always has at least one element in its class sequence, the last element recording the relation's current class (i.e. undefined, snapshot, rollback, or temporal). Any other elements in the sequence record intervals when the relation's class was either rollback or temporal.

A relation's signature (state) sequence will be empty only if the relation is currently undefined and it was never a rollback or temporal relation. If a relation is currently other than undefined, there is at least one element in its signature (state) sequence, the last element recording the relation's current signature (state). Any other elements in the sequence record the signature (state) of the relation when its class was either rollback or temporal.

The transaction-number components of all elements, but the last element, in a relation's class sequence can be viewed as time-stamps defining a fixed, closed interval during which the element's class component was the relation's class. In contrast, the third component of the last element in the sequence is always "–"; it is used to define an interval of dynamic length that always extends to the present. The transaction-number component of each element in a relation's signature (state) sequence can be viewed as a time-stamp indicating when the element's signature (state) was entered into the database and became the relation's current signature (state). Since we assume that database changes occur sequentially, the transaction-number components of a signature (state) sequence, while not necessarily consecutive, will be nevertheless strictly increasing. Thus, we can interpolate on the transaction-number component of elements in a relation's signature (state) sequence to determine the signature (state) of the relation at any time its class was either rollback or temporal.

Example. The following is a sample relation. For notational convenience in this and later examples, we show only the attribute portion of a signature (i.e. the partial function from attribute names to value domains). Each signature maps all identifiers not shown onto UNBOUND. Also for notational convenience, we assume the natural mapping from attribute names onto attribute values for each tuple (e.g. (ename→"Phil", ssn→250861414) is the first tuple shown at transaction 4).

class	*signature*	*state*
⟨(ROLLBACK, 2, 6),	⟨((sname→string, ssn→integer), 2),	⟨(∅, 2),
		({("Phil", 250861414), ("Linda", 147894290), ("Ralph", 459326889)}, 4),
	((sname→string, class→string), 5),	({("Phil", "junior"), ("Linda", "senior"), ("Ralph", "senior")}, 5),
(SNAPSHOT, 8,–)	((ssn→integer, class→string), 8)	({(250861414, "junior"), (147894290, "senior"), (459326889, "senior")}, 8)⟩
⟩	⟩	

The relation shown here was defined to be a rollback relation by transaction 2 and remained a rollback relation through transaction 6. While the relation was a rollback relation, all changes to its signature and state were recorded; its state was changed by transaction 4 and both its signature and its state were changed by transaction 5. Transaction 7 redefined the relation's class as SNAPSHOT and the relation was last updated as a snapshot relation by transaction 8. Only when a relation's current class is either rollback or temporal is the relation treated as an append-only relation. In all other cases, updates cause outdated information to be discarded. Hence, the lack of information about the relation's class, signature, and state before transaction 2 and at transaction 7 implies that the relation was either undefined or a snapshot or historical relation at those times. Note that this relation can be rolled back only to transactions 2 through 6. Also note that the last element in the class sequence defines the relation to be a snapshot relation from transaction 8 to the present.□

DATABASE STATE = IDENTIFIER → RELATION

A database state is a function that maps each identifier onto a relation. If an identifier *I* is mapped onto a relation whose current class is UNDEFINED, then *I* denotes an undefined relation. In the empty database state, all identifiers map onto undefined relations (i.e. (⟨(UNDEFINED, 0, –)⟩, ⟨ ⟩, ⟨ ⟩)).

DATABASE = DATABASE STATE × TRANSACTION NUMBER

A database is an ordered pair consisting of a database state and the transaction number assigned to the most recently committed transaction on the database state (i.e. the last transaction to cause a change to the database state).

2.3. *A semantic type system for expressions*

Before specifying the semantics of the expressions defined syntactically in Section 2.1, we introduce a *semantic type system* for expressions. All syntactically correct expressions are not necessarily semantically correct. An expression is semantically correct, with respect to a database state and a command, only if its evaluation on the database state during the command's execution produces either a snapshot or a historical state. Also, if the expression contains a rollback operator, it must be consistent with the class and signature of the relation being rolled back at the time of the transaction to which the relation is rolled back. Because the class and signature, as well as the state, of a relation are allowed to change over time, the semantic correctness of expressions also can vary over time. Hence, expressions that are semantically correct on a database state when one command is executed may not be semantically correct on the same database state when a subsequent command is executed (although the correctness of rollback operations to existing states will be unaffected by subsequent commands).

The semantic type system defined here allows us to do expression type-checking independent of expression evaluation. In Section 2.4, where we define the semantics of expressions, we will use the type system to restrict evaluation of expressions to semantically correct expressions only. Hence, any future implementation of the language can avoid the unnecessary cost associated with attempted evaluation of semantically incorrect expressions. The type system will also be used to define the semantics of commands so that commands whose execution would result in an incompatibility among a relation's class, signature, and state will never be executed. Also, separation of semantic type-checking and evaluation of expressions simplifies the formal definitions of the semantics of both expressions and commands. Note that while semantic type-checking and evaluation of some expressions (i.e. those expressions involving constant expressions and rollback operators that roll back a relation prior to the query analysis time) can be done when a query is analyzed, most semantic type-checking and expression evaluation will have to be done when the query is executed.

Semantically correct expressions evaluate to either a single snapshot state or a single historical state. We define a snapshot state's type to be an ordered pair whose first component is SNAPSHOT and whose second component is the state's signature. Similarly, we define a historical state's type to be an ordered pair whose first component is HISTORICAL and whose second component is the state's signature. A semantically correct expression's type is therefore the class and signature of the relation state resulting from the expression's evaluation and two expressions are said to be of the same type if and only if they evaluate to either snapshot or historical states on the attributes of the same signature.

We use the semantic function **T** to specify an expression's type. A semantic function is simply a function that maps a language construct onto its denotation, or meaning. **T** defines an expression as a function that maps a database state and a transaction number onto either an ordered pair or TYPEERROR, depending on whether the expression is a semantically correct expression on the database state when a command in the transaction assigned the transaction number is executed. The ordered pair will have as its first component either SNAPSHOT or HISTORICAL and as its second component the signature of the relation state that the expression represents. Hence, **T** defines the type denotation of expressions:

$$\mathbf{T}: \mathit{EXPRESSION} \rightarrow [[\mathit{DATABASE\ STATE} \times \mathit{TRANSACTION\ NUMBER}] \rightarrow [[\{\textsc{snapshot}, \textsc{historical}\} \times \mathit{RELATION\ SIGNATURE}] + \{\textsc{typeerror}\}]]$$

The result of type-checking a syntactically correct expression is the class and signature of the relation state that the expression represents if the expression is semantically correct and an error if the expression is semantically incorrect. An expression's type may depend on a database state's contents. The type of an expression involving a rollback operator also depends on the transaction number of the transaction in which the command containing the expression occurs. Hence, a database state and transaction number together define the environment in which type-checking is performed.

Before defining the semantic function **T**, we describe informally several functions used in its definition. Formal definitions for these auxiliary functions appear elsewhere [18].

N is a semantic function that maps the syntactic category *NUMERAL* of decimal numerals into the semantic domain *INTEGER* of integers.

S is a semantic function that maps each alphanumeric representation of a snapshot state in the syntactic category *S-STATE* onto its corresponding snapshot state in the semantic domain *SNAPSHOT STATE*, if it denotes a valid snapshot state on a given signature. Otherwise, **S** maps the snapshot state onto ERROR.

VALIDF is a semantic function that maps the alphanumeric representation of a Boolean predicate in the syntactic category *SIGMA EXPRESSION* onto the Boolean value TRUE or FALSE, to indicate whether the predicate is a valid Boolean predicate for the selection operator σ (or $\hat{\sigma}$) and a given signature.

VALIDX is a semantic function that maps the alphanumeric representation of a list of identifiers in the syntactic category *IDENTIFIER LIST* onto the Boolean value TRUE or FALSE, to indicate whether the identifiers denote a valid subset of the attributes in a given signature.

X is a semantic function that maps the alphanumeric representation of a list of identifiers in the syntactic category *IDENTIFIER LIST* onto an element in $\mathscr{P}$(*IDENTIFIER*), the power set of *IDENTIFIER*, if the identifiers denote a valid subset of the attributes in a given signature. Otherwise, **X** maps the list onto ERROR.

Y is a semantic function that maps each character string in the syntactic category *CLASS* onto the relation class that it denotes in the semantic domain *RELATION CLASS*.

Z is a semantic function that maps each alphanumeric representation of a relational signature in the syntactic category *SIGNATURE* onto its corresponding relational signature in the semantic domain *RELATION SIGNATURE*.

FindClass maps a relation onto the class component of the relation's class sequence whose first transaction-number component is less than or equal to a given transaction number and whose second transaction-number component is greater than or equal to the transaction number. If no such element exists in the sequence, then *FindClass* returns ERROR.

FindSignature maps a relation onto the signature component of the element in the relation's signature sequence having the largest transaction-number component less than or equal to a given transaction number, if *FindClass* does not return an error for the same transaction number. If *FindClass* returns an error or no such element exists in the sequence, then *FindSignature* returns ERROR.

LastClass maps a relation onto the class component of the last element in the relation's class sequence. If the sequence is empty, *LastClass* returns ERROR.

LastSignature maps a relation onto the signature component of the last element in the relation's signature sequence. If the relation's signature sequence is empty, *LastSignature* returns ERROR.

We now define formally the semantic function **T** for each kind of expression. For this and later definitions of semantic functions, let e be the number of value domains D_u, $1 \leqslant u \leqslant e$, and let d range over the domain *DATABASE STATE*, z, z_1, and z_2 range over the domain *RELATION SIGNATURE*, and tn range over the domain *TRANSACTION NUMBER*.

$$\begin{aligned} \mathbf{T}[\![\text{[snapshot}, Z, S]]\!](d, tn) = {} & \text{if } (\mathbf{Z}[\![Z]\!] \neq \textsc{error} \wedge \mathbf{S}[\![S]\!]\mathbf{Z}[\![Z]\!] \neq \textsc{error}) \\ & \text{then } (\textsc{snapshot}, \mathbf{Z}[\![Z]\!]) \\ & \text{else } \textsc{typeerror} \end{aligned}$$

If a constant expression represents a snapshot state on a signature, the expression's type is the ordered pair whose first component is SNAPSHOT and whose second component is the snapshot state's signature. Otherwose, evaluation of the expression's type results in an error.

Example. For this and later examples in Section 2, assume that we are given the database (DS, 8) where the database state DS maps the identifier R1 onto the relation shown in the example of Section 2.2.

$$
\begin{aligned}
\mathbf{T}[\![&[\text{snapshot}, (\text{sname:string}, \text{class:string}), (\text{sname:"Phil"}, \text{class:"junior"}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{sname:"Linda"}, \text{class:"senior"}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{sname:"Ralph"}, \text{class:"senior"})]\\
]\!](&\text{DS}, 9) = (\text{SNAPSHOT}, (\text{sname}\rightarrow\text{string}, \text{class}\rightarrow\text{string}))
\end{aligned}
$$

Here we assume that type-checking is being performed as part of transaction 9. Note, however, that the database state is not consulted to determine the constant expression's type; the expression's type is independent of the database state. Actually, the only expression whose type depends directly on the database state are identifiers and expressions involving the rollback operators.□

Evaluation of a snapshot constant's type produces an error if and only if the expression does not represent a snapshot state on a signature. As we will

$$
\begin{aligned}
&\mathbf{T}[\![E_1 \cup E_2]\!](d, tn) = \text{if } \mathbf{T}[\![E_1]\!](d, tn) = \mathbf{T}[\![E_2]\!](d, tn) = (\text{SNAPSHOT}, z)\\
&\qquad\qquad\qquad\qquad\quad \text{then } \mathbf{T}[\![E_1]\!](d, tn)\\
&\qquad\qquad\qquad\qquad\quad \text{else } \text{TYPEERROR}\\
&\mathbf{T}[\![E_1 - E_2]\!](d, tn) = \text{if } \mathbf{T}[\![E_1]\!](d, tn) = \mathbf{T}[\![E_2]\!](d, tn) = (\text{SNAPSHOT}, z)\\
&\qquad\qquad\qquad\qquad\quad \text{then } \mathbf{T}[\![E_1]\!](d, tn)\\
&\qquad\qquad\qquad\qquad\quad \text{else } \text{TYPEERROR}\\
&\mathbf{T}[\![E_1 \times E_2]\!](d, tn) =\\
&\qquad \text{if } (\mathbf{T}[\![E_1]\!](d, tn) = (\text{SNAPSHOT}, z_1) \wedge \mathbf{T}[\![E_2]\!](d, tn) = (\text{SNAPSHOT}, z_2)\\
&\qquad\qquad \wedge \forall I, I \in \mathit{IDENTIFIER}, (z_1(I) = \text{UNBOUND} \vee z_2(I) = \text{UNBOUND}))\\
&\qquad \text{then } (\text{SNAPSHOT}, \{(I, D_u) | (1 \leqslant u \leqslant e \wedge ((I, D_u) \in z_1 \vee (I, D_u) \in z_2)\}\\
&\qquad\qquad \cup \{(I, \text{UNBOUND}) | I \in \mathit{IDENTIFIER} \wedge (I, \text{UNBOUND}) \in z_1\\
&\qquad\qquad\qquad\qquad \wedge (I, \text{UNBOUND}) \in z_2\})\\
&\qquad \text{else } \text{TYPEERROR}\\
&\mathbf{T}[\![\pi X(E)]\!](d, tn) =\\
&\qquad \text{if } (\mathbf{T}[\![E]\!](d, tn) = (\text{SNAPSHOT}, z) \wedge \mathbf{VALIDX}[\![X]\!]z)\\
&\qquad \text{then } (\text{SNAPSHOT}, \{(I, D_u) | I \in \mathbf{X}[\![X]\!]z \wedge 1 \leqslant u \leqslant e \wedge (I, D_u) \in z\}\\
&\qquad\qquad\qquad \cup \{(I, \text{UNBOUND}) | I \notin \mathbf{X}[\![X]\!]z \wedge I \in \mathit{IDENTIFIER}\})\\
&\qquad \text{else } \text{TYPEERROR}\\
&\mathbf{T}[\![\sigma F(E)]\!](d, tn) = \text{if } (\mathbf{T}[\![E]\!](d, tn) = (\text{SNAPSHOT}, z) \wedge \mathbf{VALIDF}[\![F]\!]z)\\
&\qquad\qquad\qquad\qquad \text{then } \mathbf{T}[\![E]\!](d, tn)\\
&\qquad\qquad\qquad\qquad \text{else } \text{TYPEERROR}
\end{aligned}
$$

see in Section 2.4, evaluation of a constant expression's type produces an error under exactly the same conditions that evaluation of the expression produces an error. This relationship between a constant expression's type and value is both a necessary and a sufficient condition to ensure that the evaluation of any expression will result in an error when evaluation of the expression's type results in an error.

$$
\begin{aligned}
\mathbf{T}[\![I]\!](d, tn) = \text{if } & (\mathit{LastClass}(d(I)) = \text{SNAPSHOT}\\
& \vee \mathit{LastClass}(d(I)) = \text{ROLLBACK})\\
\text{then } & (\text{SNAPSHOT}, \mathit{LastSignature}(d(I)))\\
\text{else if } & (\mathit{LastClass}(d(I)) = \text{HISTORICAL}\\
& \vee \mathit{LastClass}(d(I)) = \text{TEMPORAL})\\
\text{then } & (\text{HISTORICAL}, \mathit{LastSignature}(d(I)))\\
\text{else } & \text{TYPEERROR}
\end{aligned}
$$

where the notation $d(I)$ stands for the relation denoted by the identifier I in the database state d. The type of an expression I is the ordered pair whose first component is SNAPSHOT if I's current class is either snapshot or rollback and HISTORICAL if its current class is either historical or temporal. The ordered pair's second component is always I's current signature. An error occurs if the relation is currently undefined.

Example.

$$
\begin{aligned}
&\mathbf{T}[\![\text{R1}]\!](\text{DS}, 9)\\
&\quad = (\text{SNAPSHOT}, (\text{ssn}\rightarrow\text{integer}, \text{class}\rightarrow\text{string})).\square
\end{aligned}
$$

The type of an expression involving one of the five basic snapshot operators is an ordered pair whose first component is SNAPSHOT and whose second component is the signature of the relation state produced when the expression is evaluated, if two conditions are met. The first component of the type of all subexpressions must be SNAPSHOT and the second component of the type of all subexpressions must be a signature satisfying any restrictions placed on the signatures of relation states in corresponding expressions in the snapshot algebra. For example, our definitions of union and difference require that the signatures for E_1 and E_2 be identical while our definition of Cartesian product requires that the attributes defined by the signatures for E_1 and E_2 be disjoint. (Note that we can eliminate this last restriction and effectively allow the Cartesian product of snapshot states on arbitrary signatures through the introduction of a simple attribute renaming operator [28] into the language.) If either condition is not met, evaluation of the expression's type results in an error.

$$\mathbf{T}[\![\rho(I, N)]\!](d, tn) = \text{if } \mathbf{N}[\![N]\!] < tn \wedge \mathit{FindClass}(d(I), \mathbf{N}[\![N]\!]) = \text{ROLLBACK} \\ \text{then } (\text{SNAPSHOT}, \mathit{FindSignature}(d(I), \mathbf{N}[\![N]\!])) \\ \text{else } \text{TYPEERROR}$$

A rollback expression's type is the ordered pair whose first component is SNAPSHOT and whose second component is the signature of the relation denoted by I when transaction $\mathbf{N}[\![N]\!]$ was processed, if the relation was a rollback relation at the time. Otherwise, evaluation of the expression's type results in an error. Because we assume sequential transaction processing, tn is the transaction number of the one active transaction and all committed transactions have transaction numbers less than tn. Hence, we allow rollback only to committed transactions.

Examples.

$$\mathbf{T}[\![\rho(\text{R1}, 4)]\!](\text{DS}, 9) = (\text{SNAPSHOT}, (\text{sname}\rightarrow\text{string}, \text{ssn}\rightarrow\text{integer}))$$

$$\mathbf{T}[\![\pi(\text{sname})(\rho(\text{R1}, 4))]\!](\text{DS}, 9) = (\text{SNAPSHOT}, (\text{sname}\rightarrow\text{string}))$$

$$\mathbf{T}[\![\pi(\text{sname})(\rho(\text{R1}, 4)) \times [\text{snapshot}, (\text{course:string}), (\text{course:"English"})] \\ ]\!](\text{DS}, 9) = (\text{SNAPSHOT}, (\text{sname}\rightarrow\text{string}, \text{course}\rightarrow\text{string})).\square$$

The semantic function **T** for expressions involving historical operators follows directly. The type denotations for these expressions are identical to those for expressions involving snapshot operators, except that HISTORICAL and TEMPORAL are substituted for SNAPSHOT and ROLLBACK, respectively.

Finally, we present the definition of the semantic function **T** for the last expression construct, which is used to group subexpressions.

$$\mathbf{T}[\![(E)]\!](d, tn) = \mathbf{T}[\![E]\!](d, tn)$$

2.4. Expressions

The semantic function **E** defines the denotation of expressions. **E** defines an expression as a function that maps a database state and a transaction number onto either a snapshot state (i.e. an element of the *SNAPSHOT STATE* semantic domain), a historical state (i.e. an element of the *HISTORICAL STATE* semantic domain), or ERROR.

$$\mathbf{E}: \mathit{EXPRESSION} \rightarrow [[\mathit{DATABASE\ STATE} \times \mathit{TRANSACTION\ NUMBER}] \rightarrow \\ [\mathit{SNAPSHOT\ STATE} + \mathit{HISTORICAL\ STATE} + \{\text{ERROR}\}]]$$

If an expression is a semantically correct expression on a database state, expression evaluation on the database state produces either a snapshot state or a historical state. Otherwise, expression evaluation produces an error. The environment for expression evaluation, a database state and the transaction number of the active transaction, is the same as that for expression type-checking. Note that expression evaluation has no side-effect; it leaves the database state unchanged.

Before defining the semantic function **E**, we describe informally additional auxiliary functions used in **E**'s definition. Formal definitions appear elsewhere [18].

FindState maps a relation onto the state component of the element in the relation's state sequence having the largest transaction-number component less than or equal to a given transaction number, if *FindClass* does not return an error for the same transaction number. If *FindClass* returns an error or no such element exists in the sequence, then *FindState* returns ERROR.

LastState maps a relation onto the state component of the last element in the relation's state sequence. If the relation's state sequence is empty, *LastState* returns ERROR.

We now define formally the semantic function **E** for each kind of expression allowed in the language:

$$\mathbf{E}[\![[\text{snapshot}, Z, S]]\!](d, tn) = \text{if } \mathbf{T}[\![[\text{snapshot}, Z, S]]\!](d, tn) \neq \text{TYPEERROR} \\ \text{then } \mathbf{S}[\![S]\!]\mathbf{Z}[\![Z]\!] \\ \text{else } \text{ERROR}$$

Example.

$$\mathbf{E}[\![[\text{snapshot}, (\text{sname:string}, \text{class:string}), (\text{sname:"Phil"}, \text{class:"junior"}), \\ (\text{sname:"Linda"}, \text{class:"senior"}), \\ (\text{sname:"Ralph"}, \text{class:"senior"})] \\ ]\!](\text{DS}, 9) = \{(\text{"Phil"}, \text{"junior"}), (\text{"Linda"}, \text{"senior"}), (\text{"Ralph"}, \text{"senior"})\}.\square$$

$$\mathbf{E}[\![I]\!](d, tn) = \text{if } \mathbf{T}[\![I]\!](d, tn) \neq \text{TYPEERROR then } \mathit{LastState}(d(I)) \text{ else ERROR}$$

An identifier expression, if semantically correct, always evaluates to the current state of the relation denoted by I.

Example.

$$\mathbf{E}[\![\mathrm{R1}]\!](\mathrm{DS}, 9) = \{(250861414, \text{“junior”}), (147894290, \text{“senior”}), (459326889, \text{“senior”})\}.\square$$

$$\begin{aligned}\mathbf{E}[\![E_1 \cup E_2]\!](d, tn) = \text{if } & \mathbf{T}[\![E_1 \cup E_2]\!](d, tn) \neq \text{TYPEERROR}\\ \text{then } & \mathbf{E}[\![E_1]\!](d, tn) \cup \mathbf{E}[\![E_2]\!](d, tn)\\ \text{else } & \text{ERROR}\end{aligned}$$

The definitions of **E** for the other four snapshot operators are analogous to that for the union operator. For each of these operators, the denotation of a semantically correct expression containing the operator is defined as the standard snapshot operator over the denotation of the argument(s) to that operator.

$$\begin{aligned}\mathbf{E}[\![\rho(I, N)]\!](d, tn) = \text{if } & \mathbf{T}[\![\rho(I, N)]\!](d, tn) \neq \text{TYPEERROR}\\ \text{then } & \mathit{FindState}(d(I), \mathbf{N}[\![N]\!])\\ \text{else } & \text{ERROR}\end{aligned}$$

A semantically correct rollback expression evaluates to the snapshot state of the relation denoted by I at the time of transaction $\mathbf{N}[\![N]\!]$. The rollback operator always rolls a relation backward, but never forward, in time. Because transactions always update the database as they are executed, it is impossible to roll a relation forward in time. Although relations can't be rolled forward in time, our orthogonal treatment of valid and transaction time provides support for both *retroactive* changes and *postactive* changes (i.e. changes that will occur in the future) [7]. Recall from the definition of the semantic function **T** that a rollback expression is semantically correct only if the relation was a rollback relation when the transaction was processed.

Examples.

$$\mathbf{E}[\![\rho(\mathrm{R1}, 4)]\!](\mathrm{DS}, 9) = \{(\text{“Phil”}, 250861414), (\text{“Linda”}, 147894290), (\text{“Ralph”}, 459326889)\}$$

$$\mathbf{E}[\![\pi(\mathrm{sname})(\rho(\mathrm{R1}, 4))]\!](\mathrm{DS}, 9) = \{(\text{“Phil”}), (\text{“Linda”}), (\text{“Ralph”})\}$$

$$\mathbf{E}[\![\pi(\mathrm{sname})(\rho(\mathrm{R1}, 4)) \times [\mathrm{snapshot}, (\mathrm{course{:}string}), (\mathrm{course{:}\text{"English"}})]\ ]\!](\mathrm{DS}, 9) = \{(\text{“Phil”}, \text{“English”}), (\text{“Linda”}, \text{“English”}), (\text{“Ralph”}, \text{“English”})\}.\square$$

The semantic function **E** for expressions involving historical operators follows directly. the denotations for these expressions are identical to those for expressions involving snapshot operators, except that HISTORICAL and TEMPORAL are substituted for SNAPSHOT and ROLLBACK, respectively.

We now present the definition of the semantic function **E** for the expression construct that groups subexpressions.

$$\mathbf{E}[\![(E)]\!](d, tn) = \mathbf{E}[\![E]\!](d, tn)$$

2.5. Commands

The semantic function **C** defines the denotation of commands defined syntactically in Section 2.1. **C** defines a command as a function that maps a database state and a transaction number onto a database state and a status code. Execution of a semantically correct command produces a new database state and the status code OK, indicating that the command was successfully executed. Execution of a semantically incorrect command produces the original database state unchanged and the status code ERROR, indicating that the command could not be executed.

$$\mathbf{C}\colon \mathit{COMMAND} \to [[\mathit{DATABASE\ STATE} \times \mathit{TRANSACTION\ NUMBER}] \to [\mathit{DATABASE\ STATE} \times \{\text{OK}, \text{ERROR}\}]]$$

The environment for command execution is the same as that for expression type-checking and evaluation, a database state and the transaction number of the active transaction (i.e. the transaction in which the command being executed occurs). A command produces a new database state from the given database by changing a relation.

We use semantic type-checking of expressions in the definition of **C** to restrict evaluation of expressions to semantically correct expressions only. We also incorporate error-checking, based on the type system for expressions, into **C**'s definition to guarantee consistency among a relation's class, signature, and state following update. Error-checking ensures that commands actually change relations only when the change would result in a relation with compatible class, signature, and state. Commands whose execution would result in an inconsistency among a relation's class, signature, and state are effectively ignored (i.e. they do not alter the database state).

Before defining the semantic function **C**, we describe informally several additional auxiliary functions used in its definition. The formal definition for

MSoT appears in Appendix B. Formal definitions for the other functions appear in [18].

Y′ is the same as the semantic function **Y** with the exception that it maps the special symbol $*$ onto a relation's current class.

Z′ is the same as the semantic function **Z** with the exception that it maps the special symbol $*$ onto a relation's current signature.

Consistent is a Boolean function that determines whether a class and signature are consistent with an expression's type.

MSoT (*m*odified *s*tart *o*f *t*ransaction) is a function that maps a relation and a transaction number onto the history of the relation as a rollback or temporal relation prior to the start of the transaction assigned the transaction number. We refer to this history as the relation's MSoT for that transaction. The significance of *MSoT* will become apparent when we discuss multiple-command transactions.

Example. Again assume, as in earlier examples, that we are given the database (DS, 8) where the database state component maps the identifier R1 onto the relation shown in the example in Section 2.2.

MSoT(R1, 9) =

class	*signature*	*state*
⟨(ROLLBACK, 2, 6)	⟨((sname→string, ssn→integer), 2),	⟨(∅, 2),
		({("Phil", 250861414), ("Linda", 147894290), ("Ralph", 459326889)}, 4),
	((sname→integer, class→string), 5)	({("Phil", "junior"), ("Linda", "senior"), ("Ralph", "senior")}, 5)⟩
⟩	⟩	

In this example, *MSoT* retains R1's history as a rollback relation prior to transaction 9. Although R1's current class, signature, and state were recorded before the start of transaction 9, they have been discarded because they are not part of R1's history as a rollback relation. If, however, the last element in R1's class sequence had been (ROLLBACK, 8,–) then R1's current class, signature, and state also would have been retained. In this case, *MSoT* simply would have changed the second transaction-number component of the last element in R1's class sequence to 8 to indicate that the resulting relation only records R1's history as a rollback relation through transaction 8. If R1 had never been a rollback or temporal relation, then *MSoT* would have mapped R1 onto (⟨ ⟩, ⟨ ⟩, ⟨ ⟩).□

Expand replaces the second transaction-number component in the last element of a relation's MSoT class sequence with the special element "–". *Expand* has the effect of making the length of the interval for the class component of this element dynamic, extending to the present.

NewSignature maps a relation's MSoT and a (signature, transaction number) pair onto the empty sequence, if the signature in the last element of the relation's MSoT signature sequence is equal to the signature in the (signature, transaction number) pair, or a one-element sequence containing the (signature, transaction number) pair, otherwise.

NewState maps a relation's MSoT, a (relation state, transaction number) pair, and a (class, signature) pair onto the empty sequence, if the class and signature in the last elements of the relation's MSoT class and signature sequences are consistent with the (class, signature) pair and if the state in the last element of the relation's MSoT state sequence is equal to the relation state in the (relation state, transaction number) pair. Otherwise, it evaluates to a one-element sequence containing the (relation state, transaction number) pair.

We define formally the semantics of commands using the same approach we used to define the semantics of expressions. We define the semantic function **C** for each kind of command allowed in the language. In each of the following definitions, the predicate specifies the conditions under which the command is executed. If these conditions hold, a new database state is produced and the status code OK is returned; otherwise, the database state is left unchanged and the status code ERROR is returned. The conditions specified in each definition are both necessary and sufficient to ensure that only semantically correct expressions are evaluated and that the class, signature, and state of each relation in the database state following execution of the command are consistent. In all five definitions we assume that if a_1, a_2, a_3, b_1, b_2, and b_3 are all sequences, then $(a_1, a_2, a_3)\|_3(b_1, b_2, b_3)$ denotes the triple $(a_1\|b_1, a_2\|b_2, a_3\|b_3)$, where "$\|$" is the concatenation operator on sequences. Also, the notation $d[r/I]$ stands for a new database state that differs from the database state d only in that it maps the identifier I onto the relation r.

2.5.1. Defining a relation. The define_relation command assigns to a relation, whose current class

Table 1. Define_relation command

Current Class		*New Class*	
		SingleStateClass	**MultiStateClass**
Undefined	New Class Extends MSoT Class	Not Applicable	1 Extend MSoT Append to MSoT, if Changed Append to MSoT, if Changed
	New Class Does Not Extend MSoT Class	2 Append to MSoT Append to MSoT, if Changed Append to MSoT, if Changed	Append to MSoT Append to MSoT, if Changed Append to MSoT, if Changed
SingleStateClass		3 Error	Error
MultiStateClass		Error	Error

is UNDEFINED, a new class and signature and an empty relation state consistent with the new class. The assignment becomes effective when the transaction in which the command occurs is committed. The changes that the command makes to the relation to effect this assignment depend on the relation's current class; on the last class, signature, and state, if any, in the relation's MSoT for the transaction in which the command occurs; and on whether the new class is a single-state class (i.e. SNAPSHOT or HISTORICAL) or a multi-state class (i.e. ROLLBACK or TEMPORAL). We hereafter refer to the last class, signature, and state in a relation's MSoT, if present, as the relation's MSoT class, signature, and state, respectively. The actions performed by the define_relation command, for all possible combinations of these variables, can be reduced to the three cases shown in Table 1.

If the relation's current class is UNDEFINED, the define_relation command replaces the relation with its MSoT, augmented to include the new class, signature, and state. If the new class represents a non-disjoint extension of the relation's MSoT class, the interval assigned the MSoT class is extended (i.e. made into a dynamically expanding interval by changing the second transaction-number component to "–") to include the transaction in which the command occurs. This case is limited to define_relation commands in multiple-command transactions, which we discuss in Section 2.5.5. Otherwise, the new class is appended to the MSoT class sequence. In either case, a new signature (state) is added to the MSoT signature (state) sequence only if it differs from the MSoT signature (state). If the relation's current class is other than UNDEFINED, the command encounters an error condition and leaves the relation unchanged.

The formal definition of define_relation follows directly from Table 1. Here, M ranges over the domain $RELATION + \{(\langle\ \rangle, \langle\ \rangle, \langle\ \rangle)\}$.

```
$\mathbf{C}[\![\text{define\_relation}(I, Y, Z)]\!](d, tn) =$
    if    $(M = MSoT(d(I), tn) \wedge LastClass(d(I)) = \text{UNDEFINED}$
          $\wedge\ \mathbf{Y}[\![Y]\!] \neq \text{ERROR} \wedge \mathbf{Z}[\![Z]\!] \neq \text{ERROR})$
    then if $FindClass(M, tn-1) = \mathbf{Y}[\![Y]\!]$
         then $(d[(Expand(M)\|_3(\langle\ \rangle, NewSignature(M, (\mathbf{Z}[\![Z]\!], tn)),$
                                          $NewState(M, (\emptyset, tn), (\mathbf{Y}[\![Y]\!], \mathbf{Z}[\![Z]\!])))$
              $)/I], \text{OK})$
         else $(d[(M\|_3(\langle(\mathbf{Y}[\![Y]\!], tn, -)\rangle, NewSignature(M, (\mathbf{Z}[\![Z]\!], tn)),$
                                          $NewState(M, (\emptyset, tn), (\mathbf{Y}[\![Y]\!], \mathbf{Z}[\![Z]\!])))$
              $)/I], \text{OK})$
    else $(d, \text{ERROR})$
```

Examples. In these and later examples, we show the result of executing a sequence of commands, starting with the database (DS, 8). We assume that each command corresponds to a single-command transaction that commits. For simplicity, we always refer to the current database state as DS, although it changes with each command's execution (i.e. transaction's commitment). We also restrict the commands to the relations denoted by the identifiers R1, R2, and R3 and show only the portion of the database state changed by each command's execution. We assume that DS maps the identifiers R2 and R3 onto the following relations.

class	*signature*	*state*
R2 → ⟨(ROLLBACK, 1, 5),	⟨((ename→string, ssn→integer), 1)	⟨(∅, 1),
		({("Phil", 250861414), ("Linda", 147894290), ("Ralph", 459326889)}, 3)
(UNDEFINED, 6, −)⟩	⟩	⟩

class	*signature*	*state*
R3 → ⟨(UNDEFINED, 0, −)⟩	⟨ ⟩	⟨ ⟩

Note that a relation whose current class is UNDEFINED has neither a current signature nor a current state. The relation denoted by R2 has a MSoT signature (state), but not a current signature (state). The relation denoted by R3 has neither a MSoT signature (state) nor a current signature (state).

C⟦define_relation(R2, rollback, (ename:string, ssn:integer))⟧(DS, 9)

class	*signature*	*state*
R2 → ⟨(ROLLBACK, 1, 5),	⟨((ename→string, ssn→integer), 1)	⟨(∅, 1),
		({("Phil", 250861414), ("Linda", 147894290), ("Ralph", 459326889)}, 3),
(ROLLBACK, 9, −)⟩	⟩	(∅, 9) ⟩

C⟦define_relation(R3, snapshot, (sname:string, class:string))⟧(DS, 10)

class	*signature*	*state*
R3 → ⟨(SNAPSHOT, 10, −) ⟩	⟨((sname→string, class→string), 10)⟩	⟨(∅, 10) ⟩

The first command makes the relation denoted by R2 a rollback relation over the attributes ename and ssn, effective when transaction 9 commits. Although the new class and the relation's MSoT class are equal, the intervals associated with the two are disjoint. Hence, the new class is appended to the relation's MSoT class sequence. The new signature is not appended to the relation's MSoT signature sequence because it is the same as the relation's MSoT signature. The new state, the empty set, differs from the relation's MSoT state. Hence, it is added to the relation's MSoT state sequence. The second command makes the relation denoted by R3 a snapshot relation over the attributes sname and class, effective when transaction 10 commits. Because the relation's MSoT at transaction 10 is (⟨ ⟩, ⟨ ⟩, ⟨ ⟩), the command transforms the relation's class, signature, and state sequences into single-element sequences containing the new class, signature, and state. Note that information about both relations when they were undefined has been discarded, as it is not needed for rollback.□

2.5.2. Modifying a relation. The modify_relation command assigns to a relation, whose current class is other than UNDEFINED, a new class, signature, and relation state. The assignment becomes effective when the transaction in which the command occurs is committed. The modify_relation command differs from the define_relation command in only three respects. First, the modify_relation command only updates a relation if its current class is not UNDEFINED, whereas the define_relation command does just the opposite. Second, the modify_relation command, unlike the define_relation command, allows the new class (signature) to be the relation's current class (signature). Third, the modify_relation command allows the new relation state to be the value of any semantically correct expression consistent with the new class and signature, whereas the define_relation command requires that the new state be the empty state consistent with the new class. Otherwise, the semantics of the two commands is the same. The actions performed by the modify_relation command are summarized in Table 2.

The formal definition of modify_relation follows directly from the above description of the command and Table 2.

$$
\begin{aligned}
&\mathrm{C}[\![\text{modify\_relation}(I, Y', Z', E)]\!](d, tn) = \\
&\quad \text{if} \quad (M = MSoT(d(I), tn) \wedge \mathbf{T}[\![E]\!](d, tn) \neq \text{ERROR} \wedge LastClass(d(I)) \neq \text{UNDEFINED} \\
&\qquad \wedge\, Consistent(\mathbf{Y}'[\![Y']\!](d(I)), \mathbf{Z}'[\![Z']\!](d(I)), \mathbf{T}[\![E]\!](d, tn)))
\end{aligned}
$$

Table 2. Modify_relation command

Current Class		*New Class*	
		SingleStateClass	**MultiStateClass**
SingleStateClass or MultiStateClass	**New Class Extends MSoT Class**	**Not Applicable**	**1 Extend MSoT Append to MSoT, if Changed Append to MSoT, if Changed**
	New Class Does Not Extend MSoT Class	**2 Append to MSoT Append to MSoT, if Changed Append to MSoT, if Changed**	**Append to MSoT Append to MSoT, if Changed Append to MSoT, if Changed**
Undefined		**3 Error**	**Error**

```
then if    $FindClass(M, tn - 1) = \mathbf{Y'}[\![Y']\!](d(I))$
    then $(d[(Expand(M))\|_3(\langle\ \rangle, NewSignature(M, (\mathbf{Z'}[\![Z']\!](d(I)), tn),$
                                  $NewState(M, (\mathbf{E}[\![E]\!](d, tn), tn), \mathbf{T}[\![E]\!](d, tn)))$
        $)/I]$, OK)
    else $(d[M\|_3(\langle(\mathbf{Y'}[\![Y']\!](d(I)), tn,-)\rangle, NewSignature(M, (\mathbf{Z'}[\![Z']\!](d(I)), tn)),$
                                  $NewState(M, (\mathbf{E}[\![E]\!](d, tn), tn), \mathbf{T}[\![E]\!](d, tn)))$
        $)/I]$, OK)
else $(d$, ERROR)
```

If a relation's current class is other than UNDEFINED, the modify_relation command replaces the relation with MSoT, augmented to include the new class, signature, and state. If the relation's current class is UNDEFINED, the command encounters an error and leaves the relation unchanged.

Examples.

C⟦modify_relation(R2, *, *, ρ(R2, 5) − σename = "Ralph"(ρ(R2, 5)))⟧(DS, 11)

class	*signature*	*state*
R2 → ⟨(ROLLBACK, 1, 5),	⟨((ename→string, ssn→integer), 1)	⟨(∅, 1),
		({("Phil", 250861414), ("Linda". 147894290), ("Ralph", 459326889)}, 3),
(ROLLBACK, 9,–)		(∅, 9),
		({("Phil", 250861414),
⟩	⟩	("Linda", 147894290)}, 11)⟩

C⟦modify_relation(R3, *, *, ρ(R1, 5))⟧(DS, 12)

class	*signature*	*state*
R3 → ⟨(SNAPSHOT, 12,–)	⟨((sname→string, class→string), 12)	⟨({("Phil", "junior"), ("Linda", "senior"),
⟩	⟩	("Ralph", "senior")}, 12)⟩

The first command changes the state of the relation denoted by R2 while the second command changes the state of the relation denoted by R3. The commands, however, do not change the class or signature of either relation. For the first command, the new class (i.e. R2's current class) is a non-disjoint extension of R2's MSoT class. Hence, the interval for R2's MSoT class is made into a dynamically expanding interval that includes transaction 11, but no new element is added to R2's MSoT class sequence. The new signature (i.e. R2's current signature) is the same as R2's MSoT signature, hence it is not added to R2's MSoT signature sequence. The new state differs from R2's MSoT state, hence it is appended to R2's MSoT state sequence. Because R3's MSoT at transaction 12 is still (⟨ ⟩, ⟨ ⟩, ⟨ ⟩), the second command transforms R3's class, signature, and state sequences into single-element sequences containing the new class (i.e. R3's current class), signature (i.e. R3's current signature), and state. Note that R2's state at transaction 9 through transaction 10 has been retained and remains accessible via the rollback operator ρ, but R3's state before transaction 12 has been discarded (i.e. physically deleted from the database state).

C⟦modify_relation(R3, *, (sname:string, course:string),
π(sname)(R3) × [snapshot, (course:string), (course:"English")])⟧(DS, 13)

class	*signature*	*state*
R3 → ⟨(SNAPSHOT, 13,–) ⟩	⟨((sname→string, course→string), 13) ⟩	⟨({("Phil", "English"), ("Linda", "English"), ("Ralph", "English")}, 13)⟩

This command changes R3's signature and state but leaves the relation's class unchanged. It illustrates two possible changes to a relation's signature, deletion of one attribute and addition of another attribute. Deletion of an attribute is usually expressed as a projection over the remaining attributes. Addition of an attribute requires that a value for the new attribute be determined for each tuple in the relation. Often, as in this example, a single default value is specified, which is then appended to each tuple. Note again that R3's state before transaction 13 has been discarded.□

The modify_relation command has several noteworthy properties. First, the command supports all update operations on a relation's state. Append is accommodated by an expression E, generally containing a union operator, that evaluates to a snapshot or historical state containing all the tuples in a relation's current state plus one or more tuples not in the relation's current state. Delete is accommodated by an expression E, generally containing a difference operator, that evaluates to a snapshot or historical state containing only a proper subset of the tuples in a relation's current state. Replace is accommodated by an expression E that evaluates to a snapshot or historical state that differs from a relation's current state only in the attribute values of one or more tuples.

Second, the modify_relation command ensures that a relation's class, signature, and state are consistent following update. The command changes a relation's state only if the new state is consistent with the relation's class and signature. Whenever the command changes a relation's signature, it also changes the relation's state to ensure consistency among the relation's class, signature, and state [29]. Likewise, whenever the command changes a relation's class, it also updates the relation's state, if necessary, to ensure consistency among the relation's class, signature, and state.

Finally, the modify_relation command always treats a relation's signature (state) sequence as an *append-only* sequence when the relation's current class is either rollback or temporal, but it does not automatically discard a relation's current signature (state) on update even if the relation's current class is snapshot or historical. If a relation's current class is a single-state class, the command discards the relation's current signature (state) on update only if the signature (state) is not part of the relation's history as a rollback or temporal relation.

2.5.3. Deleting a relation. The command destroy assigns to a relation, whose current class is other than UNDEFINED, the new class UNDEFINED. It also deletes, either logically or physically, the relation's current signature and state.

```
C⟦destroy(I)⟧(d, tn) =
    if    $M = MSoT(d(I), tn) \wedge LastClass(d(I)) \neq$ UNDEFINED
    then  $(d[M \|_3(\langle(\text{UNDEFINED}, tn, -)\rangle, \langle\ \rangle, \langle\ \rangle)/I]$, OK)
    else  $(d$, ERROR)
```

If the identifier I denotes a relation whose current class is other than UNDEFINED, the command simply appends the new class UNDEFINED to the relation's MSoT for the transaction in which the command occurs.

Examples.

C⟦destroy(R2⟧(DS, 14)

class	*signature*	*state*
R2 → ⟨(ROLLBACK, 1, 5),	⟨((ename→string, ssn→integer), 1)	⟨(∅, 1), ({("Phil", 250861414), ("Linda", 147894290), ("Ralph", 459326889)}, 3),

(ROLLBACK, 9, 13),		(∅, 9), ({("Phil", 250861414), ("Linda", 147894290)}, 11)
(UNDEFINED, 14,–)⟩	⟩	⟩

C⟦destroy(R3)⟧(DS, 15)

class	*signature*	*state*
R3 → ⟨(UNDEFINED, 15,–)⟩	⟨ ⟩	⟨ ⟩

Because R2 denotes a relation whose current class is ROLLBACK, the first command uses the function *MSoT* to "close" the interval associated with the relation's current class. It then appends the element (UNDEFINED, 14,–) to R2's class sequence. These actions together have the effect of logically deleting R2's current signature and state when transaction 14 commits. Note, however, that this signature and state information is still accessible via the rollback operator ρ. The second command uses the function *MSoT* to physically delete R3's current class, signature, and state. No record of R3 as a snapshot relation is retained.□

It is important to observe from these and previous examples that signature and state information associated with a relation when its class was either snapshot or historical is transient. It was physically removed when it became outdated. Hence, the language is consistent with conventional relational DBMS's that discard out-of-date signature and state information (relation R3 illustrates this). However, signature and state information associated with a relation when its class was rollback or temporal is retained, ensuring later access to past states via the rollback operator. Definition of the rollback operator assumes access to a complete record of a relation's signature and state during intervals when the relation's class was either rollback or temporal.

2.5.4. Renaming a relation. The command rename_relation binds a relation's current class, signature, and state to a new identifier.

```
C⟦rename_relation($I_1$, $I_2$)⟧($d$, $tn$) =
    if    ($LastClass(d(I_1)) \neq$ UNDEFINED $\wedge$ $LastClass(d(I_2))$ = UNDEFINED
          $\wedge$ Y⟦$Y$⟧ = $LastClass(d(I_1))$ $\wedge$ Z⟦$Z$⟧ = $LastSignature(d(I_1))$
          $\wedge$ C⟦define_relation($I_2$, $Y$, $Z$)⟧($d$, $tn$) = ($d'$, OK)
          $\wedge$ C⟦modify_relation($I_2$, *, *, $I_1$)⟧($d'$, $tn$) = ($d''$, OK)
          $\wedge$ C⟦destroy($I_1$)⟧($d''$, $tn$) = ($d'''$, OK))
    then  ($d'''$, OK)
    else  ($d$, ERROR)
```

The rename_relation first assigns to the relation denoted by I_2 the current class and signature of the relation denoted by I_1. It then assigns to I_2 the current state of I_1. Finally, it assigns the class UNDEFINED to I_1 and deletes, either logically or physically, I_1's current signature and state. Note that the execution environments for rename_relation's three subordinate commands, while containing different database states, contain the same transaction number. Hence, the changes to both I_1 and I_2 become effective when a single transaction commits.

Example. Recall that R1 is the relation shown in Section 2.2.

C⟦rename_relation(R1, R3)⟧(DS, 16)

class	*signature*	*state*
R1 → ⟨(ROLLBACK, 2, 6),	⟨((sname→string, ssn→integer), 2),	⟨(∅, 2),
		({("Phil", 250861414), ("Linda", 147894290), ("Ralph", 459326889)}, 4),
	((sname→string, class→string), 5)	({("Phil", "junior"), ("Linda", "senior"), ("Ralph", "senior")}, 5)
(UNDEFINED, 16,–)⟩	⟩	⟩

class	*signature*	*state*
R3 → ⟨(SNAPSHOT, 16,–)	⟨((ssn→integer, class→string), 16)	⟨({(250861414, "junior"), (147894290, "senior"), (459326889, "senior")}, 16)⟩
⟩	⟩	

This command binds the current class, signature, and state of the relation denoted by R1 to the identifier R3. Hence, R3 becomes a snapshot relation when transaction 16 commits. The command also transforms R1 into an undefined relation, effective when transaction 16 commits. Because R1's current class, signature, and state are not part of the relation's history as either a rollback or temporal relation, they are physically deleted.□

2.5.5. A sequence of commands. If two or more commands appear in sequence, the commands are executed sequentially. If a command executes without error, the next command is executed using the database state resulting from the previous command's execution. If all the commands execute without error, the commands are mapped onto the final database state and the status code OK. If, however, any command's execution causes an error, the remaining commands are not executed and the status code ERROR is returned.

$$\mathbf{C}[\![C_1, C_2]\!](d, tn) = \text{if } \mathbf{C}[\![C_1]\!](d, tn) = (d', \text{OK}) \text{ then } \mathbf{C}[\![C_2]\!](d', tn) \text{ else } (d, \text{ERROR})$$

Two or more commands appearing in sequence are all commands in the same transaction. Their execution environments have different database states but the same transaction number. Hence, if the commands change the same relation only the last changes to the relation's class, signature, and state are recorded in the final database state. Recall that while a relation's new class, signature, and state may depend on its current class, signature, and state, all commands define the resulting relation in terms of the relation's modified start of transaction. Also, if the commands change several relations, all the changes become effective when the transaction commits.

Examples. In the previous examples, we assumed that the commands were all taken from single-command transactions. We now show the result of executing multiple commands from the same transaction. Recall from the example in Section 2.5.3 that R2 at the current transaction is undefined.

C⟦define_relation(R2, rollback, (ename:string, ssn:integer)),
  modify_relation(R2, *, *, ρ(R2, 5)),
  modify_relation(R2, *, *, R2 − σename = "Linda"(R2))⟧(DS, 17)

class	*signature*	*state*
R2 → ⟨(ROLLBACK, 1, 5),	⟨((ename→string, ssn→integer), 1)	⟨(∅, 1),
		({("Phil", 250861414), ("Linda", 147894290), ("Ralph", 459326889)}, 3),
(ROLLBACK, 9, 13),		(∅, 9),
		({("Phil", 250861414), ("Linda", 147894290)}, 11),
(ROLLBACK, 17,–) ⟩	⟩	({("Phil", 250861414), ("Ralph", 459326889)}, 17)⟩

C⟦destroy(R2), destroy(R3)⟧(DS, 18)

class	*signature*	*state*
R2 → ⟨(ROLLBACK, 1, 5),	⟨((ename→string, ssn→integer), 1)	⟨(∅, 1),
		({("Phil", 250861414), ("Linda", 147894290), ("Ralph", 459326889)}, 3),
(ROLLBACK, 9, 13),		(∅, 9),
		({("Phil", 250861414), ("Linda", 147894290)}, 11),
(ROLLBACK, 17, 17)		({("Phil", 250861414), ("Ralph", 459326889)}, 17)
(UNDEFINED, 18,–)⟩	⟩	⟩

class	*signature*	*state*
R3 → ⟨(UNDEFINED, 18,–)⟩	⟨ ⟩	⟨ ⟩

In the first example, all three commands change R2. Yet, only the last changes to the relation's class, signature, and state are recorded in the database state. Although the first command defined R2 as a rollback relation and the other commands changed

R2's state, only the final change in state is recorded. Hence, all the commands in a single transaction that change the same relation are treated as an atomic update operation. Note that temporary relations can be defined, modified, and then deleted within a transaction without their creation being recorded. In the second example, both R2 and R3 are deleted when transaction 18 commits.□

$$\mathbf{P}[\![\text{begin\_transaction } C \text{ abort\_transaction}]\!](d, tn) = ((d, tn), \text{OK})$$

Aborted transactions are transactions which the user initiates that for some reason, dictated either by the user or by the system, abort rather than commit. They do not change the database.

$$\mathbf{P}[\![P_1; P_2]\!](d, tn) = \text{if } \mathbf{P}[\![P_1]\!](d, tn) = ((d', tn'), \text{OK}) \text{ then } \mathbf{P}[\![P_2]\!](d', tn') \text{ else } \mathbf{P}[\![P_2]\!](d, tn)$$

If a program contains multiple transactions, they are processed in sequence. If the first transaction commits and produces a new database, the second transaction is processed using the new database. Otherwise, the second transaction is processed using the original database.

Finally, we require that each arbitrary sequence of transactions representing a program map onto the database resulting from the execution of the transactions, in order, starting with the empty database. The empty database, (**EMPTY**, 0), is defined using the semantic function **EMPTY**: *IDENTIFIER* → (⟨(UNDEFINED, 0, –)⟩, ⟨ ⟩, ⟨ ⟩). Hence, the database-state component of the empty database is defined to be the function that maps all identifiers onto undefined relations; the transaction-number component of the empty database is defined to be 0. This requirement is both necessary and sufficient to ensure that the transaction-number components of elements in the class, signature, and state sequences of each relation in the database are strictly increasing. A database will always be the cumulative result of all the transactions that have been performed on it since it was created.

2.6. Programs

The semantic function **P** defines the denotation of programs in our language, where a program is a sequence of one or more transactions. Transactions, in turn, may be either single-command or multiple-command transactions. **P** defines a program as a function that maps a database onto a database and a status code. A program is the only language construct that changes a database. Execution of a transaction that commits produces a new database and the status code OK, while execution of a transaction that aborts produces the original database unchanged and the status code ERROR.

$$\mathbf{P}: \mathit{PROGRAM} \rightarrow [\mathit{DATABASE} \rightarrow [\mathit{DATABASE} \times \{\text{OK}, \text{ERROR}\}]]$$

Note that the environments for command and program execution, although similar, are not identical. The environment for command execution is a database state and the transaction number of the active transaction. In contrast, the environment for programs execution is a database, which is an ordered pair consisting of a database state and the transaction number of the most recently committed transaction on that database state.

We now define formally the semantic function **P** for each kind of program.

$$\mathbf{P}[\![\text{begin\_transaction } C \text{ commit\_transaction}]\!](d, tn) = \text{if } \mathbf{C}[\![C]\!](d, tn+1) = (d', \text{OK}) \text{ then } ((d', tn+1), \text{OK}) \text{ else } ((d, tn), \text{ERROR})$$

Committed transactions represent transactions that commit if their commands all execute without error. If all the commands in a transaction execute without error, the transaction is committed. The database's database-state component is updated to record the changes that the commands make to relations, the database's transaction-number component is incremented to record the transaction number of this most recently committed transaction, and the status code OK is produced. If any command's execution produced an error, the transaction is aborted. The database is left unchanged and the status code ERROR is produced. The database is valid independent of the status code.

We now define the semantic function **P**′ that maps a program onto the database resulting from the execution of the program's transactions, starting with the empty database.

$$\mathbf{P}': \mathit{PROGRAM} \rightarrow \mathit{DATABASE}$$

$$\mathbf{P}'[\![P]\!] = \mathit{First}(\mathbf{P}[\![P]\!](\mathbf{EMPTY}, 0))$$

where *First* is the function that maps an ordered pair onto the first component of the ordered pair.

2.7. *Language properties*

We now state, as theorems, four properties of our algebraic language for database query and update. Informal proofs of these theorems are given in Appendix A. The first property was stated initially as an objective in Section 1.

Theorem 1. *The language is a natural extension of the relational algebra for database query and update.*

By natural extension, we mean that our semantics subsumes the expressive power of the relational algebra for database query and update. Expressions in the language are a strict superset of those in the relational algebra. Also, if we restrict the class of all relations to UNDEFINED and SNAPSHOT, then a natural extension implies that (a) the signature and state sequences of a defined relation will have exactly one element each: the relation's current signature and state; (b) a new state always will be a function of the current signature and state of defined relations via the relational algebra semantics; and (c) deletion will correspond to physical deletion.

The second property argues that the semantics is minimal, in a specific sense. Other definitions of minimality, such as minimal redundancy or minimal space requirements, are more appropriate for the physical level, where actual data structures are implemented, than for the algebraic level.

Theorem 2. *The semantics of the language minimizes the number of elements in a relation's class, signature, and state sequence needed to record the relation's current class, signature, and state and its history as a rollback or temporal relation.*

The third property ensures that the language accommodates implementations that use WORM optical disk to store non-current class, signature, and state information, another objective of our extensions.

Theorem 3. *Transactions change only a relation's class, signature, and state current at the start of the transaction.*

3. ADDITIONAL ASPECTS OF THE ROLLBACK OPERATORS

The rollback operators in our language are more powerful than suggested in the previous section, in several ways. First, the rollback operators, as defined, are restricted to the retrieval of a single snapshot or historical state from a named relation current at the time of a specified transaction. In reality, however, the rollback operators derive a single snapshot or historical state from one or more of the named relation's stored states rather than simply retrieving a single state. The rollback operators actually roll back a relation to the subsequence of the relation's state sequence corresponding to an interval of time of arbitrary length, if the relation's class and signature remained constant over that interval of time. The rollback operators return the single state composed of tuples from all the states in the specified subsequence of relation states (effectively, a relational union, either snapshot or historical, is performed). The rollback operators thus take two transaction times as arguments:

$$E ::= \rho(I, N, N) \mid \hat{\rho}(I, N, N)$$

Second, the rollback operators do not simply retrieve a snapshot or historical state from a named relation but rather an augmented version of that state. To the state's explicit attributes, defined in its signature, the rollback operators add new explicit attributes corresponding to the state's implicit time attributes (i.e. transaction times for snapshot states, transaction and valid times for historical states). The rollback operators' addition of these new attributes to the state's existing explicit attributes allows the user to display the values of the state's implicit time attributes without allowing direct access to the attributes themselves. These explicit values are considered to be in the domain of user-defined time. This behavior requires that the semantic function **T** compute a relational signature containing these additional attributes.

Third, the rollback operator ρ can be applied to temporal relations as well as rollback relations. If ρ rolls back a relation to a time when the relation's class was TEMPORAL, ρ will convert the relation's historical state current at the time into a corresponding snapshot state and return this new snapshot state. Likewise, the rollback operator $\hat{\rho}$ can be applied to rollback relations as well as temporal relations. If $\hat{\rho}$ rolls back a relation to a time when the relation's class was ROLLBACK, $\hat{\rho}$ will convert the relation's snapshot state current at that time into a corresponding historical state and return this new historical state.

While these extensions are conceptually straightforward, the notation required to define them formally is cumbersome and will not be presented.

4. SUMMARY AND RELATED WORK

In summary, this paper has defined an algebraic language for database query and update that subsumes the relational algebra, can accommodate an arbitrary historical algebra, and supports both snapshot and historical rollback. The language also has a simple semantics and supports scheme evolution. Only two additional operators, ρ and $\hat{\rho}$, were necessary. The additions required for transaction time did not compromise any of the useful properties of the (conventional) snapshot algebra. Type-checking was also introduced, freeing the encapsulated algebra from dealing with expressions not consistent with the (possibly time-varying) scheme.

The primary contribution is an algebraic means of supporting schema evolution in the context of general support for transaction time. As an algebraic language for database query and update, our language can serve as the underlying evaluation mechanism for queries and updates in a temporal data manipulation language that supports evolution of a database's contents and schema. It can also be used as the basis for proving various physical implementations of temporal database management systems correct. Our language also is compatible with efforts to add transaction time to the relational data model at both the user-interface and physical levels. At least three temporal query languages have been proposed that support rollback operations [12, 30, 31] and several studies have investigated efficient storage and access strategies for temproal database [32–39].

There have been two other attempts to incorporate both valid time and transaction time in an algebra. In Ben-Zvi's proposal [12], valid time and transaction time were supported through the addition of implicit time attributes to each tuple in a relation. The algebra was extended with the *Time-View* algebraic operator which takes a relation and two times as arguments and produces the subset of tuples in the relation valid at the first time (the valid time) as of the second time (the transaction time). The Time-View operator thus rolls back a relation to a transaction time but returns only a subset of the tuples in the relation at that transaction time (i.e. those tuples at some specified time). This restricted definition of the Time-View operator is tied inextricably to a particular handling of valid time. Our approach is compatible with any historical algebra. Gadia represents valid time and transaction time as two symmetrical dimensions in a Boolean algebra of multidimensional time stamps [40]. He allows rollback operations on transaction time through a generalized restriction operator, which may be applied to any of a relation's time dimensions. He does not, however, address the problems of database update or schema evolution.

While a few authors have envisaged the benefits of a time-varying schema [12, 30, 41, 42], only one other extension of the relational algebra, that proposed by Ben-Zvi, includes support for schema versioning. Ben-Zvi proposes that a temporal relation's schema itself be represented as a temporal relation, thus providing a uniform treatment for evolution of a relation and its schema [12]. He does not, however, provide formal semantics for schema evolution in the context of general support for transaction time. Martin proposes a non-algebraic solution to the problem of an evolving schema in temporal databases using modal temporal logic [43]. A schema temporal logic is proposed to deal with changes in schema. A set of schema temporal logic formulae are associated with a schema to describe its evolution and temporal queries are interpreted in the context of these formulae. This approach, unlike ours, forces synchronization between valid time and schema changes. Again, formal semantics are not provided. Finally, Adiba, in describing mechanisms for the storage and manipulation of historical multi-media data, advocates, like Ben-Zvi, that the history notion used to model changes in a database's contents also be used to model changes in the database's schema [44].

While there has been significant interest in database reorganization and evolution (also termed restructuring) [3, 29, 45–51], such approaches have assumed that the schema (and hence the contents) of the entire database will be modified during restructuring, ensuring that only one schema is in force. These approaches address schema evolution but not schema versioning. Since we formalize the schema as a sequence ordered by transaction time, several schemas can be in force, selectable through the rollback operator. A second difference is that we focus solely on algebraic support for schema evolution, while the other papers considered the related issues of determining what changes to the schema are necessary and what those changes imply regarding the new state to be calculated. Certainly, all these issues must be addressed in developing a comprehensive solution to schema evolution.

In contrast to these previous approaches, the WAND system did permit several generations of schemas to be simultaneously present [52]. This system differs from our approach in two respects. First, the WAND system was based on the network model, whereas our approach is based on the relational model. More significantly, schema evolution was supported in the WAND system to allow dynamic restructuring of the database. While data in the WAND system could also be associated with one of several generations of schemas, the data were always restructured to match the most recent schema as they were referenced. Multiple generations were introduced to achieve concurrency between restructuring and execution of application programs. Hence, the underlying model did not support transaction time or rollback. The WAND system was effectively a snapshot DBMS that permitted applications to access and change the database while a global restructuring was being performed.

ORION, a prototype object-oriented database system developed at MCC, takes a similar approach [45]. An important difference is that when the schema in ORION is modified, no disk-resident data instances need be updated. Instead, when an instance is referenced by an application program and fetched into memory, it is transformed into an instance conforming to the schema currently in effect. Again, only one schema is ever in effect; the implementation places the burden of updating the data across a schema change on subsequent retrievals.

Schema versioning has also been investigated in the context of object-oriented database models [46, 53]. However, their versioning differs significantly from that proposed here. First, their versions are identified by user-supplied names, rather than by transaction numbers, as in our model. Each object update operation names a schema version to which it is to be applied, and an update to one version of an object could affect the data stored in other versions. In our model, only the most recent version can be updated, and updates do not affect other versions. In the ENCORE object-oriented DBMS [54, 55], versions are identified by transaction numbers, as in our model, but updates can be made to any version, as in the other object-oriented models. However, in ENCORE, an update applied to one version does not affect the other versions. At the risk of over-simplifying, object-oriented schema versions resemble traditional relational views [56] and do not involve transaction time as defined in this paper.

Several researchers have used denotational semantics to define formally the semantics of databases, DBMS's and query languages. Subieta proposes an approach for defining query languages formally using denotational semantics [57]. This approach allows powerful query languages with precise semantics to be defined for most database models. Rishe proposes that denotational semantics be used to provide a uniform treatment of database semantics at different information levels based on hierarchies of domains of mappings from "less semantic" representations of information into "more semantic" representations [58]. Neither Subieta nor Rishe, however, include in their approaches any facilities for dealing with transaction time or an evolving schema. Lee proposes a denotational semantics for administrative databases, where databases are regarded as a collection of logical assertions [59]. Here, the denotation of an expression in a first-order predicate calculus is based, in part, on its evaluation in a time dimension, analogous to valid time, in a possible world, analogous to a cross-section of a database state at a transaction.

An obvious next step would be to implement an evolving schema that fully supports the rollback operator. One approach we are considering converts the system relations in our prototype [32] to be rollback relations, rather than snapshot relations as they are now. Changes to the semantic analysis portions of the query analysis would be required, but it appears that changes to the backend of the DBMS would be minimal.

Another step would be to investigate extensions to the language. A straightforward extension of the language would introduce algebraic operators that map between the domain of snapshot states and the domain of historical states directly. The introduction of such operators into the snapshot and historical algebras would render the algebras multisorted. Because the two algebras, without these operators, are unisorted and because we wish to retain this property for the algebras, we have elected not to introduce such conversion operators into our language.

A second extension would introduce an algebra of signatures, analogous to the algebras of snapshot and historical states, to remove the restriction that signature specifications in the commands define_relation and modify_relation be a relation's current signature or a constant. This extension would support signature changes dependent on both the current and past signatures of relations in the database.

A third extension would remove the requirements of a relation's schema being constant over the transaction interval specified in the rollback operation. The major problem is in calculating the schema for the resulting relation. A general but simple approach has not yet been found.

Finally, the effect of schema evolution on applications programs accessing the database should be considered [52]. Maintaining consistency between such programs and the database schema becomes more difficult. Similarly, query pre-compilation, such as performed in System R [60], may or may not be effective, depending on whether the time-stamps provided to the rollback operators are constants or are values supplied by the application program. However, it appears that techniques similar to those employed by the WAND system, those appearing in ORION, and those proposed [46] could serve to amortize the cost of schema changes.

Acknowledgements—We wish to thank Bharat Jayaraman and Peter Mills for suggesting many corrections and improvements to this paper and the referees for suggesting significant improvements to the formalism used here. This research was supported by NSF Grants DCR-8402339 and IRI-8902707, and ONR grant N00014-86-K-0680. Research by the first author also was supported by the United States Air Force. Research by the second author was supported in part by an IBM Faculty Development Award.

REFERENCES

[1] C. J. Date. *An Introduction to Database Systems.* Addison-Wesley, Reading, Mass. (1976).

[2] J. D. Ullman. *Principles of Database Systems*, 2nd edn. Computer Science Press, Potomac, Md. (1982).

[3] G. H. Sockut and R. P. Goldberg. Database reorganization—principles and practice. *ACM Comput. Surveys* **11**(4), 371–395 (1979).

[4] E. McKenzie. Bibliography: temporal databases. *ACM SIGMOD Rec.* **15**(4), 40–52 (1986).

[5] E. McKenzie and R. Snodgrass. Extending the relational algebra to support transaction time. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, San Francisco (1987).

[6] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM* **13**(6), 377–387 (1970).

[7] R. Snodgrass and I. Ahn. A taxonomy of time in databases. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, Austin, Tex. (1985).

[8] R. Snodgrass and I. Ahn. Temporal databases. *IEEE Comput.* **19**(9), 35–42 (1986).

[9] C. J. Bontempo. Feature analysis of query-by-example. In *Relational Database Systems*, pp. 409–433. Springer, New York (1983).

[10] R. Overmyer and M. Stonebraker. Implementation of a time expert in a database system. *ACM SIGMOD Rec.* **12**(3), 51–59 (1982).

[11] *ENFORM Reference Manual.* Tandem Computers, Inc, Cupertino, Cal. (1983).

[12] J. Ben-Zvi. The time relational model. PhD. Diss., Computer Science Department, UCLA (1982).

[13] J. Clifford and A. Croker. The historical relational data model (HRDM) and algebra based on lifespans. In *Proc. Int. Conf. Engineering*, Los Angeles (1987).
[14] S. K. Gadia. A homogeneous relational model and query languages for temporal databases. *ACM Trans. Database Systems* **13**(4), 418–448 (1988).
[15] S. K. Gadia. Toward a multihomogeneous model for a temporal database. In *Proc. Int. Conf. Data Engng*, Los Angeles (1986).
[16] S. Jones, P. Mason and R. Stamper. LEGOL 2.0: a relational specification language for complex rules. *Information Systems* **4**(4), 293–305 (1979).
[17] N. A. Lorentzos and R. G. Johnson. TRA: a model for a temporal relational algebra. In *Proc. Conf. Temporal Aspects in Information Systems, AFCET*, France (1987).
[18] E. McKenzie. An algebraic language for query and update of temporal databases. PhD. Diss., Computer Science Department, University of North Carolina at Chapel Hill (1988).
[19] S. B. Navathe and R. Ahmed. TSQL—A language interface for history databases. In *Proc. Conf. Temporal Aspects in Information Systems, AFCET*, France (1987).
[20] A. U. Tansel. Adding time dimension to relational model and extending relational algebra. *Information Systems* **11**(4), 343–355 (1986).
[21] C. S. Yeung. Query languages for a heterogeneous temporal database. Master's Thesis, EE/CS Department, Texas Tech University (1986).
[22] M. J. C. Gordon. *The Denotational Description of Programming Languages*. Springer, New York (1979).
[23] J. E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, Mass. (1977).
[24] D. S. Scott. Data types as lattices. *SIAM J. Comput.* **5**(3), 522–587 (1976).
[25] C. Strachey. Towards a formal semantics. In *Formal Language Description Languages for Computer Programming*, pp. 198–220. North Holland, Amsterdam (1966).
[26] E. McKenzie and R. Snodgrass. An evaluation of algebras incorporating time. Technical Report TR89-22, Computer Science Department, University of Arizona (1989).
[27] D. A. Schmidt. *Denotational Semantics, A Methodology for Language Development*. Allyn & Bacon, Newton, Mass. (1986).
[28] D. Maier. *The Theory of Relational Databases*. Computer Science Press, Rockville, Md. (1983).
[29] S. B. Navathe and J. P. Fry. Restructuring for large databases: three levels of abstraction. *ACM Trans. Database Systems* **1**(2), 138–158 (1976).
[30] G. Ariav. A temporally oriented data model. *ACM Trans. Database Systems* **11**(4), 499–527 (1986).
[31] R. Snodgrass. The temporal query language TQuel. *ACM Trans. Database Systems* **12**(2), 247–298 (1987).
[32] I. Ahn. Towards an implementation of database management systems with temporal support. In *Proc. Int. Conf. Data Engineering*, Los Angeles (1986).
[33] I. Ahn. Performance modeling and access methods for temporal database management systems. PhD. Diss., Computer Science Department, University of North Carolina at Chapel Hill (1986).
[34] I. Ahn and R. Snodgrass. Performance evaluation of a temporal database management system. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, Washington, D.C. (1986).
[35] I. Ahn and R. Snodgrass. Performance analysis of temporal queries. *Information Sci.* **49**, 103–146 (1989).
[36] V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner and J. Woodfill. Designing DBMS support for the temporal dimension. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, Boston, Mass. (1984).
[37] D. Rotem and A. Segev. Physical organization of temporal databases. In *Proc. Int. Conf. Data Engineering*, Los Angeles (1987).
[38] A. Shoshani and K. Kawagoe. Temporal data management. In *Proc. Conf. Very Large Databases*, Kyoto, Japan (1986).
[39] S. Thirumalai and S. Krishna. Data organization for temporal databases. Technical Report, Raman Research Institute, India (1988).
[40] S. K. Gadia and C. S. Yeung. A generalized model for a relational temporal database. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, Chicago (1988).
[41] J. Shiftan. An assessment of the temporal differentiation of attributes in the implementation of a temporally oriented DBMS. PhD. Diss., Information Systems Area, Graduate School of Business Administration, New York University (1986).
[42] D. Woelk, W. Kim and W. Luther. An object-oriented approach to multimedia databases. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, Washington, D.C. (1986).
[43] N. G. Martin, S. B. Navathe and R. Ahmed. Dealing with temporal schema anomalies in history databases. In *Proc. Conf. Very Large Databases*, Brighton, England (1987).
[44] M. E. Adiba and N. Bui Quang. Historical multimedia databases. In *Proc. Conf. Very Large Databases*, Kyoto, Japan (1986).
[45] J. Banerjee, W. Kim, H.-J. Kim and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, San Francisco, (1987).
[46] H.-J. Kim and H. F. Korth. Schema versions and DAG rearrangement views in object-oriented databases. Technical Report, Department of Computer Science, Georgia Institute of Technology (1989).
[47] V. M. Markowitz and J. A. Makowsky. Incremental reorganization of relational databases. In *Proc. Conf. Very Large Databases*, Brighton, England (1987).
[48] S. B. Navathe. Schema analysis for database restructuring. *ACM Trans. Database Systems* **5**(2), 157–184 (1980).
[49] N. Roussopoulos and L. Mark. Schema manipulation in self-describing and self-documenting data models. *Int. J. Comput. Information Sci.* **14**(1), 1–28 (1985).
[50] N. C. Shu, B. C. Housel, R. W. Taylor, S. P. Ghosh and V. Y. Lum. EXPRESS: a data extraction, processing, and restructuring system. *ACM Trans. Database Systems* **2**(2), 134–174 (1977).
[51] N. C. Shu. Automatic data transformation and restructuring. In *Proc. Int. Conf. Data Engineering*, Los Angeles (1987).
[52] R. Gerritsen and H. L. Morgan. Dynamic restructuring of databases with generation data structures. In *Proc. ACM Annual Conf.*, Houston (1976).
[53] W. Kim and H. T. Chou. Versions of schema in OODB. In *Proc. Conf. Very Large Databases*, Long Beach, Cal. (1988).
[54] A. H. Skarra and S. B. Zdonik. The management of changing types in an object-oriented database. In *Proc. Conf. Object-Oriented Programming Systems, Languages and Applications*, Portland, Or. (1986).
[55] A. H. Skarra and S. B. Zdonik. Type evolution in an object-oriented database. In *Research Directions in Object-Oriented Programming of Computer Systems*, pp. 393–415. MIT Press, Cambridge, Mass. (1987).
[56] D. D. Chamberlin, J. N. Gray and I. L. Traiger. Views, authorization, and locking in a relational data base system. In *AFIPS Conf. Proc.*, Anaheim, Cal. (1975).

[57] K. Subieta. Denotational semantics of query languages. *Information Systems* **12**(1), 69–82 (1987).

[58] N. Rishe. On denotational semantics of data bases. In *Proc. Int. Conf. Mathematical Foundations of Programming Semantics*, Manhattan (1985).

[59] R. M. Lee. A denotational semantics for administrative databases. In *Proc. IFIP WG 2.6 Working Conf. Data Semantics*, Hasselt, Belgium (1985).

[60] D. D. Chamberlin, M. M. Astrahan, W. F. King, R. A. Lorie, J. W. Mehl, T. G. Price, M. Schkolnick, P. Selinger Griffiths, D. R. Slutz, B. W. Wade and R. A. Yost. Support for repetitive transactions and ad hoc queries in System R. *ACM Trans. Database Systems* **6**(1), 70–94 (1981).

APPENDIX A

Proofs of Language Properties

In this appendix, we provide informal proofs for three properties of our language, stated as theorems in Section 2.7.

Theorem 1

First, we show that expressions in our language are a strict superset of those in the relational algebra. Suppose we only allow expressions involving constants that denote snapshot states, identifiers that denote relations whose current class is SNAPSHOT, and the five relational operators. Then, expressions in the language are exactly those allowed in the relational algebra. But expressions in our language also may involve constants that denote historical states, identifiers that denote relations whose current class is other than SNAPSHOT, and both historical and rollback operators. Hence, expressions in our language are a strict superset of those in the relational algebra.

Next, we show that our semantics reduces to the conventional semantics of database state and database update via the relational algebra. Suppose we restrict the class of all relations to UNDEFINED and SNAPSHOT. Then,

(a) The signature and state sequences of a defined relation will have exactly one element each, the relation's current signature and state. The relation can have no history as a rollback or temporal relation; hence its MSoT always will be $(\langle\ \rangle, \langle\ \rangle, \langle\ \rangle)$. Because the define_relation and modify_relation commands change a relation's signature sequence by appending no more than one element to the relation's MSoT signature sequence, these commands always will produce a relation with a single-element signature sequence. The same holds for the relation's state sequence.

(b) A new state always will be a function of the current signature and state of defined relations via the relational algebra semantics. Both the define_relation and modify_relation commands determine a new state via expression evaluation. The only semantically correct expressions are those involving constants that denote snapshot states, identifiers that denote relations whose current class is SNAPSHOT, and the five relational operators. These expressions are exactly those allowed in the relation algebra, their value depending on the current state and signature of defined relations only.

(c) Deletion will correspond to physical deletion. The destroy command changes a relation by appending an element to the relation's MSoT class sequence; it never adds information to the relation's signature or state sequences. The destroy command always will produce a relation whose signature and state sequences are empty, which corresponds to physical deletion of a relation's current signature and state.■

Theorem 2

Assume that the number of elements in a relation's class sequence exceeds the minimum needed to record the relation's current class and its history as a rollback or temporal relation. Then, (a) there are two consecutive elements in the sequence that can be combined or (b) there is an element in the sequence that can be removed. Consider case (a). Consecutive elements in the class sequence can be combined only if they record the same class over non-disjoint intervals. But the commands only append a new element to a relation's class sequence if it either differs from the relation's MSoT class or its interval is disjoint from that of the relation's MSoT class. Hence, no two consecutive elements in a relation's class sequence can have the same class but non-disjoint intervals. Now consider case (b). Commands always produce a new relation by appending new class information to a relation's MSoT class sequence. But, it can be shown that all elements in a relation's MSoT class sequence record intervals when the relation was either a rollback or temporal relation. Hence, no element can be removed. If no two elements can be combined and no element can be removed, our assumption is contradicted and the number of elements in the class sequence must be minimal. Similar arguments hold for the relation's signature and state sequences.■

Theorem 3

This property is a consequence of the way the *MSoT* function is defined and used. We first prove the property for a relation's signature sequence and then for its class and state sequences.

A relation's current signature at the start of a transaction is the last element in the relation's signature sequence. Assume, therefore, that a transaction changes an element that is in the relation's signature sequence at the start of the transaction but is not the last element in the sequence. Such a change must occur during the execution of a command. When the first command in a transaction executes, *MSoT* discards the last element in the relation's signature sequence, if the relation's current class is either SNAPSHOT or HISTORICAL. Otherwise, it retains all the elements. When each subsequent command in the transaction is executed, *MSoT* only discards any element that the preceding command added to the sequence. Hence, *MSoT* never changes an element in a relation's signature sequence that precedes the last element in the sequence at the start of the transaction. Commands, although they may append an element to the relation's MSoT signature sequence, never change existing elements. Hence, commands never change an element in a relation's signature sequence that precedes the last element in the sequence at the start of the transaction and our assumption is contradicted. The same argument holds for the relation's state sequence.

The above argument holds for a relation's class sequence with the following provisos. When the first command in a transaction executes, *MSoT* discards the last element in the relation's class sequence if the relation's current class is UNDEFINED. Also, if the relation's current class is either ROLLBACK or TEMPORAL, *MSoT* changes the last element in the sequence to "close" the interval assigned to the relation's current class at the start of the transaction. When each subsequent command in the transaction is executed, *MSoT* "re-closes" this same interval, if extended by the preceding command. Hence, *MSoT* never changes an element in a relation's class sequence that precedes the last element in the sequence at the start of the transaction. Commands may change the last element in a relation's MSoT class sequence to "extend" the interval assigned to the class component of that element, but only if the new class and the relation's MSoT class are equal and their intervals abut. This occurs only when the last element in the relation's MSoT class sequence corresponds to the last element in the relation's class sequence at the start of the transaction (i.e. the class of the relation at the start of the transaction was either ROLLBACK or TEMPORAL). Otherwise, the intervals could not abut as there would exist an intervening interval when the relation's class was either SNAPSHOT, HISTORICAL, or UNDEFINED. Hence, commands never change an element in a relation's class sequence that precedes the last element in the sequence at the start of the transaction.■

APPENDIX B

MSoT and its Subordinate Functions

In this appendix, we present a formal definition for *MSoT* (modified start of transaction) and descriptions of its subordinate functions. For these definitions, let *tn* and *tn'* range over the domain [*TRANSACTION NUMBER* + {–}], *u* and *u'* range over the domain [*RELATION CLASS* × *TRANSACTION NUMBER*]*, *v* range over the domain [*RELATION SIGNATURE* × *TRANSACTION NUMBER*]*, *w* range over the domain

[[*SNAPSHOT STATE* × *TRANSACTION NUMBER*] + [*HISTORICAL STATE* × *TRANSACTION NUMBER*]]*.

MSoT maps a relation (u, v, w) and a transaction number *tn* onto the history of the relation as a rollback or temporal relation before the start of transaction *tn*:

$$MSoT\text{:}[[RELATION + \{(\langle\,\rangle, \langle\,\rangle, \langle\,\rangle)\}] \times TRANSACTION\ NUMBER] \rightarrow [RELATION + \{(\langle\,\rangle, \langle\,\rangle, \langle\,\rangle)\}]$$

```
$MSoT((u, v, w), tn) =$
    if $(u' = PrefixClasses(u, tn) \wedge u' \neq \langle\,\rangle \wedge tn' = LastTrNumber(u'))$
    then if $MultiStateClass(LastClass((u', v, w)))$
        then $(Close(u', tn - 1), PrefixSigs(v, tn), PrefixStates(w, tn))$
        else $(PrefixClasses(u, tn'), PrefixSigs(v, tn'), PrefixStates(w, tn'))$
    else $(\langle\,\rangle, \langle\,\rangle, \langle\,\rangle)$
```

Close maps a relation's class sequence *u* and a transaction number *tn* onto the subsequence recorded through transaction *tn*. It also sets the second transaction-number component in the last element of the resulting sequence to *tn* if the component is either "–" or greater than *tn*.

LastTrNumber maps a relation's class sequence onto the transaction number of the transaction that appended the last element to the sequence. If the relation's class sequence is empty, *LastTrNumber* returns ERROR.

MultiStateClass is a Boolean function that determines whether a class is either ROLLBACK or TEMPORAL.

PrefixClasses maps a relation's class sequence *u* and a transaction number *tn* onto the subsequence recorded before the start of transaction *tn*.

PrefixSigs maps a relation's signature sequence *v* and a transaction number *tn* onto the subsequence recorded before the start of transaction *tn*.

PrefixStates maps a relation's state sequence *w* and a transaction number *tn* onto the subsequence recorded before the start of transaction *tn*.