REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

AD-A232 656

(2)

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Jan. 15, 1991	FINAL: Jan. 15, 1989 - Jan. 14, 1991

4. TITLE AND SUBTITLE

Mathematical Foundations of Databases

5. FUNDING NUMBERS

AFOSR Grant 89 - 0244

61102F 2304/A2

6. AUTHOR(S)

Seymour Ginsburg, Principal Investigator
Stephen Kurtzman, Research Assistant
Xiaoyang Wang, Research Assistant

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Computer Science Department
University of Southern California
Los Angeles, CA 90089

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Office of Scientific Research /NM
Bolling Air Force Base, DC 20332-6448

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFOSR-89-0244

11. SUPPLEMENTARY NOTES

ELECTE
MAR 1 1 1991
S D

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Two distinct topics were studied, each by one of the research assistants and each destined to be part of his doctoral dissertation. The first topic, "Properties of Spreadsheet Histories", formalized the use of spreadsheets for modelling the history of accounting-like information. The investigated subtopics included database operations (such as selection, projection, each of the database operations were also presented. The second topic, "Declarative Sequence Operations and Their Usage in Query Languages", introduced a family of sequence operations based on the regular expressions from formal language theory. The items examined included their mathematical properties (such as their expressive power) and their usage in various query languages (e.g., SQL) of database systems. A number of example queries were also exhibited.

14. SUBJECT TERMS

Database theory, spreadsheets, historical information, sequences, sequence operations, query languages, regular expressions

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

AFOSR FINAL REPORT, JANUARY 15, 1991

Title: Mathematical Foundations of Databases

Principle Investigator: Seymour Ginsburg, Fletcher Jones Professor of Computer Science

AFOSR Grant Number: 89-0244

Other Investigators: Stephen Kurtzman, Student
Xiaoyang Wang, Student

Purpose: To examine theoretical aspects of relational databases(and its extensions) suggested by computer science considerations.

Accomplishments: Two distinct topics were investigated during the grant period. Each was studied by one of the above research assistants and is to become an integral part of his PhD thesis. A brief summary of the results obtained are now given. Enclosed with this review are two reports, describing in more detail the results obtained.

The first report, "Properties of Spreadsheet Histories", is by Stephen Kurtzman. In this report, some theoretical aspects of one of the most widely used types of small-business data-processing software, namely the spreadsheet program, are studied. In particular, a formal model for spreadsheet histories is presented and examined with respect to two questions. First, the database operations of selection, projection, cohesion and union are considered. The primary question here is whether or not these operations preserve the formal model. The answer is yes for selection, cohesion and intersection, and no for the remaining two. A necessary and sufficient condition is then given for projection and union to preserve the basic model. The second question concerns a notion of spreadsheet-history equivalence based on the projection operation. This concept, projection simulation, is defined and three sufficiency conditions presented for when it preserves an important subclass, the "history bounded", of the data model.

The second report, "Declarative Sequence Operations and Their Usage in Query Languages", is by Xiaoyang Wang. In this report, a family of declarative sequence operations (based on regular expressions from formal language theory) is introduced and studied. The operations were chosen for their power and simplicity in describing the access of sequential information. There are two parts to the investigation. The first part is devoted to the study of these sequence operations. In particular, they are characterized by a type of automaton. The decomposition of operations is then considered and an infinite non-collapsing hierarchy of sequence operations established. The second part of the report introduces both a database model with sequences as a basic data construct, and a query language defined in terms of the sequence operations. Numerous examples are given to demonstrate the practical utility of the sequence operations.

Declarative Sequence Operations and Their Usage in Query Languages[†]

Xiaoyang Wang
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0782

Abstract

A family of declarative sequence operations based on regular expressions is introduced and their properties studied. Using these operations, an extension of SQL is presented over a simple database model with sequence constructors.

1 Introduction

In a typical complex-object database system, tuple, set and sequence (or list) are the three major "bulk" data constructors [3, 5, 15]. While tuples and sets have been extensively studied [1, 2, 4, 9, 14], very few investigations about sequences in databases have been reported, and notably, no declarative query system on sequences has ever been proposed in the literature. The purpose of this paper is to introduce such a system using regular expressions as declarative operations over sequences.

The basic idea of this paper is to use the regular expressions as patterns. One can view the patterns as describing the ways of merging several sequences or of selecting a subsequence from one sequence. Regular expressions can be used to describe many natural patterns and in a simple way. This fact leads to powerful, yet simple, declarative languages on databases with sequences. As an example of using the sequence operations in query languages, a simple data model involving sequences and its query language are

[†]This report summarizes the work supported by the Air Force Office of Scientific Research (AFOSR) grant 89-0244. A more detailed presentation, including formal proofs, will be provided in the author's Ph.D. thesis.

defined in this paper. The data model is basically a relational one but with every entry of a tuple being a sequence of (zero or more) (basic) values. This is an overly simplified complex-object database model (which contains neither sets nor nested structures). The simplicity of the model permits one to focus attention on sequences. Nevertheless, the use of the sequence operations in the query language over this simple model is quite general. The sequence operations can also be used in query languages over more complicated data models.

The rest of the paper is organized as follows. In Section 2, the motivation of the work is presented. In Section 3, the sequence operations based on regular expressions are formally defined. Also in Section 3, a type of a-transducer is defined to characterize the operations based on regular expressions and to serve as "operational semantics" of the operations. Some properties of the operations are studied in Section 3. The data model mentioned above is defined in Section 4. In Section 5, an extension of SQL, SSQL or *Structured Sequence Query Language*, is proposed using the sequence operations described in Section 3. We conclude the paper with some remarks in Section 6.

2 Background and Motivations

Sequence constructors are used in applications where order is significant. For example, consider the tour schedules in Figure 1, where the order of the cities being visited in a tour is as displayed (thus, the cities are not simply in a set). The following are two natural queries:

(1) Find all tours whose second city is Atlanta; and

(2) Find all tours visiting only the cities in Tour 356, and in the same order.

To handle these queries, at least two methods in the literature can be used.

One method is to view Figure 1 as a nested relation with arrival and departure dates as time stamps (see, for example, [6, 10, 18] and, to some extent, [7]). The order of the cities in a tour is implicit here, since the order is according to the arrival dates of the cities. Using the implicit orders, two sub-phrases may be necessary to express query (1): one to make sure that Atlanta appears in the tour, and another to see that there is one and only one city before it. This is a complicated way of dealing with such a simple query.

Another method is to directly view the cities as sequences, i.e., the order of the cities is the order they appear in the sequence. In this method, the sequence form permits the

TOUR_NO	CITY	ARRIVAL	DEPARTURE	COST
356	New York	3/14/90	3/16/90	1004
	Atlanta	3/16/90	3/18/90	
	Miami	2/18/90	3/20/90	
456	Los Angeles	3/18/90	3/20/90	1409
	San Francisco	3/20/90	3/22/90	
	Portland	3/22/90	3/25/90	
	Vancouver	3/25/90	3/26/90	
556	San Francisco	3/21/90	3/23/90	699
	Denver	3/23/90	3/25/90	

Figure 1: Tour schedules.

use of indices in expressing queries. For example, in the query language of O_2 [5], query (1) above can simply be written as:

Select x.TOUR_NO
From x **in** Tour_schedule
Where x.CITY[2]="Atlanta"

However, there is no natural way of expressing query (2) only using indices, since query (2) involves the question of whether one sequence is a subsequence of another.

In this paper, we use explicit sequences and employ regular expressions as operations on sequences. Both query (1) and (2) can then be easily written using the operations. Intuitively, a regular expression denotes a set of patterns describing how to merge several sequences into a new one. For example, $(x_1x_2)^*$ represents patterns indicating that the elements of odd numbered positions and the elements of even numbered positions of a merged sequence are from different sources (x_1 and x_2, respectively). Because of this, we use $[\![(x_1x_2)^*]\!]_2$ as a binary merging operation. For each pair of sequences of equal length u_1 and u_2, $[\![(x_1x_2)^*]\!]_2(u_1, u_2)$ is the "perfect shuffle" of the pair. See Figure 2 below.

Similarly, we can match the patterns described by a regular expression against a single sequence and extract out subsequences from some target positions. For example, we can use $(x_1 \cup x_2)^*$ to match against sequences and extract out the elements from the positions

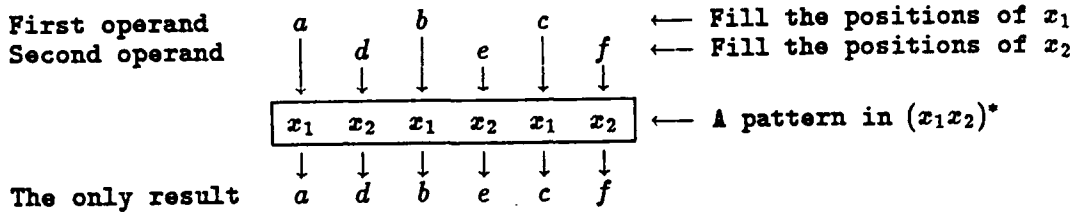

Figure 2: The merging operation $[\![(x_1x_2)^*]\!]_2$ on abc and def.

x_1 occupies. We use $[\![(x_1 \cup x_2)^*]\!]_2^{-1}$ to denote such an operation (where the superscript -1 means to extract out the elements from the positions of x_1). See Figure 3 for an example. Obviously, $[\![(x_1 \cup x_2)^*]\!]_2^{-1}(u)$ returns all subsequences of u.

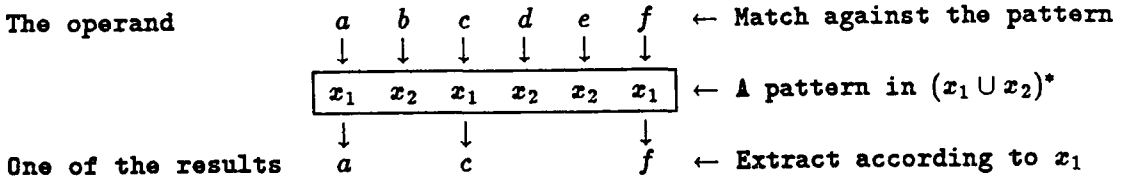

Figure 3: The selecting operation $[\![(x_1 \cup x_2)^*]\!]_2^{-1}$ on $abcdef$.

Utilizing a selecting operation, query (2) can easily be expressed as follows in SSQL (defined in Section 4):

Select TOUR_NO
From Tour_schedule
Where CITY in $[\![(x_1 \cup x_2)^*]\!]_2^{-1}$(**Select** CITY
From Tour_schedule
Where TOUR_NO=356)

in which the where clause tests whether a sequence is a subsequence of another one. Also, for each positive integer i, we may use $u[i]$ as a shorthand for $[\![x_1^{i-1}x_2x_1^*]\!]_2^{-2}(u)$, i.e., extracting the ith element of u. We can then express query (1) as easily as the query language of O_2 does.

3 Sequence Operations

In this section, we formally define a family of operations over sequences and study their properties. Throughout this section, we assume an infinite alphabet Σ_∞, i.e., an infinite set of abstract elements, and a countably infinite set of variables $\mathbf{V}_\infty$, with $\mathbf{V}_\infty \cap \Sigma_\infty = \emptyset$.

3.1 Regular Expressions as Sequence Operations

In order to define sequence operations based on regular expressions, the following notion is needed.

Notation Let x be a variable in $\mathbf{V}_\infty$ and C a finite subset of Σ_∞. Then $\mathbf{P}(x, C)$ is the set of boolean formulas with $x = c$ and $x \neq c$, where c is in C, as atomic ones.

A sequence u is said to satisfy p in $\mathbf{P}(x, C)$ if and only if each element of u satisfies p. The empty sequence ε satisfies all conditions. Let $V_n = \{x_1, \ldots, x_n\}$ be a set of n variables. Then $\mathbf{P}(V_n, C) = \bigcup_{x \in V_n} \mathbf{P}(x, C)$. Let P be a subset of $\mathbf{P}(V_n, C)$. Then $P(x_i)$ denotes the formula $(p_{i1}) \wedge \ldots \wedge (p_{ik})$ where $\{p_{i1}, \ldots, p_{ik}\} = P \cap \mathbf{P}(x_i, C)$.

To define the sequence operations, we also borrow the notions of projection π and selection σ from relational algebra. In particular, let (1) $\pi_i(u_1 \ldots u_n) = v_1 \ldots v_n$ where v_j is a projection on the ith column (in the relational algebra sense) of (the tuple) u_j for each $1 \leq j \leq n$ and (?) $\sigma_\varphi(u)$ be the maximum subsequence of u whose element satisfies the condition φ. We are now ready to define the (sequence) merging operations.

Definition 1 Let $n \geq 1$ be a positive integer, C a finite subset of Σ_∞, Q a regular expression over $V_n = \{x_1, \ldots, x_n\}$ and P in $\mathbf{P}(V_n, C)$. Then $[\![Q, P]\!]_n$ is an n-ary sequence operation, called *n-ary (sequence) merging operation*, such that for all subsets $L_1, \ldots, L_n$ of Σ_∞^*,

$$[\![Q, P]\!]_n(L_1, \ldots L_n) = \{u | \text{there exists } v \text{ in } (\Sigma_\infty \times V_n)^* \text{ such that } u = \pi_1(v), \pi_2(v) \text{ is in } L(Q) \text{ and, for each } 1 \leq i \leq n, \pi_1\sigma_{2=x_i}(v) \text{ is in } L_i \text{ and satisfies } P(x_i)\}.$$

Thus, an n-ary merging operation defines a mapping from n sets of sequences (over Σ_∞) to a single set of sequences (over Σ_∞). In a merging operation $[\![Q, P]\!]_n$, Q describes patterns used to select elements from corresponding input sequences to form merged ones, and P acts as a filter used to allow only certain input sequences participating in the merging. (When P is empty, it is usually omitted.)

Example Consider $[\![(x_1x_2)^*, (x_2 \neq f)]\!]_2(\{ab\}, \{cd, ef\})$. Since ef does not satisfy $P(x_2) \equiv (x_2 \neq f)$, ef is excluded from the merging. Therefore, only ab and cd are going to be merged according to the patterns described by $(x_1x_2)^*$. Since $x_1x_2x_1x_2$ is a word in $L((x_1x_2)^*)$, the sequence $acbd$ is obtained. It is easy to see that $acbd$ is the only result. Hence, $[\![(x_1x_2)^*, (x_2 \neq f)]\!]_2(\{ab\}, \{cd, ef\}) = \{acbd\}$.

Example It is easy to see that $[\![x_1^*x_2^*]\!]_2(L_1, L_2)$ returns the concatenation of L_1 and L_2, i.e., $[\![x_1^*x_2^*]\!]_2(L_1, L_2)$ is the set of all sequences of the form uv where u is in L_1 and v is in L_2.

Regular expressions can also be used to extract out subsequences from given sequences as follows.

Definition 2 Let $[\![Q, P]\!]_n$ be a sequence merging operation and I a subset of $\{1, \ldots, n\}$. Then $[\![Q, P]\!]_n^{-I}$ is a unary sequence operation, called an *n-ary (sequence) selecting operation*, such that for each subset L of Σ_∞^*

$$[\![Q, P]\!]_n^{-I}(L) = \{u \mid \text{there exists } v \text{ in } (\Sigma_\infty \times V_n)^* \text{ such that } \pi_1(v) \text{ is in } L,\ \pi_2(v) \text{ is in } L(Q),\ u = \pi_1\sigma_{2 \in x_I}(v) \text{ and, for each } 1 \leq i \leq n,\ \pi_1\sigma_{2=x_i}(v) \text{ satisfies } P(x_i)\}$$

where $x_I = \{x_i \mid i \text{ in } I\}$.

Thus, an n-ary selecting operation defines a mapping from a set of sequences to another set of sequences. When I consists of only a single element, we will write the element instead of the set consisting of that element.

Intuitively, in a selecting operation $[\![Q, P]\!]_n^{-I}$, Q defines a set of patterns to extract out subsequences from designated (by I) positions of input sequences, and P describes the conditions certain positions of the input sequences must satisfy.

Example Consider $[\![x_1^*x_2x_3^*, (x_2 = b)]\!]_2^{-3}(\{abcd\})$. Now (i) $x_1x_2x_3x_3$ is in $L(x_1^*x_2x_3^*)$, (ii) $abcd$ is an input sequence, and (iii) b satisfies $P(x_2) \equiv (x_2 = b)$ where b is the element in $abcd$ corresponding to x_2 in $x_1x_2x_3x_3$. Hence, elements in $abcd$ corresponding to x_3 in $x_1x_2x_3x_3$ are extracted. Therefore, cd is one of the selecting results. It is easy to see that it is the only result. Thus, $[\![x_1^*x_2x_3^*, (x_2 = b)]\!]_2^{-3}(\{abcd\}) = \{cd\}$.

Example Let k be a positive integer and L a subset of Σ_∞^*. Then $[\![x_1^kx_2^*]\!]_2^{-1}(L) = \text{Prefix}_k(L)$ and $[\![x_1^*x_2^k]\!]_2^{-2}(L) = \text{Suffix}_k(L)$.

Consider a composition of merging and selecting operations in the following example.

Example Let $F(L) = [\![(x_1x_2)^*]\!]_2^{-2}([\![(x_1x_2)^*]\!]_2([\![(x_1x_2)^*]\!]_2^{-1}(L), [\![x_1^*x_2^*]\!]_2^{-1}(L)))$. If L consists of an even length word w, then $F(L)$ returns the set consisting of the first half of w. For example, $F(\{abcd\}) = \{ab\}$. Notice that two variables are used in defining F. The same operation cannot be realized with only one variable. Later we will see that, in general, additional mappings can be realized if more variables are used.

3.2 Generic a-Transducers

We next define a type of a-transducer which gives an "operational semantics" to the sequence operations based on regular expressions. We now formally define the device.

Definition 3 A generic n-tape sequential transducer with accepting states, abbreviated *generic n-tape a-transducer,* is a 7-tuple $M_n = (n, C, K, x, H, p_0, F)$, where

(1) n is an positive integer.
(2) C is a finite subset of Σ_∞ (the constants).
(3) K is a finite set (of states).
(4) x is not in C (the variable).
(5) H is a subset of $K \times (C \cup \{x\}) \times \{1, \ldots, n\} \times K \times (C \cup \{x, \varepsilon\})$ such that for each (p_1, a, t, p_2, b) in H, either $b = a$ or $b = \varepsilon$.
(6) p_0 is in K (the start state).
(7) $F \subseteq K$ (the set of accepting states).

The variable symbol x in a generic n-tape a-transducer acts as a place holder. Whenever a symbol not in C is seen by the device it uses a transition rule containing x. Therefore, a generic a-transducer is like a pattern or a schema. The behavior of a generic a-transducer is formally defined in the following. First though, we need some auxiliary notions. We will use $\hat{d}$ to denote d or ε in the remainder of this section.

Notation Let $M_n = (n, C, K, x, H, p_0, F)$ be a generic n-tape a-transducer and A a set. For each $h = (p_1, x, t, p_2, \hat{x})$ in H and a in A, let $h[a] = (p_1, a, t, p_2, \hat{a})$ such that $\hat{a} = \varepsilon$ if and only if $\hat{x} = \varepsilon$. Let[1] $H[A] = \{h[a] | h \text{ in } H \text{ and } a \text{ in } A\}$.

[1] If h does not contain x, $h[a] = h$ for each a.

Notation Let $M_n = (n, C, K, x, H, p_0, F)$ be a generic n-tape a-transducer. Define $\vdash_{M_n}$ (or $\vdash$ when M_n is understood) to be the relation on $K \times \Sigma_\infty^* \times \ldots \times \Sigma_\infty^*$ (Σ_∞^* appears $n+1$ times) by letting

$$(p_1, w_1, \ldots, aw_t, \ldots, w_n, v) \vdash (p_2, w_1, \ldots, w_t, \ldots, w_n, vb)$$

if (p_1, a, t, p_2, b) is in $H[\Sigma_\infty - C]$. Let $\vdash^*$ be the reflexive, transitive closure of $\vdash$. Let $\vdash^k$ be the relation $(\vdash)^k$ for[2] $k \geq 0$.

We are now ready to define the mapping performed by a generic n-tape a-transducer.

Definition 4 Let $M_n = (C, K, x, H, p_0, F)$ be a generic n-tape a-transducer and L be a set of sequences on Σ_∞. Then $M_n(L_1, \ldots, L_n) = \{w | (p_0, v_1, \ldots, v_n, \varepsilon) \vdash^* (p, \varepsilon, \ldots, \varepsilon, w)$ for some p in F and v_i in L_i for all $1 \leq i \leq n\}$.

The next result shows that the "composition" of generic a-transducers is also a generic a-transducer. First though, we define a subclass of generic a-transducers.

Definition 5 A generic n-tape a-transducer $M_n = (n, C, K, x, H, p_0, F)$ is ε-free if, for each (p_1, a, t, p_2, b) in H, $b = a$. M_n is uniform if, for each pair $(p, a, t, p', \hat{a})$ and $(q, b, t, q', \hat{b})$ in H, $(p, b, t, p', \hat{b})$ is also in H.

Proposition 1 Let $k \geq 1$ be a integer, M_i be generic n_i-tape a-transducer for each $1 \leq i \leq k$ and M_0 generic k'-tape a-transducer where $k' \geq k$. Then there is a generic $(\sum_i n_i + k' - k)$-tape a-transducer M such that for all subsets L_{ij} $(1 \leq i \leq k, 1 \leq j \leq n_i)$ of Σ_∞^*, and subsets L_l $(k+1 \leq l \leq k')$ of Σ_∞^*, $M(L_1, \ldots, L_{11}, \ldots, L_{kn_k}, L_{k+1}, \ldots, L_{k'})$ $= M_0(M_1(L_{11}, \ldots, L_{1n_1}), \ldots, M_k(L_{k1}, \ldots, L_{kn_k}), L_{k+1}, \ldots, L_{k'})$. Furthermore, M is ε-free (uniform) if each M_i $(0 \leq i \leq k)$ is ε-free (uniform).

3.3 Characterizations and Properties of the Sequence Operations

In this subsection, we characterize the sequence operations in terms of the generic a-transducers and study several properties of the operations. The first result shows that ε-free uniform n-tape a-transducers have the same expressive power as the n-ary merging

[2] Let $(\vdash)^0$ be the identity relation.

operations. Notice that a mapping from $\Sigma_\infty^* \times \ldots \times \Sigma_\infty^*$ (Σ_∞^* appears n times) to Σ_∞^* is called an n-ary sequence operation.

Theorem 1 Let F be a sequence operation. Then F is equivalent to an n-ary merging operation if and only if it is equivalent to an n-tape ε-free uniform generic a-transducer.

It is easy to see that not every a-transducer is equivalent to an ε-free uniform one. Therefore, the above theorem suggests that there exists an a-transducer which has no equivalent merging operation. There are examples which show that it is indeed the case. Thus, we have:

Theorem 2 For each $n \geq 1$, there are n-ary a-transducers which have no equivalent merging operations.

Theorem 1 also shows that each merging operation has an equivalent uniform ε-free a-transducer. Therefore, by Theorem 1 and Proposition 1, the following corollary holds.

Corollary 1 Let $[\![Q_0, P_0]\!]_k$ be a k-ary merging operation and $[\![Q_i, P_i]\!]_{n_i}$ be n_i-ary merging operations for each $1 \leq i \leq k$. Then there exists a $(\sum n_i)$-ary merging operation $[\![Q, P]\!]_n$ such that $[\![Q, P]\!]_n(L_{11}, \ldots, L_{1n_1}, \ldots, L_{k1}, \ldots, L_{kn_k}) = [\![Q_0, P_0]\!]_k([\![Q_1, P_1]\!]_{n_1}(L_{11}, \ldots, L_{1n_1}), \ldots, [\![Q_k, P_k]\!]_{n_k}(L_{k1}, \ldots, L_{kn_k}))$.

The next theorem shows that the selecting operations are equivalent to unary a-transducers (not necessary uniform or ε-free ones).

Theorem 3 Let F_1 be a unary sequence operation. Then F_1 is equivalent to a generic 1-tape a-transducer if and only if it is equivalent to a selecting operation.

By the above theorem and Proposition 1, it is easily seen that the selecting operations are closed under composition.

Corollary 2 Let $[\![Q_1, P_1]\!]_n^{-I_1}$ and $[\![Q_2, P_2]\!]_n^{-I_2}$ be two selecting operations. Then there is a $[\![Q_3, P_3]\!]_n^{-I_3}$ such that $[\![Q_3, P_3]\!]_n^{-I_3}(L) = [\![Q_1, P_1]\!]_n^{-I_1}([\![Q_2, P_2]\!]_n^{-I_2}(L))$ for all L.

We now consider the following question: Can one define additional sequence operations by using more variables? The answer is positive. Instead of giving examples of the sequence operations realizable using n variables but not $n-1$ variables, we first characterize when a merging operation is decomposable, i.e., is a composition of some other merging operations. Obviously, if and only if a merging operations is decomposable, it can be realized by using fewer variables.

Notation Let V be a subset of V_n, w and u sequences over V_n and Q a regular expression such that $L(Q) = \{w\}$. Then $w[V//u] = \{w_1 | [\![Q]\!]_n^{-I_V}(w_1) = u$ and $[\![Q]\!]_n^{-I_{V_n - V}}(w_1) = w\}$ where $I_V = \{i | x_i \in V\}$ and $I_{V_n - V} = \{i | x_i \in V_n - V\}$. Given a sequence w over V_n and a subset V of V_n, let Q be a regular expression such that $L(Q) = \{w\}$. Then $w|V = [\![Q]\!]_n^{-I_V}(w)$ where $I_V = \{i | x_i \in V\}$. Let L be a subset of $\mathbf{V}_\infty^*$. Then $L|V = \bigcup_{w \in L} w|V$.

Definition 6 Let Q be a regular expression over V_n. A subset V of V_n is said to be independent in V_n with respect to Q if $L(Q) = \bigcup_{w \in L(Q)}$ and $_{v \in L(Q)|V} w[V//v]$.

Theorem 4 Let $[\![Q, P]\!]_n$ be an n-ary merging operation. Then $[\![Q, P]\!]_n$ is decomposable if and only if there is a proper subset V, with $\#(V) > 1$, of V_n such that V is independent in V_n with respect to Q.

To see whether a merging operation $[\![Q, P]\!]_n$ is not decomposable, we thus only need to test that each proper subset of V_n of arity greater than 1 is not independent with respect to Q. For example, $[\![(x_1x_2 \cup x_2x_3)^*]\!]_3$ is easily seen not decomposable. It is also easy to see that for each $n > 2$, $[\![(x_1x_2 \cup x_2x_3 \cup \cdots \cup x_{n-1}x_n)^*]\!]$ is not decomposable. Hence, we have the following result.

Theorem 5 For each $n > 2$, there is an n-ary merging operation which is not decomposable.

Let $[\![Q, P]\!]_n^{-I}$ be a selecting operations. If P is empty, then it is easily seen that there is a regular sequence Q_1 over V_2 such that $[\![Q_1]\!]_2^{-1}$ is equivalent to $[\![Q, P]\!]_n^{-I}$. If P is not empty, then we can see that the more variables used, the more mappings can be realized.

4 Tuple Sequence Database Model

In this section, a data model is described which is basically relational, but uses sequences of [illegible] as entry elements for [illegible].

In the following, $\mathcal{U}$ is assumed to be a non-empty set of attribute names. Members of $\mathcal{U}$ are denoted by A, B and C etc., possibly subscripted. For each A in $\mathcal{U}$, there is an associated non-empty set DOM(A). Let $\mathcal{R}$ denote the set of all finite subset of $\mathcal{U}$. Each member of $\mathcal{R}$ is a relational schema and is denoted by R, possibly subscripted. Each finite subset of $\mathcal{R}$ is called a database schema.

In the Tour-schedule example, suppose $\mathcal{U}$ contains TOUR_NO, CITY, ARRIVAL, DEPARTURE and COST. Then {TOUR_NO, CITY, ARRIVAL, DEPARTURE, COST} is a member of $\mathcal{R}$, and therefore a relational schema.

For each A in $\mathcal{U}$, Seq(DOM(A)) is the set of finite sequences of elements from DOM(A), i.e., Seq(DOM(A)) = $\{a_1 \ldots a_n | n \geq 1$ and a_i in DOM(A) for each $1 \leq i \leq n\} \cup \{\epsilon\}$, where ϵ is the empty sequence. A tuple t of $R = \{A_1, \ldots, A_m\}$ in $\mathcal{R}$ is a total function from R to $\bigcup_{i=1}^{m}$ Seq(DOM(A_i)) such that $t(A_i)$ is in Seq(DOM(A_i)) for each $1 \leq i \leq m$.

A tuples is usually presented in table form. In the tour schedule example, San Francisco and Denver are in DOM(CITY). Therefore, the sequence "San Francisco, Denver" is in Seq(DOM(CITY)). Similarly, 556 is in DOM(TOUR_NO) and the sequence"556" in Seq(DOM(TOUR_NO)). Hence, we have the following tuple:

TOUR_NO.	CITY	ARRIVAL	DEPARTURE	COST
556	San Francisco	3/21/90	3/23/90	699
	Denver	3/23/90	3/25/90	

An instance of a relational schema R is a finite set of tuples over R. A database instance I of a database schema $\{R_1, \ldots, R_k\}$ is a sequence $I_{R_1}, \ldots, I_{R_k}$ such that I_{R_i} is an instance of R_i for each $1 \leq i \leq k$. In the tour schedule example, the table in Figure 2 is an instance of the relational schema {TOUR_NO, CITY, ARRIVAL, DEPARTURE, COST}.

The model presented above is a proper extension of the relational model [8]. Further extensions may be necessary for practical purposes. For example, instead of using sequences of only basic values, we may allow sequences of tuples as elements of another tuple. However, for simplicity, we only consider the simple model presented.

5 SSQL: an Extension of SQL

In this section, we propose an extension of SQL, SSQL or *Structured Sequence Query Language*, as a query language over the data model defined in the last section. The

query language is essentially SQL but with sequence operations appearing in the Select and Where clauses in original SQL queries. Because of the length restriction, we only consider the select statement. Also, we will be quite informal in presenting SSQL. In the following, a sequence operation F is called single valued if $F(U_1, \ldots, U_n)$ contains only a single sequence whenever U_i is a set of one sequence for all $1 \leq i \leq n$.

We begin with an example to show the basic features of SSQL. The syntax will be defined later. Consider the query "Display the tour numbers and the second cities of all tours visiting only the cities in Tour 356 and in the same order." In SSQL, the query can be expressed as follows.

```
(1) Select  TOUR_NO, CITY[2]
(2) From    Tour_schedule
(3) Where   CITY in [[(x1 ∪ x2)*]]2^-1(
(4)                                    Select  CITY
(5)                                    From    Tour_schedule
(6)                                    Where TOUR_NO=356 )
```

Lines (4)–(6) are as an ordinary SQL query and return the sequence of cities of tour 356. Line (3) is another form of $u \in [\![x_1 \cup x_2]\!]_2^{-1}(w)$ where u is the list of the cities in question and w is the list of the cities returned from lines (4)–(6). Therefore, line (3) tests if the cities in question is a subsequence of the cities of tour 356. Finally, line (1) returns the tour number which satisfies the test of line (3) and the second city of the tour. Note that CITY[2] is a shorthand of $[\![x_1 x_2 x_1^*]\!]_2^{-2}(\text{CITY})$.

Notice that in the above example, we used sequence operations in a set membership test and in the select clause. In general, an SSQL select statement is in the form:

Select $F_1(R_{11}.A_{r_1}, \ldots, R_{1r_1}.A_{r_1}), \ldots, F_l(R_{l1}.A_{r_l}, \ldots, R_{lr_l}.A_{r_l})$
From $R_1, \ldots, R_k$
Where Ψ

where F_i is a r_i-ary sequence operation for each $1 \leq i \leq l$ (notice that the attribute names in a specific F_i are all the same and if F_i is the unary operation $[\![x_1^*]\!]_1$, then it is usually omitted), Ψ is a formula involving logical connectives **and**, **or** and **not**, arithmetic comparison operators $=$, $<=$, and so on, set comparison operators $\subseteq$ and $\supseteq$ etc., and the membership test **in**. When the arithmetic comparison operators are used, each element

of the sequence on the left hand side is compared to each element of the sequence on the right hand side, and the result of the arithmetic comparison is true if and only if all these comparisons are true. A (nested) select statement can appear in either side of a *set comparison* and the right side of **in**. Single valued sequence operations can be used on the sequences on either side of all comparisons, while other sequence operations can only be used on sequences on either side of a set comparison and on the right side of a membership test. Finally, only one attribute can appear in select clause of a nested statement.

As SQL queries, SSQL queries can be interpreted in tuple calculus forms. The general form of SSQL query can be translated into the following:

$$\{s|(\exists t_1,\ldots,t_k)(R_1(t_1)\wedge\ldots R_k(t_k)\wedge\Psi\wedge s.A_{r_1}\in P_1(t_{11}.A_{r_1},\ldots,t_{1r_1}.A_{r_1})\wedge\ldots \\ \wedge s.A_{r_l}\in P_l(t_{l1}.A_{r_l},\ldots,t_{lr_l}.A_{r_l}))\}.$$

The formal definition and detailed discussion of the tuple calculus over the data model presented in Section 3 is omitted from this paper. However, we can see that the semantics of SSQL is very similar to that of SQL.

We will not go into a detailed discussion of SSQL here. Instead, we give some example queries written in SSQL. The examples are based on the tour schedule example in Section 1 (see Figure 2).

Example "List all pairs of tours such that the time periods of the two tours do not overlap."

Select t_1.TOUR_NO, t_2.TOUR_NO
From Tour_schedule t_1, Tour_schedule t_2
Where $[\![x_1^* x_2]\!]_2^{-2}(t_1.\text{DEPARTURE}) <= t_2.\text{ARRIVAL}[1]$

Following the convention of SQL, t_1 and t_2 are aliases for the relation Tour_schedule.

Example "Find all pairs of tours such that the first city of the second tour is within the first tour."

Select t_1.TOUR_NO, t_2.TOUR_NO
From Tour_schedule t_1, Tour_schedule t_2
Where $t_2.\text{CITY}[1]$ **in** $[\![x_1^* x_2 x_1^*]\!]_2^{-2}(t_1.\text{CITY})$

This example shows that testing whether an element is in a sequence can be expressed easily.

Example "Find out all tours in which Los Angeles and San Francisco are visited consecutively."

Select TOUR_NO
From Tour_schedule
Where San Francisco in $[\![x_1^* x_2 x_3 x_1^*, x_2 = \text{Los Angeles}]\!]_3^{-3}(\text{CITY})$ **or**
Los Angeles in $[\![x_1^* x_2 x_3 x_1^*, x_2 = \text{San Francisco}]\!]_3^{-3}(\text{CITY})$

Example "List all tours which visit at least two more cities between Los Angeles and San Francisco."

Select TOUR_NO
From Tour_Schedule
Where CITY in $[\![x_1^* x_2 x_3 x_3^+ x_4 x_1^*, x_2 = \text{Los Angeles}, x_4 = \text{San Francisco}]\!]_4^{-\{1,2,3,4\}}(\text{CITY})$

Example "List all tours which visit an even number of cities."

Select TOUR_NO
From Tour_Schedule
Where CITY in $[\![(x_1 x_2)^*]\!]_2^{-\{1,2\}}(\text{CITY})$

Example "List all tours which visit the same number of cities as Tour 456."

Select TOUR_NO
From Tour_Schedule
Where CITY in $[\![(x_1 x_2)^*]\!]_2^{-1}([\![(x_1 x_2)^*]\!]_2(\text{CITY},$
Select CITY
From Tour_Schedule
Where TOUR_NO = 456))

6 Related Research and Conclusion

One of the papers in the literature devoted entirely to the sequences in databases is [12]. Indeed, the only data constructor in the data model of [12] is the sequence. An algebraic query language is proposed on the nested sequences in the data model. However, the selection of operations is *ad hoc* in nature. There is no single formal mathematical system behind the operations. Nevertheless, [12] introduces many powerful operations which cannot be simulated by the sequence operations of this paper. It may be interesting to see how to extend the sequence operations of this paper to include them.

The Tangram stream query processing system [16, 17] is an attempt to use streams (or sequences) as a common processing model both in AI systems and database systems. The relations in a relational database are viewed as sequences of tuples rather than sets of tuples. Query processing then becomes sequence processing. A powerful combination of logic and functional programming language Log(F) is used as a base system in stream (sequence) processing. Since Log(F) is a general programming language, all computable transformations of sequences, including the highly intractable ones, are possible. However, it is not clear how tuples and sets can retain their own characteristics in Log(F).

Our interest in the queries on sequences stems from the study of a group of context-related interval queries [11]. However, the focus of [11] is on the preservation of the computation-tuple sequence schemes. It is worth noting that all queries in [11] can basically be simulated by the sequence operations defined in this paper.

The contribution of this paper is the introduction of the first family of declarative sequence operations based on a theoretically sound formalism, namely, regular expressions. It is shown, through examples, that sequence operations are quite natural as well as powerful.

Using sequence operations in query languages is straightforward. We illustrate this in a simple extension of SQL. Actually, the way of using the sequence operations in SSQL is rather general. In the same way, we can extend various query languages on complex objects to include these sequence operations to handle sequences.

Several important questions remain to be answered. One of them is the completeness of these operations. In what sense is a set of sequence operations complete? Obviously, there are still many conceivable operations not representable in the family introduced in this paper. Hence, another question is how to extend the family.

References

[1] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. Technical Report 846, INRIA, May 1988.

[2] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, 1987.

[3] A. Albano, L. Cardelli, and R. Orisini. Galileo: A strongly typed language for complex objects. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[4] F. Bancihon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD: a powerful and simple database language. In *Proc. of the 13th VLDB Conference*, pages 97–105, Brighton, 1987.

[5] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the O_2 object-oriented database system. In *Database Programming Languages: 2nd International Workshop.* Morgan-Kaufmann, Inc., June 1989.

[6] J. Clifford and A. U. Tansel. On an algebra for historical relational databases: two views. In *Proc. of 1985 ACM SIGMOD International Conference on Management of Data*, pages 247–265, 1985.

[7] J. Clifford and D. S. Warren. Formal semantics for time in databases. *ACM Transactions on Database Systems*, 8(2):214–254, June 1983.

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

[9] P. C. Fisher and S. J. Thomas. Operators for non-first-normal-form relations. In *Proc. IEEE Computer Software Applications conference*, pages 464–475, 1983.

[10] S. K. Gadia and J. H. Vaishnav. A query language for a homogeneous temporal database. In *Proc. of 4th ACM Symp. on Principles of Database Systems*, pages 51–56, 1985.

[11] S. Ginsburg, D. Simovici, and X. Wang. Content-related interval queries on object histories. Accepted for publication in *Information and Computation.*

[12] R. H. Güting, R. Zicari, and D. M. Choy. An algebra for structured office documents. Technical Report RJ 5559 (56648), IBM Almamden Research Center, 1987.

[13] J. E. Hopcroft and J. D. Ullman. *Formal languages and their relation to automata.* Addison-Wesley, 1969.

[14] R. Hull and J. Su. On the expressive power of database queries with intermediate types. Technical Report 88-53, Computer Science Department, University of Southern California, 1988.

[15] R. Hull and J. Su. On bulk data type constructors and manipulation primitives: a framework for analyzing expressive power and complexity. In *Database Programming Languages: 2nd International Workshop*, pages 396–410. Morgan-Kaufmann, Inc., 1989.

[16] D. S. Parker. Integrating AI and DBMS through stream processing. In *Proc. of the 5th International conference on Data Engineering*, pages 259–260, 1989.

[17] D. S. Parker, R. R. Muntz, and H. L. Chau. The Tangram stream query processing system. In *Proc. of the 5th International conference on Data Engineering*, pages 556–563, 1989.

[18] R. Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

Properties of Spreadsheet Histories*

Stephen Kurtzman

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0782

Abstract

This report looks at two questions about the Spreadsheet History Model. First, several operations on *spreadsheet histories* (selection, projection, cohesion, intersection, and union) which are analogues of some common relational-database operations are examined. The primary question of concern is whether or not their results are *SHS representable*. Necessary and sufficient conditions are presented for those operators which do not always preserve the SHS formalism. The second question concerns a notion of spreadsheet-history equivalence based on the computation-tuple-sequence projection operation (*projection simulation*). Three sufficiency conditions are given for determining when projection simulation preserves the history-bounded SHS.

* This report summarizes the results concerning the spreadsheet-history model which were obtained under funding from the Air Force Office of Scientific Research (AFOSR) grant 89-0244. A more detailed presentation will appear in [K 91].

1 Introduction

One of the most widely used types of small-business data-processing software is the spreadsheet program [DLL 88; Go 87; WS 86]. The broad appeal of the spreadsheet is due to its straightforward tabular method for describing computational relationships between data. Spreadsheet programs, such as C-Calc [DSD], Excel [Ms 89], and Lotus 1-2-3 [Lot 85], have automated the design and use of spreadsheets. While there have been discussions on spreadsheet-programming methodologies [Be 86; RPL 89], little of a theoretical nature is known about spreadsheets themselves. The purpose of our research is to rigorously examine a data model for describing *spreadsheet histories* and their properties.

The present report consists of four sections, including this introduction. Section 2 gives the basic definitions of the spreadsheet-history model. The model describes the use of spreadsheets to represent historical, accounting-like data. It consists of sequences of *computation tuples* defined by a *spreadsheet-history scheme* (SHS).

Section 3 examines several operations on spreadsheet histories (selection, projection, cohesion [GTa 89], intersection, and union) which are analogues of some common relational-database operations [Co 70; Ul 82; Ma 83]. The primary question of concern here is whether or not their results are SHS *representable*. The answer is yes for selection, cohesion, and intersection, and no for the others. A necessary and sufficient condition is given for projection and union to characterize when each preserves the SHS model. Some additional characteristics of selection, projection and cohesion are also presented.

Section 4, concerns a notion of spreadsheet-history equivalence based on the computation-tuple-sequence projection operation. The concept of projection simulation is presented and three sufficiency conditions are given for determining when projection simulation preserves the history-bounded SHS.

2 Spreadsheet Histories

In this section a formal model for spreadsheets and their histories is introduced.

In simple terms, a spreadsheet is a finite set of related data. Each datum occupies a unique location and is either specified directly, using a constant, or indirectly, using a function. Each datum location is called a *cell*. In Microsoft Excel, the cells are arranged in a 16,384 row by 256 column rectangle, and are addressed by row and column indices. In principle, the number of rows and columns in a spreadsheet may be arbitrarily large. The functions are written in terms of the data locations in the spreadsheet.

EXAMPLE 1.1: Consider a spreadsheet representation of a stock-purchase history. Figure 1. 1 shows the spreadsheet as it might appear using the Microsoft Excel program. Row 1 contains text that indicates the meaning of the data in each column. Each of the rows 2 through 12 contains a single stock transaction. For each transaction:

(1) Information is recorded for:

DATE — The date on which the transaction occurred.

TRANS — The transaction type, either BUY, SELL, or DIV(IDEND). For simplicity, the dividend transaction only records dividends which are disbursed as shares of stock.

SHARES — The number of shares of stock involved in the transaction. Again, for simplicity, only whole shares of stock may be entered.

PSV — The per-share value of the stock for the current transaction (i.e. the buy or sell price).

(2) And values are calculated for:

VALUE — The total dollar value of the transaction.

PROFIT — The profit earned for the transaction. (A negative value indicates a loss.) If TRANS is BUY, then the profit is zero. If TRANS is DIV, then the profit is equal to the VALUE. If TRANS is SELL, then the profit requires a more complicated calculation. According to tax laws, when a share of stock is sold, the profit is equal to the price received minus the original

	A	B	C	D	E	F	G
1	DATE	TRANS	SHARES	PSV	VALUE	PROFIT	CUMSH
2	7/2/86	BUY	1000	$6.00	$6,000.00	$0.00	1000
3	8/15/86	BUY	2000	$5.50	$11,000.00	$0.00	3000
4	9/10/86	BUY	3000	$5.00	$15,000.00	$0.00	6000
5	12/31/86	DIV	50	$6.00	$300.00	$300.00	6050
6	1/20/87	BUY	4000	$3.00	$12,000.00	$0.00	10050
7	6/30/87	DIV	40	$5.00	$200.00	$200.00	10090
8	12/31/87	DIV	50	$5.00	$250.00	$250.00	10140
9	6/30/88	DIV	60	$8.00	$480.00	$480.00	10200
10	8/13/88	SELL	5000	$10.00	$50,000.00	$23,000.00	5200
11	12/31/88	DIV	30	$8.00	$240.00	$240.00	5230
12	2/22/89	SELL	3000	$11.00	$33,000.00	$21,850.00	2230

Figure 1. 1

price paid. When selling stock you are required to sell in a first-in first-out (FIFO) order.

CUMSH The cumulative number of shares of stock owned at the completion of the current transaction.

The data in cells A2, B2, C2, and D2 are entered as input. The numbers in cells E2 and F2 are calculated using the formulas (written in the notation employed by Microsoft Excel) =D2*C2, and =SMacs!TProfit(ROW()), respectively and in the specified order. [The formula ROW() returns the row number of the current cell and SMacs!TProfit() invokes the TProfit macro stored in the file named "SMacs." The TProfit macro is a user-written function that calculates the profit (using the FIFO formula required by the tax laws) for the transaction recorded on the row number passed to it.] The value in G2 is also entered as input, but because of the nature of the application, the value in G2 must agree with the value in C2.

The data in line 3 represents the second event in the stock-purchase history. The information in cells A3 through D3 is specified directly. The data in cells E3, F3 and G3 are calculated by the formulas =D3*C3, =SMacs!TProfit(ROW()), and =IF(B3="SELL",G2-C3,G2+C3) respectively, and in the specified order. [The formula =IF(*cond*,*expr1*,*expr2*) evaluates the logical expression *cond* and returns value of *expr1* if *cond* is true and the value of *expr2* if *cond* is false.] Note that the formula in cell E3 is a relativized version of

the one found in cell E2. Most spreadsheet programs provide a command to copy the formulas from one column to another in this relativized fashion — see the "copy" and related commands in [DSD; Lot 85; Ms 89].

For the remaining lines, the data in columns A, B, C and D are input as constants, the numbers in columns E and G are calculated using relativized versions of the formulas in E3 and G3 respectively, and the values in column F are calculated using the same formula as found in cell F3. The calculations are performed for in row number order. □

Each line (except 1) of the spreadsheet in Figure 1. 1 represents a single event in the stock-purchase history. In use, a new line is added to the spreadsheet each time some stock is received or sold. The history is modeled by a sequence of events or transactions. This type of historical data modeling also occurs in many other spreadsheet applications, for example the checkbook-management spreadsheets in [CA; DSD], the sales, cash-flow, and budget-forecasting spreadsheets in [CA], and the "what-if" models in [Jo 89].

In this simplified example, each stock transaction is kept on a single line of the spreadsheet. For a more detailed accounting, it may be preferable to display each event on a single spreadsheet. Such details of data display are important human-factors concerns, but are irrelevant for the analysis carried out in this manuscript.

Before proceeding to the formal model, some preliminary definitions are in order. It is assumed that there exists an infinite set of *domain values* (denoted Dom_∞) and an infinite set of *attributes* (denoted U_∞). The set U_∞ is partitioned into two infinite disjoint sets, I_∞ and E_∞, respectively called *input* attributes and *evaluation* attributes. For each A in U_∞, Dom(A) is a subset of Dom_∞ of at least two elements. All attributes are assumed to be elements of U_∞. The symbols A, B, and C (possibly subscripted or primed) will denote attributes and U, V, and W (possibly subscripted or primed) will denote nonempty, finite sets of attributes. As is customary in the relational database literature, the union sets of attributes will be denoted by juxtaposition. So, UV shall mean $U \cup V$. And, committing a slight abuse of notation, AB and UA shall respectively mean $\{A\} \cup \{B\}$ and $U \cup \{A\}$.

There is a total order $<_\infty$ over U_∞ such that (i) $A <_\infty B$ for each A in I_∞ and B in E_∞; and (ii) for each B in E_∞, there exists a B′ in E_∞ such that $B <_\infty B'$. Let X be a finite non-

empty subset of U_∞ and $A_1, \ldots, A_n$ the listing of the elements of X according to $<_\infty$. Then <X> will denote the sequence $A_1 \cdots A_n$ and Dom(<X>) the Cartesian product $Dom(A_1) \times \cdots \times Dom(A_n)$. For $i \geq 2$, $<X|A_i>$ denotes the prefix $A_1, \ldots, A_{i-1}$. [A prefix of a sequence $p_1 \ldots p_m$ is a subsequence of the form $p_1 \ldots p_i$ for some i, $1 \leq i \leq m$.]

An important aspect of Example 1. 1 is the regularity of form exhibited by each line of data in Figure 1. 1. To capture this detail in the formal model, each line will be considered as constituting a separate spreadsheet and each spreadsheet will be represented by a single *computation tuple* defined over a finite set of attributes. The cells of a spreadsheet which are specified by constants will be modeled by *input attributes* while those cells which have values determined by functions will be represented by *evaluation attributes*. The entire sequence of spreadsheets will be represented by a *computation-tuple sequence*.

Segmenting the attributes into inputs and evaluations reflects the different roles played by the cells in the spreadsheet. This partitioning will be specified by an *attribute scheme*. An *attribute scheme* over <U> is an ordered pair (<I>, <E>), where <U> = <I><E>, $I = I_\infty \cap U \neq \emptyset$ and $E = E_\infty \cap U \neq \emptyset$. [Given sequences of attributes $<U_1> = A_1, \ldots, A_m$ and $<U_2> = B_1, \ldots, B_n$, $<U_1><U_2>$ will denote the concatenation of the sequences, i.e., $A_1, \ldots, A_m, B_1, \ldots, B_n$.] That is, an attribute scheme divides <U> into a sequence of input attributes, I, and a sequence of evaluation attributes, E.

A *computation tuple* over <U> is an element in Dom(<U>). A *computation-tuple sequence* over <U> is a finite, nonempty sequence of computation tuples over <U>. The set of all computation-tuple sequences over <U> is denoted by SEQ(<U>). For each <U> and $\rho \geq 1$, $SEQ(<U>, \rho) = \{\bar{u} \text{ in } SEQ(<U>) \mid |\bar{u}| \geq \rho\}$. [The length of a computation-tuple sequence $\bar{u}$ is denoted $|\bar{u}|$.]

The symbol Λ will denote the *empty sequence*, that is, the sequence which contains no tuples. For each <U>, $SEQ(<U>, 0) = \{\Lambda\} \cup SEQ(<U>, 1)$.

Unless otherwise stated, u, v, and w (possibly subscripted or primed) represent computation tuples. Similarly, $\bar{u}$, $\bar{v}$, and $\bar{w}$, represent computation-tuple sequences. And $\tilde{u}$, $\tilde{v}$, and $\tilde{w}$, represent either computation-tuple sequences or the empty sequence. The catenation of sequences and tuples will be denoted by juxtaposition.

Let u be a computation tuple over <U> and A an attribute in U. The value of u on A will be denoted by $u(A)$. If <V> is a subsequence of <U>, then $u[<V>]$ and $\pi_V(u)$ both denote the tuple v over <V> with $v(A) = u(A)$ for each A in V. The tuple v is called the *projection of u onto* <V>. For each $\bar{u} = u_1...u_n$ in SEQ(<U>), let $\pi_V(\bar{u}) = \pi_V(u_1)...\pi_V(u_n)$. For the empty sequence Λ, we define $\pi_V(\Lambda)$ to be Λ. For each[1] $\mathcal{U} \subseteq$ SEQ(<U>), let $\pi_V(\mathcal{U}) = \{\pi_V(\bar{u}) \mid \bar{u} \text{ in } \mathcal{U}\}$. Note that π_V is the computation-tuple sequence analogue of the relational database projection operator [Co 70].

The first component of the spreadsheet-history model is the spreadsheet scheme.

DEFINITION: A *spreadsheet scheme* over <U> is a triple S = (<I>, <E>, $\boldsymbol{S}$), where

- (<I>, <E>) is an attribute scheme over <U>; and
- $\boldsymbol{S} = \{s_C \mid C \text{ in } E, s_C \text{ is a partial recursive function from SEQ}(<U>, \rho(s_C)) \times \text{Dom}(<U \mid C>) \text{ to Dom}(C), \text{ where } \rho(s_C) \geq 0\}$. □

The functions in $\boldsymbol{S}$ are called *spreadsheet functions*. The number $\rho(s_C)$ is called the *rank* of s_C and $\rho(S) = max\{\rho(s_C) \mid C \text{ in } E\}$ the *rank* of S.

The rank determines the number of computation tuples which must exist in a sequence before the spreadsheet function can be applied.

The purpose of a spreadsheet scheme is to define a set of "valid" spreadsheet sequences, that is, sequences which are consistent with the functions in the scheme.

DEFINITION: Let S = (<I>, <E>, $\boldsymbol{S}$) be a spreadsheet scheme over <U>. For each C in E, denote the set of sequences *valid with respect to* s_C by[2] $VSEQ(s_C) = \{u_1...u_n \mid u_i(C) = s_C(u_1...u_{i-1}, u_i[<U \mid C>]) \text{ for all } \rho(s_C) < i \leq n\}$; and for each $\emptyset \neq E' \subseteq E$, let $VSEQ(E') = \bigcap_{C \text{ in } E'} VSEQ(s_C)$. Let VSEQ(S) = VSEQ($\boldsymbol{S}$). □

Given a spreadsheet function s_C, every sequence in SEQ(<U>) of length at most $\rho(s_C)$ is in $VSEQ(s_C)$.

DEFINITION: For each $\mathcal{U} \subseteq$ SEQ(<U>) and positive integer k, let $prefix(\mathcal{U}) = \{\bar{u}' \mid \bar{u}' \text{ is a prefix of some } \bar{u} \text{ in } \mathcal{U}\}$ and $prefix^k(\mathcal{U}) = \{\bar{u}' \text{ in } prefix(\mathcal{U}) \mid |\bar{u}'| \leq k\}$. If $\mathcal{U} = prefix(\mathcal{U})$ then $\mathcal{U}$ is said to be *prefix closed*.

---

1. The symbol $\subseteq$ denotes set inclusion while $\subset$ denotes proper set inclusion.

2. For $\rho(s_C) = 0$ and $i = 1$, $u_1...u_{i-1} = \Lambda$.

Notice that VSEQ(s_C) is prefix closed. Thus, VSEQ(S) is also prefix closed.

To complete the spreadsheet-history model, there must be a mechanism to provide the evaluation-attribute values at the beginning of a computation-tuple sequence. This is done using a prefix-closed set of sequences of length at most the rank of the spreadsheet scheme.

DEFINITION: Given a spreadsheet scheme S over <U>, an *initialization (with respect to S)* is a recursively enumerable, prefix-closed subset $\boldsymbol{I}$ of $\{\bar{u}$ in VSEQ(S) $\mid |\bar{u}| \leq max\{1, \rho(S)\}\}$. Given an initialization $\boldsymbol{I}$, let VSEQ($\boldsymbol{I}$) denote the set $\boldsymbol{I} \cup \{\bar{u}$ in SEQ(<U>) $\mid \bar{u} = \bar{u}_1\bar{u}_2$ for some $\bar{u}_1$ in $\boldsymbol{I}$ of length $\rho(S)\}$. □

Clearly, VSEQ($\boldsymbol{I}$) is prefix closed.

A set of spreadsheet histories is defined by a *spreadsheet-history scheme*. Formally:

DEFINITION: A *spreadsheet-history scheme* (abbreviated SHS) over <U> is an ordered pair H = (S, $\boldsymbol{I}$), where

- S is a spreadsheet scheme over <U>; and
- $\boldsymbol{I}$ is an initialization with respect to S.

Let $\rho(H)$, called the *rank* of H, be $max\{1, \rho(S)\}$. □

An SHS determines valid spreadsheet histories as follows:

DEFINITION: For each SHS H = (S, $\boldsymbol{I}$) let VSEQ(H) = VSEQ(S) ∩ VSEQ($\boldsymbol{I}$). A spreadsheet sequence is said to be *valid (for H)* if it is in VSEQ(H). □

Since both VSEQ(S) and VSEQ($\boldsymbol{I}$) are prefix closed, so is VSEQ(H).

EXAMPLE 1.1 (continued): The stock-purchase history can be recast using the formal model. The labels in line 1 of Figure 1. 1 will be ignored since they do not enter into any of the calculations. Each of the other lines will be represented by spreadsheets (computation tuples). An SHS over <U> for the income history is H = ((<I>, <E>, $\boldsymbol{S}$), $\boldsymbol{I}$), where

- <I> = <DATE, TRANS, SHARES, PSV>.
- <E> = <VALUE, PROFIT, CUMSH>.
- The domains of the attributes are the obvious ones.

$s_{VALUE}(u_1 \ldots u_n, u_{n+1}[\langle U \mid VALUE \rangle]) = u_{n+1}(PSV) \times u_{n+1}(SHARES)$

$s_{PROFIT}(u_1 \ldots u_n, u_{n+1}[\langle U \mid PROFIT \rangle]) =$

$$\begin{cases} f(\pi_I(u_1 \ldots u_n), u_{n+1}[\langle I \rangle]) & \text{if } u_{n+1}(TRANS) = SELL \text{ and } (u_{n+1}(SHARES) \le u_n(CUMSH)) \\ \text{undefined} & \text{if } u_{n+1}(TRANS) = SELL \text{ and } (u_{n+1}(SHARES) > u_n(CUMSH)) \\ u_{n+1}(VALUE) & \text{if } u_{n+1}(TRANS) = DIV \\ 0 & \text{if } u_{n+1}(TRANS) = BUY \end{cases}$$

$s_{CUMSH}(u_1 \ldots u_n, u_{n+1}[\langle U \mid CUMSH \rangle]) =$

$$\begin{cases} u_n(CUMSH) - u_{n+1}(SHARES) & \text{if } u_{n+1}(TRANS) = SELL \text{ and } (u_{n+1}(SHARES) \le u_n(CUMSH)) \\ \text{undefined} & \text{if } u_{n+1}(TRANS) = SELL \text{ and } (u_{n+1}(SHARES) > u_n(CUMSH)) \\ u_n(CUMSH) + u_{n+1}(SHARES) & \text{otherwise.} \end{cases}$$

Figure 1. 2. Function definitions for Example 1. 1

- I = { u in Dom(<U>) | u(TRANS) = BUY, u(SHARES) > 0, u(VALUE) = u(PSV) × u(SHARES), u(PROFIT) = 0, and u(CUMSH) = u(SHARES). }

- The functions in S = { s_{VALUE}, s_{PROFIT}, s_{CUMSH} } are defined for each $u_1 \ldots u_n$ in SEQ(<U>, 1) and tuple u_{n+1} in Dom(<U>) as shown in Figure 1. 2.

In Figure 1. 2, $f(\pi_I(u_1 \ldots u_n), u_{n+1}[\langle I \rangle])$ is the function from SEQ(<I>, 0) × Dom(<I>) into Dom(PROFIT) which returns the profit for the sale of stock using the FIFO method required by the tax laws. This is a straightforward, albeit detailed, calculation The

earliest-purchased unsold stock can be found by summing the most recently received shares until the total equals or exceeds the current number of shares owned. The specifics are omitted. (Note that f is defined solely over input-attribute values even though a program for f could make use of some of the VALUE and CUMSH attributes in the sequence.) □

In the spreadsheet-history model, a spreadsheet function is defined with respect to a computation-tuple sequence. In real-world applications, a spreadsheet function is often determined solely by a bounded number of computation tuples at the end of the sequence. Such *history bounded* functions represent those spreadsheet functions which, in one sense, can be implemented efficiently. More formally:

DEFINITION: Let s_C be a spreadsheet function in the spreadsheet scheme S = (<I>, <E>, $\boldsymbol{S}$) over <U> and k a non-negative integer. If $s_C(\tilde{u}\tilde{v}, w[<U|C>]) = s_C(\tilde{v}, w[<U|C>])$ for all sequences $\tilde{u}\tilde{v}w$ in SEQ(<U>), where $|\tilde{v}| = k$, then s_C is said to be *k-history bounded*. A spreadsheet function is said to be *history bounded* if it is k-history bounded for some k. If all spreadsheet functions in S are k-history bounded, then S is also said to be *k-history bounded*. Likewise, S is said to be *history bounded* if it is k-history bounded for some k.

If s_C is k-history bounded, then $k \geq \rho(s_C)$. Suppose s_C is k-history bounded for some k. If is easily seen that s_C is also m-history bounded for all $m \geq k$.

DEFINITION: An SHS H = (S, $\boldsymbol{I}$) over <U> is said to be *k-history bounded* (*history bounded*) if S is k-history bounded (history bounded). □

In Example 1. 1 the spreadsheet functions s_{VALUE} and s_{CUMSH} are both history bounded, but s_{PROFIT} is not. Thus, SHS is not history bounded.

In the sequel, the primary concern will be to examine the conditions under which operations on sets of spreadsheet histories preserve the SHS formalism. Consequently, before proceeding to the technical results, we need one final definition.

DEFINITION: A set $\mathcal{U}$ or an SHS H is said to be (*history-bounded-*) *SHS representable* if there exists a (history-bounded) SHS H′ such that $\mathcal{U}$ = VSEQ(H′) or VSEQ(H) = VSEQ(H′) respectively. □

3 Relational Operators

This section presents some spreadsheet-history analogues to the relational-database operators [Co 70; Ul 82; Ma 83].

We first address a selection operator analogue, called the *historical-selection operator*. Historical-selection operators are unary functions from $2^{SEQ(<U>)}$ to $2^{SEQ(<U>)}$ which return prefix-closed sets of histories. Our first result shows that the historical-selection operator preserves SHS representability. The last result demonstrates the undecidability of determining when a historical-selection operator maps a set to itself.

DEFINITION: For each computable mapping Θ from SEQ(<U>) to {true, false}, let σ_Θ be the function defined for each subset $\mathcal{U}$ of SEQ(<U>) by $\sigma_\Theta(\mathcal{U})$ = prefix({$\bar{u}$ in $\mathcal{U}$ | $\Theta(\bar{u})$ = true}). The function σ_Θ is called a *historical selection operator*. □

In the preceding definition, the mapping Θ acts as a selection criterion to pick a subset of the histories in $\mathcal{U}$, namely, those histories $\bar{u}$ for which $\Theta(\bar{u})$ = true. The query σ_Θ takes the prefix closure of the collection chosen by Θ. The query is "historical" in the sense that it is possible to reach each history in $\sigma_\Theta(\mathcal{U})$ from a length-one history.

The first major result of this section will show that every historical-selection operator preserves SHS representability, i.e., if $\mathcal{U} \subseteq$ SEQ(<U>) is SHS representable and Θ is a computable mapping from SEQ(<U>) to {true, false}, then $\sigma_\Theta(\mathcal{U})$ is SHS representable.

THEOREM 3. 1: Let H be an SHS over <U>, and Θ a computable mapping from SEQ(<U>) to {true, false}. Then σ_Θ(VSEQ(H)) is SHS representable. □

The proof of Theorem 3. 1 uses a spreadsheet function that "performs" the "selection". Whether or not σ_Θ(VSEQ(H)) is history-bounded-SHS representable cannot in general be inferred from the construction of the function.

A historical-selection operator selects a subset of histories using a criterion of interest. Because σ_Θ(VSEQ(H)) is prefix closed, some care must be taken when formulating a selection criterion. Consider the following.

Example 3. 1: Let H be the stock-purchase SHS from Example 1. 1 and $\Theta(\bar{u})$ = true if $\bar{u}$ contains a stock purchase of 5000 or more shares. Ostensibly, σ_Θ(VSEQ(H)) should re-

turn only those histories which contain large stock purchases. But because σ_Θ(VSEQ(H)) is prefix closed, σ_Θ(VSEQ(H)) = VSEQ(H). [Consider an arbitrary sequence $u_1 \ldots u_n$ in VSEQ(H). Let u_{n+1} be the tuple where u_{n+1}(DATE) = u_n(DATE), u_{n+1}(TRANS) = "BUY," u_{n+1}(SHARES) = 5000, u_{n+1}(PSV) = \$5.00, u_{n+1}(VALUE) = $s_{VALUE}(u_1 \ldots u_n, u_{n+1}[<U \mid VALUE>])$, u_{n+1}(PROFIT) = $s_{PROFIT}(u_1 \ldots u_n, u_{n+1}[<U \mid PROFIT>])$, and u_{n+1}(CUMSH) = $s_{CUMSH}(u_1 \ldots u_n, u_{n+1}[<U \mid CUMSH>])$. Then $u_1 \ldots u_{n+1}$ is in σ_Θ(VSEQ(H)). Hence, by prefix closure, so is $u_1 \ldots u_n$.] □

Example 3. 1 shows that σ_Θ(VSEQ(H)) may not be a proper subset of VSEQ(H). The next result shows that it is recursively unsolvable to determine whether or not σ_Θ(VSEQ(H)) = VSEQ(H) for an arbitrary historical selection.

THEOREM 3. 2: It is recursively unsolvable to determine for an arbitrary U, an arbitrary SHS H over <U> and an arbitrary computable mapping Θ defined from SEQ(<U>) to {true, false} whether or not σ_Θ(VSEQ(H)) = VSEQ(H). □

Some of the properties of projection will now be presented. Clearly, projection preserves prefix closure. However, it does not in general preserve SHS representability. Indeed, let H be an SHS over <U> = <I><E>, then neither π_I(VSEQ(H)) nor π_E(VSEQ(H)) is SHS representable. Problems may still arise even if the projection operator is restricted to a subset V of U such that $V \cap I_\infty \neq \emptyset$ and $V \cap E_\infty \neq \emptyset$. It is possible that the loss of information under the projection mapping could preclude the existence of spreadsheet functions for the resulting VSEQ.

PROPOSITION 3. 3: Let H be an SHS over <U> = <I><E>, <U′> = <I′><E′>, $\emptyset \neq I' \subseteq I$, and $\emptyset \neq E' \subseteq E$. Then $\pi_{U'}$(VSEQ(H)) is SHS representable if and only if there exists some r such that for all $n > r$ and all pairs of sequences $u_1 \ldots u_n$ and $w_1 \ldots w_n$ in VSEQ(H),

(*) $\pi_{U'}(u_1 \ldots u_{n-1}) = \pi_{U'}(w_1 \ldots w_{n-1})$ and $\pi_{I'}(u_n) = \pi_{I'}(w_n)$ imply $u_n[E'] = w_n[E']$. □

The next result demonstrates that the VSEQ of every spreadsheet-history scheme is the projected image of a history-bounded spreadsheet-history scheme.

Theorem 3. 4: For each SHS H over <U>, there exists a ρ(H)-history-bounded SHS H′ such that π_U maps VSEQ(H′) one-to-one onto VSEQ(H). □

The proof of Theorem 3. 4 encodes the entire previous history in a single attribute value. However, the complexity of encoding the historical information may be greater than the complexity of the functions in the initial scheme.

A computation-tuple sequence analogue to the relational-database join operator, called cohesion, was defined in [GTa 89]. We now examine the cohesion of spreadsheet histories. First we shall show that the cohesion of two SHS-representable sets is SHS representable. Then we shall address a problem concerning minimum representations.

DEFINITION: Given <U> and <V>, the *cohesion* of $\overline{u}$ in SEQ(<U>) and $\overline{v}$ in SEQ(<V>), denoted $\overline{u} \copyright \overline{v}$, is

1) the computation-tuple sequence $\overline{w}$ in SEQ(<UV>) such that $\pi_U(\overline{w}) = \overline{u}$ and $\pi_V(\overline{w}) = \overline{v}$ if $\pi_A(\overline{u}) = \pi_A(\overline{v})$ for each A in $U \cap V$, and
2) undefined otherwise.

The *cohesion* of $\mathcal{U} \subseteq$ SEQ(<U>) and $\mathcal{V} \subseteq$ SEQ(<V>), denoted $\mathcal{U} \copyright \mathcal{V}$, is the set $\{ \overline{u} \copyright \overline{v} \mid \overline{u}$ in $\mathcal{U}, \overline{v}$ in $\mathcal{V}\}$. □

Since the attributes in U_∞ are ordered by $<_\infty$, <UV> = <VU>. Hence, $\overline{u} \copyright \overline{v} = \overline{v} \copyright \overline{u}$ for each $\overline{u}$ in SEQ(<U>) and $\overline{v}$ in SEQ(<V>).

A question which naturally arises is: Does cohesion preserve SHS representability? In other words, is VSEQ(H_1) © VSEQ(H_2) SHS representable for all SHS H_1 and H_2? The answer is yes. To demonstrate this, we need:

DEFINITION: Let $H_1 = ((<I_1>, <E_1>, \boldsymbol{S}_1), \boldsymbol{I}_1)$ and $H_2 = ((<I_2>, <E_2>, \boldsymbol{S}_2), \boldsymbol{I}_2)$ be SHS over <U> and <V> respectively, and $r = max\{\rho(H_1), \rho(H_2)\}$. For $i = 1, 2$ and each B in E_i, let s_{iB} denote the spreadsheet function for B in $\boldsymbol{S}_i$. The *cohesion* of H_1 and H_2, denoted $H_1 \copyright H_2$, is the SHS $((<I_1I_2>, <E_1E_2>, \boldsymbol{S}), \boldsymbol{I})$, where:

- $\boldsymbol{I}$ = prefix^r(VSEQ(H_1) © VSEQ(H_2)).
- For each B in $(E_1 \cup E_2) - (E_1 \cap E_2)$, s_B is the rank-r spreadsheet function defined by $s_B(\overline{w}, x) = s_{1B}(\pi_U(\overline{w}), x[<U \mid B>])$ if B is in E_1 and $s_B(\overline{w}, x) = s_{2B}(\pi_V(\overline{w}), x[<V \mid B>])$ if B is in E_2.
- For each B in $E_1 \cap E_2$, s_B is the rank-r spreadsheet function defined by

$$s_B(\overline{w}, x) = \begin{cases} s_{1B}(\pi_U(\overline{w}), x[<U \mid B>]) & \text{if } s_{1B}(\pi_U(\overline{w}), x[<U \mid B>]) = \\ & \quad s_{2B}(\pi_V(\overline{w}), x[<V \mid B>]) \\ \text{undefined} & \text{otherwise.} \end{cases}$$ □

The first result concerning the cohesion of SHS shows that the cohesion operator preserves SHS representability.

THEOREM 3.5: Let H_1 and H_2 be SHS. Then

(*) $\quad VSEQ(H_1 \copyright H_2) = VSEQ(H_1) \copyright VSEQ(H_2).$

Furthermore, if H_1 and H_2 are history-bounded-SHS representable, then so is $H_1 \copyright H_2$. □

We now turn our attention to a question about "minimum representations" with respect to cohesion. To motivate this idea, let H_1 and H_2 be SHS over <U> and <V> respectively. Suppose a collection of spreadsheet histories $\mathcal{U} \subseteq VSEQ(H_1)$ is maintained on computer one and $\mathcal{V} \subseteq VSEQ(H_2)$ is maintained on computer two. To calculate $\mathcal{U} \copyright \mathcal{V}$ on computer one, $\mathcal{V}$ must be transmitted from computer two via some communication channel. To reduce the use of the channel, only histories in $\mathcal{V}$ which will participate in the cohesion should be transmitted. It is easily seen that $\pi_V(\mathcal{U} \copyright \mathcal{V})$ is exactly the set which should be sent. However, at site two the contents of $\mathcal{U}$ cannot be known without prior communication. Barring the existence of information about $\mathcal{U}$ at computer two, the best that can be done is to send only those histories in $\mathcal{V}$ which can participate in at least one cohesion with some history in $VSEQ(H_1)$.

Ideally, we would like to find an SHS H_2' such that $VSEQ(H_1) \copyright VSEQ(H_2) = VSEQ(H_1) \copyright VSEQ(H_2')$ and $VSEQ(H_2')$ is a minimum with respect to containment in $VSEQ(H_2)$. (Of course, it would have to be established that such an SHS exists.) The scheme H_2' could then be used as a filter to possibly reduce the number of histories sent to computer one (i.e., only histories in $\mathcal{V} \cap VSEQ(H_2')$ need be sent).

If such an H_2' exists, then a corresponding minimal H_1' might also exist by analogy. If H_1' and H_2' are minimal with respect to each other, then we call the pair a *minimum representation* of H_1 and H_2. More formally:

DEFINITION: Let H_1 be an SHS over <U> and H_2 an SHS over <V>. An ordered pair (H_1', H_2') of SHS is called a *minimum representation* of (H_1, H_2) (with respect to cohesion) if

1) $VSEQ(H_1) \copyright VSEQ(H_2) = VSEQ(H_1') \copyright VSEQ(H_2')$, and
2) $VSEQ(H_1') \subseteq VSEQ(H_1'')$ and $VSEQ(H_2') \subseteq VSEQ(H_2'')$ for all H_1'' over <U> and H_2'' over <V> such that $VSEQ(H_1) \copyright VSEQ(H_2) = VSEQ(H_1'') \copyright VSEQ(H_2'')$. □

The next theorem asserts that every pair of SHS has a minimum representation.

THEOREM 3. 6: Let H_1 and H_2 be SHS over <U> and <V> respectively. Then (H_1, H_2) has a minimum representation (H_1', H_2'). Furthermore, $VSEQ(H_1') = \pi_U(VSEQ(H_1) \copyright VSEQ(H_2))$ and $VSEQ(H_2') = \pi_V(VSEQ(H_1) \copyright VSEQ(H_2))$. □

Suppose both H_1 and H_2 are history-bounded-SHS representable. From Theorem 3. 5, we know that $H_1 \copyright H_2$ is also history-bounded-SHS representable. The question arises: if (H_1', H_2') is a minimum representation of (H_1, H_2), then are H_1' and H_2' also history-bounded-SHS representable? It is straightforward to construct a counterexample.

Turning to intersection, note that $VSEQ(H_1) \cap VSEQ(H_2) = VSEQ(H_1) \copyright VSEQ(H_2)$. Thus, as a corollary to Theorem 3. 5, we have:

PROPOSITION 3. 7: Let H_1 and H_2 be SHS over <U>. Then $VSEQ(H_1) \cap VSEQ(H_2)$ is SHS representable. Furthermore, if both H_1 and H_2 are history bounded, then $VSEQ(H_1) \cap VSEQ(H_2)$ is history-bounded-SHS representable. □

The state of affairs for the union operator is not as good. Let H_1 and H_2 be SHS over some <U>. In general, $VSEQ(H_1) \cup VSEQ(H_2)$ is not SHS representable because the spreadsheet schemes may have disparate functions for their evaluation attributes. However, if the spreadsheet functions in the two SHS are compatible with each other then the union may be SHS representable.

DEFINITION: Let H_1 and H_2 be SHS over <U>. Then H_1 and H_2 are said to be *compatible* if there exists a spreadsheet scheme S over <U> such that $VSEQ(H_1) \subseteq VSEQ(S)$ and $VSEQ(H_2) \subseteq VSEQ(S)$. □

PROPOSITION 3. 8: Let H_1 and H_2 be SHS over <U>. Then $VSEQ(H_1) \cup VSEQ(H_2)$ is SHS representable if and only if H_1 and H_2 are compatible. □

4 Projection Simulation

In [GK 88], the question of when two SHS (or CSS) describe the same set of histories was studied. The definition of "sameness", called "projection simulation," was based on the computation-tuple-sequence analogue to the relational-database projection operator. Under AFOSR support, this study was extended to the subclass of history-bounded SHS. In this section, the formal definition of projection simulation is presented and then examined with respect to history-bounded SHS.

Suppose we wish to implement a database application in which stock-transaction histories are represented as computation-tuple sequences that are valid with respect to the SHS H of Example 1.1. To uniquely identify a history we need to know its initialization and its subsequent inputs. From this and the history scheme, we can derive the value for each evaluation attribute of each tuple in the history. Storing just the initialization and inputs is a space-efficient way to maintain the database. From a computational-efficiency perspective, however, this may be a poor method. For each query based upon the values of evaluation attributes, the database system would have to calculate these values. Furthermore, each time an update is performed (i.e., a new tuple is added to a history) it may be necessary to recalculate the values of the evaluation attributes for the entire history. [An update is not valid unless all of the evaluation-attribute values in the new tuple are defined. In general, these values depend on the previous evaluation-attribute values.]

Suppose it is known that 95% of the queries will be based on the input attributes and the PROFIT evaluation attribute (the transaction profit). To strike a balance between space and computation efficiency, we may want to store only the PROFIT evaluation-attribute values. We could then process 95% of the queries without having to calculate the other evaluation-attribute values. (The other 5% of the queries would still require these calculations.) Ideally, we would like to find an SHS H′ over <V> = <I><PROFIT> such that the projection operator π_V maps VSEQ(H) one-to-one onto VSEQ(H′). We could then maintain the database using the SHS H′ and eliminate the need to calculate the values for the evaluation attributes in E – {PROFIT} during an update.

In a sense, the two SHS H and H′ would define the same set of stock-transaction histories because each sequence in VSEQ(H) would correspond to a unique sequence in VSEQ(H′), one comprised of the same inputs and a nonempty subset of the same evaluation attribute values. The roles played by the attributes in H′ are identical to those played by the same attributes in H. Intuitively speaking, because we have a one-to-one onto mapping, the information lost by not maintaining the attributes in E – {PROFIT} is redundant in that it is not necessary for discriminating between different histories.

We shall call the concept of sameness defined above, *projection simulation* or *p-simulation*. Formally, we have:

DEFINITION: Let H_1 be an SHS over <U> = <I><E> and H_2 an SHS over <V> = <I><E′>, with E′ ⊂ E. H_2 *projection simulates* (*p-simulates*) H_1 if π_V maps VSEQ(H_1) one-to-one onto VSEQ(H_2). □

We now present three separate conditions sufficient to insure that p-simulation preserves history boundedness. Each condition is stated in terms of a special type of data dependency called a *history-bounded dependency*.

DEFINITION: Let <U> be a finite sequence of attributes and r a non-negative integer. A *history-bounded dependency* over <U> is an ordered pair (X, Y), where XY ⊆ U. A set $\mathcal{U} \subseteq$ SEQ(<U>) is said to *rank-r satisfy* (X, Y), denoted $\mathcal{U} \models_r$ (X, Y), if for each B in Y and each pair of sequences $u_1 \ldots u_{r+1}$ and $v_1 \ldots v_{r+1}$ in Interval($\mathcal{U}$), $\pi_X(u_1 \ldots u_r) = \pi_X(v_1 \ldots v_r)$ and[3] $\pi_{<X|B>}(u_{r+1}) = \pi_{<X|B>}(v_{r+1})$ imply $u_{r+1}(B) = v_{r+1}(B)$. □

Note that if $\mathcal{U} \models_r$ (X, Y), then $\mathcal{U} \models_s$ (X, Y) for all s, $s \geq r$.

If $\mathcal{U} \models_r$ (X, Y) for some r, then we sometimes write $\mathcal{U} \models$ (X, Y) when the particular value of r is unimportant.

Let s_B be an r-history-bounded spreadsheet function defined from SEQ(<U>,r) × Dom(<U|B>) to Dom(B). It is readily seen that VSEQ(s_B) $\models_r$ (U, B).

If $\mathcal{V} \subseteq \mathcal{U}$ and $\mathcal{U} \models_r$ (X, Y), then $\mathcal{V} \models_r$ (X, Y). [Indeed, let $u_1 \ldots u_{r+1}$ and $v_1 \ldots v_{r+1}$ be in Interval($\mathcal{V}$). Since $u_1 \ldots u_{r+1}$ and $v_1 \ldots v_{r+1}$ are also in Interval($\mathcal{U}$), for each B in Y $\pi_X(u_1$

3. If B $<_\infty$ C for each C in X, then the value of the expression <X|B> is the empty sequence. In this case, the expression $\pi_{<X|B>}(u)$ is defined to be a special value, called the *empty tuple*, which has the property that $\pi_{<X|B>}(u) = \pi_{<X|B>}(v)$ for all tuples u and v.

... u_r) = $\pi_X(v_1 \ldots v_r)$ and $\pi_{<X|B>}(u_{r+1}) = \pi_{<X|B>}(v_{r+1})$ imply that $u_{r+1}(B) = v_{r+1}(B)$, i.e., $\mathcal{V} \models_r (X, Y)$.]

History-bounded dependencies are similar in spirit to functional dependencies (FDs) [Co 72] in the relational database model. Analogous to the rules of FD inference [DC 73; Arm 74; Ma 83], there are rules for inferring new history-bounded dependencies from a given set of history-bounded dependencies. The following proposition states several such rules.

PROPOSITION 4.1: Let X_1, X_2, Y_1 and Y_2 be non-empty subsets of U and $\mathcal{U} \subseteq$ SEQ(<U>). Then

(a) $\mathcal{U} \models_r (X_1, Y_1)$ implies $\mathcal{U} \models_r (X_1X_2, Y_1)$;

(b) $\mathcal{U} \models_r (X_1, Y_1)$ and $\mathcal{U} \models_s (X_2, Y_2)$ imply $\mathcal{U} \models_{max\{r,s\}} (X_1X_2, Y_1Y_2)$; and

(c) $\mathcal{U} \models_r (X_1, Y_1)$ and $\mathcal{U} \models_s (Y_1X_2, Y_2)$ imply $\mathcal{U} \models_{r+s} (X_1X_2, Y_2)$.

□

THEOREM 4.2: Let H be a history-bounded SHS over <U> = <I><E>, $\emptyset \neq Y \subset E$, V = U − Y, and VSEQ(H) ⊨ (V, Y). Then each SHS over <V> which p-simulates H is history-bounded-SHS representable. □

In the proof of Theorem 4. 2, the condition VSEQ(H) ⊨ (V, Y) guarantees that history-bounded spreadsheet functions for the (E − Y)-attributes can be expressed solely in terms of the V-attributes. In other words, VSEQ(H) ⊨ (V, Y) implies VSEQ(H) ⊨ (V, E − Y). This last condition, VSEQ(H) ⊨ (V, E − Y), is necessary for H to be p-simulated by a history-bounded SHS H′ over <V>.

THEOREM 4.3: Let H = ((<I>, <E>, $\{s_B \mid B \text{ in } E\}$), I) be a history-bounded SHS over <U>, $\emptyset \neq Y \subset E$ and V = U − Y. If VSEQ(H) ⊨ (V, E − Y) and s_B is total for each B in Y, then each SHS over <V> which p-simulates H is history-bounded-SHS representable. □

The final result of this section demonstrates the special nature of zero-history-bounded spreadsheet functions.

THEOREM 4.4: Let H = ((<I>, <E>, S), I) be a history-bounded SHS over <U>, $\emptyset \neq E' \subset E$, and <V> = <I><E′>. If s_B is zero-history bounded for each B in E − E′, then there exists a history-bounded SHS over <V> which p-simulates H. □

Speaking intuitively, Theorem 4. 4 says that the roles played by zero-history-bounded spreadsheet functions in an SHS may be subsumed by the other evaluation attributes without affecting the history-boundedness of the SHS.

Bibliography

Arm 74 Armstrong, W.W. "Dependency structures of data base relationships." Proceedings IFIP Congress. Amsterdam: North Holland, 1974.

Be 86 Berry, T. "How to Structure Spreadsheets." Bus. Software. Oct. 1986: 56–58.

CA Cobb, D., and Anderson, L. 1-2-3 for Business. Indianapolis: Que Corporation.

Co 70 Codd, E. F. "A Relational Model of Data for Large Shared Data Banks." CACM 13:6 (June 1970): 377–387.

Co 72 Codd, E.F. "Further normalization of the data base relational model." Englewood Cliffs, N.J.: Prentice Hall, 1972.

DC 73 Delobel, C. and Casey, R.G. "Decomposition of a database and the theory of boolean switching functions." IBM J. Res. Development 17 (1973) 374–386.

DLL 88 Dishkin, B., Lahey, V., and Lahey, K.: "Appraisers' Utilization of Computer Technology." Appraisal Journal 56 (1988): 179-189.

DSD C-Calc Spreadsheet Reference Manual. Kirkland, Wa: DSD Corporation.

GK 88 Ginsburg, S., and Kurtzman, S. "Spreadsheet Histories, Object-Histories, and Projection Simulation." ICDT '88 2nd International Conference on Database Theory Bruges, Belgium, August/September 1988 Proceedings, Lecture Notes in Computer Science, no. 326. Berlin: Springer-Verlag, 1988.

GTa 89 Ginsburg, S., and Tang, C. "Cohesion of Object Histories." Theoretical Computer Science 63 (1989): 63-90.

Go 87 Gomersall, N.: "Beyond the Spreadsheet". Journal of Accountancy 100 (1987): 169.

Jo 89 Jorgensen, C. Mastering 1-2-3 Release 3. Alameda, Ca: Sybex, 1989.

K 91 Kurtzman, S. "Properties of Spreadsheet Histories." Ph.D. thesis, University of Southern California, in preparation.

Lot 85 1-2-3 Reference Manual Release 2. Cambridge, Ma: Lotus Development Corporation, 1985.

Ma 83 Maier, D. The Theory of Relational Databases. Rockville, Md: Computer Science Press, 1983.

Ms 89 Microsoft® Excel Reference: Complete Spreadsheet with Business Graphics and Database Version 2.2. Redmond, Wa: Microsoft Corporation 1989.

RPL 89 Ronen, B., Palley, M., and Lucas, H., Jr. "Spreadsheet Analysis and Design." Comm. ACM 32 (1989): 84–93

Ul 82 Ullman, J. Principles of Database Systems, Second Edition. Rockville, Md: Computer Science Press, 1982.

WS 86 Wolfe, C., and Smith, L. "Recommending a Microcomputer System to a Small-Business Client." The Ohio CPA Journal, Spring, 1986.