# Querying Shapes of Histories

*Rakesh Agrawal* *Giuseppe Psaila*[*] *Edward L. Wimmers* *Mohamed Zaït*

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

## Abstract

We present a shape definition language, called $\mathcal{SDL}$, for retrieving objects based on shapes contained in the histories associated with these objects. It is a small, yet powerful, language that allows a rich variety of queries about the shapes found in historical time sequences. An interesting feature of $\mathcal{SDL}$ is its ability to perform blurry matching. A "blurry" match is one where the user cares about the overall shape but does not care about specific details. Another important feature of $\mathcal{SDL}$ is its efficient implementability. The $\mathcal{SDL}$ operators are designed to be greedy to reduce non-determinism, which in turn substantially reduces the amount of back-tracking in the implementation. We give transformation rules for rewriting an $\mathcal{SDL}$ expression into a more efficient form as well as an index structure for speeding up the execution of $\mathcal{SDL}$ queries.

## 1 Introduction

Historical time sequences constitute a large portion of data stored in computers. Examples include histories of stock prices, histories of product sales, histories of inventory consumption, etc. Assume a simple data model in which the database consists of a set of objects. Associated with each object is a set of sequences of real values. We call these sequences *histories* and each history has a name. For example, in a stock database, associated with each stock may be histories of opening price, closing price, the high for the day, the low for the day, and the trading volume.

The ability to select objects based on the occurrence of some shape in their histories is a requirement that arises naturally in many applications. For example, we may want to retrieve stocks whose closing price history contains a head and shoulder pattern [4]. We should be able to specify shapes roughly. For example, we may choose to call a trend uptrend even if there were some down transitions as long as they were limited to a specified number.

To this end, we propose a shape definition language, called $\mathcal{SDL}$. It is a small, yet powerful, language that allows a rich variety of queries about the shapes found in histories. The most interesting feature of $\mathcal{SDL}$ is its capability for blurry matching. A "blurry" match is one where the user cares about the overall shape but does not care about specific details. For example, the user may be interested in a shape that is five time periods long and contains at least three ups but no more than one down. $\mathcal{SDL}$ has been designed to make it easy and natural to express such queries. Another important feature of $\mathcal{SDL}$ is that it has been designed to be efficiently implementable. Most of the $\mathcal{SDL}$ operators are greedy and therefore there is very little non-determinism (in the sense of multiple match possibilities) inherent in an $\mathcal{SDL}$ shape, which in turn substantially reduces the amount of back-tracking in the implementation. In addition, $\mathcal{SDL}$ provides the potential for rewriting a shape expression into a more efficient form as well as the potential for indexes for speeding up the implementation.

$\mathcal{SDL}$ benefits from a rich heritage of languages based on regular expressions, but this earlier work

[*]Current Address: Politecnico di Milano, Italy.

*Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.*

**Proceedings of the 21st VLDB Conference**
**Zürich, Switzerland 1995**

| Symbol | Description | *lb* | *ub* | *iv* | *fv* |
|---|---|---|---|---|---|
| `up` | slightly increasing transition | .05 | .19 | `anyvalue` | `anyvalue` |
| `Up` | highly increasing transition | .20 | 1.0 | `anyvalue` | `anyvalue` |
| `down` | slightly decreasing transition | -.19 | -.05 | `anyvalue` | `anyvalue` |
| `Down` | highly decreasing transition | -1.0 | -.19 | `anyvalue` | `anyvalue` |
| `appears` | transition from a zero value to a non-zero value | 0 | 1.0 | `zero` | `nonzero` |
| `disappears` | transition from a non-zero value to a zero value | -1.0 | 0 | `nonzero` | `zero` |
| `stable` | the final value nearly equal to the initial value | -.04 | .04 | `anyvalue` | `anyvalue` |
| `zero` | both the initial and final values are zero | 0 | 0 | `zero` | `zero` |

Table 1: An Illustrative Alphabet $\mathcal{A}$

has a different design focus that influences which expressions are easy to write, understand, optimize, and evaluate. For example, while the blurry matching of $\mathcal{SDL}$ is reminiscent of approximate matching for strings (e.g., [9]) or for patterns in time series [2], $\mathcal{SDL}$ allows the user to impose arbitrary conditions on the blurry match but requires that the user specify those conditions completely. The event specification languages in active databases [3] [5] [6] concentrate on detecting the endpoints of events rather than concentrating on intervals as $\mathcal{SDL}$ does. The $\mathcal{SEQ}$ work of [8] focused on building a framework for describing constructs from various existing sequence models.

**Organization of the Paper** The rest of the paper is organized as follows. In Section 2, we introduce $\mathcal{SDL}$ informally through examples; the formal semantics is given in Appendix A. In Section 3, we discuss the design rationale of $\mathcal{SDL}$. We discuss its expressive power, its capability for blurry matching, its ease of use, and its efficient implementability. In Section 4, we give transformation rules for rewriting an $\mathcal{SDL}$ expression into an equivalent but a more efficient form. In Section 5, we describe an index structure and show how it can be used to speed up the evaluation of $\mathcal{SDL}$ queries. We conclude in Section 6 with a summary. For an expanded version of this paper, see [1].

## 2 Shape Definition Language

We will introduce our shape definition language, $\mathcal{SDL}$, informally through examples. The formal semantics is given in Appendix A. Every object in the database has associated with it several named histories. Each history is a sequence of real values.

The behavior of a history can be described by considering the values assumed by the history at the beginning and the end of a unit time period; that is, by considering transitions from an instant to the following one. It is immediate then that a history generates a *transition sequence* based on an alphabet whose symbols describe classes of transitions.

### 2.1 Alphabet

The syntax for specifying alphabet is :

(`alphabet` (*symbol lb ub iv fv*))

Here *symbol* is a symbol of the alphabet being defined and the rest four descriptors provide the definition for the symbol. The first two, *lb* and *ub*, are the lower and upper bounds respectively of the allowed variation from the initial value to the final value of the transition. The latter two, *iv* and *fv*, can be one of `zero`, `nonzero` and `anyvalue`, and specify constraints on the initial and final value respectively of the transition.

Table 1 gives an illustrative alphabet $\mathcal{A}$. Consider the time sequence $\mathcal{H}$ in Figure 1.

Given alphabet $\mathcal{A}$, a transition sequence corresponding to $\mathcal{H}$ will be:

```
(zero appears up up up down stable Down
 down disappears)
```

Depending on the alphabet, there can be more than one transition sequence corresponding to a time sequence. For example, another transition sequence corresponding to $\mathcal{H}$ is:

```
(zero stable up up up down stable Down down
 stable)
```

This ambiguity does not cause inconsistency at query time because the user specifies the particular shape to be matched. For example, if the user

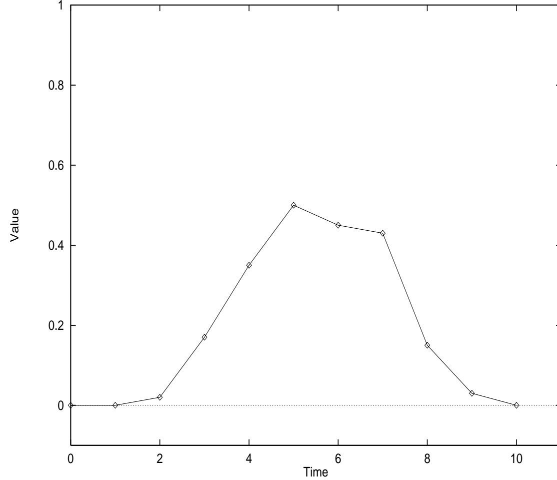


Figure 1: Time Sequence $\mathcal{H}$ = (0 0 .02 .17 .35 .50 .45 .43 .15 .03 0)

had asked for `stable`, we will resolve the ambiguity between `stable` and `zero` in the favor of `stable`.

We will use the alphabet $\mathcal{A}$ and the time sequence $\mathcal{H}$ throughout the paper to give concrete examples. We will use the notation $\mathcal{H}[i,j]$ to represent the subsequence of $\mathcal{H}$ consisting of elements from position $i$ to the position $j$ inclusive, 0 being the first position. $\mathcal{H}[i,i]$ will represent the null sequence since an elementary shape (see Section 2.2.1) requires at least one transition.

### 2.2 Shape Descriptors

Using the alphabet of the language, we can define classes of shapes that can be matched in histories or parts of them. The application of a shape descriptor $P$ to a time sequence $S$ produces a set of all the subsequences in $S$ that match the shape $P$. If no subsequence in $S$ matches $P$, then the result is an empty set. Depending on the descriptor, a null sequence can match a shape. For the convenience of the user, however, the null sequences are not reported to the user.

The syntax for defining a shape is:

(`shape` *name*(*parameters*) *descriptor*)

A shape definition is identified by means of a *name* for the shape, which is followed by a possibly empty list of *parameters* (see Section 2.4) and then a *descriptor* for the shape. For example, here is a definition of a `spike`:

```
(shape spike() (concat Up up down Down))
```

This definition has no parameters. The meaning of the descriptor will become clear momentarily.

#### 2.2.1 Elementary Shapes

The simplest shape descriptor is an *elementary shape*. All the symbols of the alphabet correspond to elementary shapes. When an elementary shape is applied to a time sequence $S$, the resulting set contains all the subsequences of $S$ that contain only the specified elementary shape.

For example, the shape descriptor `(stable)` applied to the time sequence $\mathcal{H}$ given in Figure 1 yields the set $\{\mathcal{H}[0,1], \mathcal{H}[1,2], \mathcal{H}[9,10]\}$, where $\mathcal{H}[0,1]$ = (0 0), $\mathcal{H}[1,2]$ = (0 .02) and $\mathcal{H}[9,10]$ = (.03 0). The descriptor `(zero)` yields the set $\{\mathcal{H}[0,1]\}$. Note that the subsequence $\mathcal{H}[0,1]$ is contained in the result set of both the descriptors because the transition corresponding to this subsequence satisfies the definitions of both `stable` and `zero`. Finally, the shape descriptor `(Up)` results in an empty set because $\mathcal{H}$ contains no `Up` transition.

### 2.3 Derived Shapes

Starting with the elementary shapes, complex shapes can be derived by recursively combining elementary and previously defined shapes. We describe next the set of operators available for this purpose.

**Multiple Choice Operator `any`.** The `any` operator allows a shape to have multiple values. The syntax is

(`any` $P_1$ $P_2$ ... $P_n$)

where $P_i$ is a shape descriptor. When a shape obtained by means of the `any` operator is applied to a time sequence $S$, the resulting set contains all the subsequences of $S$ that match at least one of the $P_i$ shapes.

For example, the shape `(any zero appears)` applied to the time sequence $\mathcal{H}$ yields the set $\{\mathcal{H}[0,1], \mathcal{H}[1,2]\}$, where $\mathcal{H}[0,1]$ = (0 0) which is a `zero` transition and $\mathcal{H}[1,2]$ = (0 .02) which is an `appears` transition.

**Concatenation Operator `concat`.** Shapes can be concatenated by using the operator `concat`:

(`concat` $P_1$ $P_2$ ... $P_n$)

When a shape obtained by using the `concat` operator is applied to a time sequence $S$, first the

shape $P_1$ is matched. If a matching subsequence $s$ is found, $P_2$ is matched in the subsequence of $S$ immediately following the last element of $s$ and the match is accepted if it is strictly contiguous to $s$, etc. For example, the shape descriptor

```
(concat up up up (any stable down)
        (any stable down) (any down Down))
```

specifies that we are interested in detecting if an upward trend (indicated by three consecutive `ups`) has reversed (indicated by two `stables` or `downs`, followed by a `down` or `Down`). When applied to the time sequence $\mathcal{H}$, it yields the set $\{\mathcal{H}[2,8]\}$, where $\mathcal{H}[2,8] = (.02\ .17\ .35\ .50\ .45\ .43\ .15)$. The transition sequence corresponding to this subsequence is `(up up up down stable Down)`.

**Multiple Occurrence Operators `exact`, `atleast`, `atmost`.** Shapes composed of multiple contiguous occurrences of the same shape can be defined using three other operators, `exact`, `atleast` and `atmost`:

(`exact` $n$ $P$)
(`atleast` $n$ $P$)
(`atmost` $n$ $P$)

When a shape obtained using `exact/atleast/atmost` is applied to a time sequence $S$, it matches all subsequences of $S$ that contain exactly/at least/at most $n$ contiguous occurrences of the shape $P$. In addition, the resulting subsequences are such they are neither preceded nor followed by a subsequence that matches $P$. For example,

| | |
|---|---|
| `(exact 2 up)` | yields $\emptyset$. |
| `(atleast 2 up)` | yields $\{\mathcal{H}[2,5]\}$, where $\mathcal{H}[2,5] = (.02\ .17\ .35\ .50)$. |
| `(atmost 2 up)` | yields $\{[k,k] \mid 0 \le k \le 1 \vee 6 \le k \le 10\}$. |

The first shape results in an empty set because there is no subsequence in $\mathcal{H}$ which is exactly two transitions long, consisting entirely of `up` transitions, and neither preceded nor followed by an `up` transition. The second shape matched the subsequence consisting of three contiguous `up` transitions.

The result for the third shape merits further discussion. The shape `(atmost 2 up)` matches the null sequence at those positions of $\mathcal{H}$ that do not participate in an `up` transition. The other null sequences are not in the answer since they participate in a sequence of 3 consecutive `ups`. Since the final answer in this case is a set of null sequences and we do not report null sequences, the user will see $\emptyset$ as the answer. Allowing a null sequence to match `atmost` $n$ $P$ has the virtue that we can naturally specify

```
(concat (atleast 2 up) (atmost 1 Down))
```

and match it to $\mathcal{H}[2,5]$ corresponding to the transition sequence `up up up`.

**Bounded Occurrences Operator `in`.** The `in` operator is the most interesting $\mathcal{SDL}$ operator. It permits blurry matching by allowing users to state an overall shape without giving all the specific details. The syntax is

(`in` *length shape-occurrences*)

Here *length* specifies the length of the shape in number of transitions. The *shape-occurrences* has two forms.

In the first form, the *shape-occurrences* can be one of

(`precisely` $n$ $P$)
(`noless` $n$ $Q$)
(`nomore` $n$ $R$)

or a composition of them using the logical operators `or` and `and`.

When a shape defined using this form is applied to a time sequence $S$, the resulting set contains all subsequences of $S$ that are *length* long in terms of number of time periods (transitions) and contain precisely (no less than/ no more than) $n$ occurrences of the shape $P$ ($Q/R$). The $n$ occurrences of $P$ ($Q/R$) need not be contiguous in the matched subsequence; there may be arbitrary gap between any two of them. They may also overlap. For example, the shape descriptor

```
(in 5 (and (noless 2 (any up Up))
           (nomore 1 (any down Down))))
```

specifies that we are interested in subsequences five intervals long that have at least two ups (either `up` or `Up`) and at most one down (either `down` or `Down`). When applied to the time sequence $\mathcal{H}$, it yields the set $\{\mathcal{H}[2,7]\}$, where $\mathcal{H}[2,7] = (.02\ .17\ .35\ .50\ .45\ .43)$. The transition sequence corresponding to this subsequence is `(up up up down stable)`. Note that the subsequence $\mathcal{H}[3,8] = (.17\ .35\ .50$

.45 .43 .15) ≡ `(up up down stable Down)` is not in the answer because it has two downs. As another example, consider the shape

```
(in 7 (precisely 0 Down))
```

We are looking for sequences seven time periods long that do not have any `Down` transitions. $\mathcal{H}[0,7]$ is the only subsequence of $\mathcal{H}$ that satisfies this constraint.

The operators `precisely`, `noless`, `nomore` should not be confused with the multiple occurrence operators `exact`, `atleast`, and `atmost`. The latter are "first class" operators that can be used to introduce shapes to be matched, whereas the former can only appear within the `in` operator and constrain the sub-shapes. More importantly, `precisely`, `noless`, and `nomore` allow overlaps and gaps, whereas `exact`, `atleast`, and `atmost` do not.

The second form for the *shape-occurrences* is:

$$(\texttt{inorder}\ P_1\ P_2 \ldots P_n)$$

where $P_i$ is a shape descriptor. When a shape obtained using this form is applied to a time sequence, each of the resulting subsequences is *length* long and contains the shapes $P_1$ through $P_n$ in that order. $P_i$ and $P_{i+1}$ may not overlap, but they may have arbitrary gap. For example, the shape descriptor

```
(in 7 (inorder (atleast 2 (any up Up))
        (in 4 (noless 3 (any down Down)))))
```

specifies that we are interested in subsequences seven time periods long. The matching subsequence must contain a subsequence that has atleast two ups and that must be followed by another subsequence four intervals long that contains at least three downs. When applied to the time sequence $\mathcal{H}$, it yields the set $\{\mathcal{H}[2,9]\}$, where $\mathcal{H}[2,9]$ = (.02 .17 .35 .50 .45 .43 .15 .03) ≡ `(up up up down stable Down down)`.

### 2.4 Parameterized Shapes

Shape definitions can be parameterized by specifying the names of the parameters in the parameter list following the shape name and using them in the definition of the shape in place of concrete values. Here is an example of a parameterized spike:

```
(shape spike(upcnt dncnt)
       (concat (exact upcnt (any up Up))
            (exact dncnt (any down Down))))
```

When a parameterized shape $P$ is used in the definition of another shape $Q$, the parameters of $P$ must be bound. They can be bound to concrete values or to the parameters of $Q$. Here is an example:

```
(shape doublepeak(width ht1 ht2)
       (in width (inorder spike(ht1 ht1)
                          spike(ht2 ht2))))
```

## 3 Design of $\mathcal{SDL}$

$\mathcal{SDL}$ provides the following key advantages:

- a natural and powerful language for expressing shape queries
- capability for blurry matching
- reduction of output clutter
- an efficient implementation

### 3.1 Expressive Power of $\mathcal{SDL}$

Using $\mathcal{SDL}$, one can express a wide variety of queries about the shapes found in a history. Given a sequence and a shape, one type of query (called *continuous matching* in [8]) finds all the subsequences that match the shape; the other type of query (referred to as "regular matching" in this paper) produces a boolean indicating whether the entire sequence matches the shape.

Since $\mathcal{SDL}$ includes the operators `concat`, `any`, and `atleast`, $\mathcal{SDL}$ is equivalent in expressive power to regular expressions for regular matching. This equivalence is proven in [1]. Because $\mathcal{SDL}$ is designed to provide ease of expression together with an efficient implementation, it has several features to enhance its effectiveness. The `atleast` operator, which is a variant of the $*$ operator of regular expressions, provides both efficiency gains and expressiveness enhancements for continuous matching. The $*$ operator, once it has found the required number of matches, is allowed (nondeterministically) either to exit or to continue matching; whereas `atleast` is a greedy operator that does not exit until it has found as many matches as it can. In the regular matching case, the greedy nature of `atleast` does not cause a loss of expressive power since one can always write the shape so that subsequent shapes are not affected by the greedy nature of `atleast`. Details of this construction are given in [1].

In the case of continuous matching, the greedy semantics of `atleast` allow $\mathcal{SDL}$ to take advantage of contextual information to eliminate useless clutter. For example, given the shape (`atleast` 5 `up`), $\mathcal{SDL}$ will find all the maximal subsequences that have at least five consecutive `up`s. In other words, $\mathcal{SDL}$ does not report the non-maximal subsequences thereby eliminating useless clutter. Regular expressions would not be able to eliminate the clutter since they are unable to "look-ahead" to provide contextual information. If there happen to be seven consecutive `up`s in the history, $\mathcal{SDL}$ will report this single subsequence of length 7 whereas the regular expression would report six different (largely overlapping) subsequences; there would be three subsequences of length 5, two subsequences of length 6, as well as the entire subsequence of length 7. If, in the future, finding all such subsequences becomes important, a non-greedy version of `atleast` could be added easily to $\mathcal{SDL}$.

### 3.2 Ease of Expression in $\mathcal{SDL}$

$\mathcal{SDL}$ is designed to make it easy and natural to express shape queries. For example, the `atleast` operator provides a compact representation of repetitions that seems natural even to someone not familiar with regular expression notation. $\mathcal{SDL}$ provides a (non-recursive) macro facility (with parameters) that enhances readability by allowing commonly occurring shapes to be abstracted.

One of the most exciting features of $\mathcal{SDL}$ is the inclusion of the `in` operator that permits "blurry" matching in which the user cares about the overall shape but does not care about specific details. For example, to indicate a uptrend with a subsequence specified by the `in` operator, the user might specify (`nomore` 2 `down`) thereby limiting the number of `down`s that can occur in the subsequence. While the `in` operator can be simulated using regular expressions, it is not easy to do so. The details of the construction can be found in [1] and involve keeping track of how many times diverse finite automatons have entered accepting states. The `in` operator presents a much more natural method for expressing the desired shape.

It is instructive to give an example. Assume that $a_1, \ldots, a_n$ are "disjoint" elementary shapes (where two elementary shapes are disjoint if they never match the same transition sequence). Consider the problem of finding a "permutation" expression that matches exactly those sequences of length $n$ that have precisely one occurrence of each $a_i$. The straightforward approach of listing all such possible strings grows factorially. It is well-known that the permutation expression can be compacted a bit to exponential size but no further compaction is possible in regular expression notation. (See [1] for more details and for proofs.) Since at least exponential size is required, expressing permutations in regular expression notation is tedious, error-prone, and not particularly readable.

Parameterized shapes (macros) can dramatically reduce the size of a permutation expression. One can define (inductively) the parameterized shapes $P_i$ to describe all permutations of $i$ elements as follows:

```
(shape P_1(x_1)(x_1))
(shape P_2(x_1, x_2) (any (concat x_1 P_1(x_2))
                          (concat x_2 P_1(x_1))))
(shape P_3(x_1, x_2, x_3) (any (concat x_1 P_2(x_2, x_3))
                               (concat x_2 P_2(x_1, x_3))
                               (concat x_3 P_2(x_1, x_2))
(shape P_i(x_1, ..., x_i)
       (any (concat x_1 P_{i-1}(x_2, ..., x_i))
         ...(concat x_i P_{i-1}(x_1, ..., x_{i-1}))))
```

Since each $P_i$ has size $O(i^2)$, a permutation expression for $n$ elements has size $O(n^3)$.

Blurring matching provides an even more effective permutation expression. For example, (`in` $n$ (`and` (`precisely` 1 $a_1$)...(`precisely` 1 $a_n$)) does the trick in only linear size. It is instructive to examine the features of blurry matching that permit such a compact permutation expression. Blurry matching permits the use of conjunctive as well as disjunctive expressions. It is well known that adding "and" to regular expressions does not increase the expressive power of regular expressions but does permit more compact expressions (see Chapter 3 exercises in [7]). A permutation expression is such an example. The regular expression $(a_1|\ldots|a_n)$ can be used to describe all the characters. By concatenating $n$ copies, it is possible to express in $O(n^2)$ size all sequences of length exactly $n$. It is also easy to see that the regular expression $(a_1|\ldots|a_{i-1}|a_{i+1}|\ldots|a_n)^*\ a_i\ (a_1|\ldots|a_{i-1}|a_{i+1}|\ldots|a_n)^*$ expresses all sequences that have exactly one $a_i$. By conjuncting these expressions together, we obtain a regular expression with conjunctions that expresses permutations and has size $O(n^2)$. As already noted, a (pure) regular expression that expresses permutations must have exponential size.

The compactness of permutation expressions in blurry shape notation is primarily due to the fact that blurry shapes permit conjunctions. Blurry shapes also enhance readability by allowing overlap directly whereas regular expressions (even with conjunctions) can handle overlap only indirectly by coding up the overlap in a different regular expression. Even though the permutation example is somewhat contrived to permit the easy analysis of the complexity and expressive of $\mathcal{SDL}$ versus regular expressions, it is representative of a large class of blurry queries that search for shapes which may occur in any order.

### 3.3 Efficient Implementability for $\mathcal{SDL}$

Since the semantics of $\mathcal{SDL}$ specifies that operators such as `atleast` be greedy, `any` is the only operator that introduces any "non-determinism". (In this context, non-determinism means that there is some starting point that has at least two different subsequences that match starting from that particular starting point.) This implies that the amount of back-tracking an $\mathcal{SDL}$ implementation needs to do is substantially reduced. For example, in the shape (`concat` (`atleast` 4 $P$)(`atleast` 3 $Q$)) under the normal regular expression semantics, after 4 $P$'s were found, the evaluator (i.e. automaton) would have to keep searching for $P$ as well as begin searching for $Q$. In the $\mathcal{SDL}$ semantics, the search for $Q$ would not begin until all the $P$'s had been found.

In addition, $\mathcal{SDL}$ provides the potential for rewriting a shape expression into a more efficient form (Section 4) as well as the potential for indexes (Section 5).

## 4 Shape Rewriting

We now present a set of transformation rules to rewrite a shape expression into an *equivalent* but a *more efficient* expression. $\mathcal{SDL}$ shape operators can be classified into the following groups:

- `concat`, `exact`, `atleast`, `atmost`, and `inorder`: Shape arguments must appear in the specified order without overlap.
- `precisely`, `noless`, and `nomore`: Shape arguments must appear in the specified order but can overlap.
- `and`, `or` and `any`: Shape arguments may appear in any order.

An operator can be rewritten using only operators belonging to the same group.

### 4.1 Idempotence, Commutativity, and Associativity

An operator has the idempotence property if the duplicates of a shape can be removed. It has the commutativity property if shapes can be permuted. The associativity property is useful for unnesting similar operators, after which redundant shapes can be removed using idempotence and commutativity. The `any`, `or`, and `and` operators are idempotent, commutative, and associative. The `concat` and `inorder` operators are associative (but not idempotent and commutative).

Here is an example of the application of these properties:

(`any` $P_1$ (`any` $P_2$ $P_1$))
$\Leftrightarrow$ (`any` $P_1$ $P_2$ $P_1$) – associativity
$\Leftrightarrow$ (`any` $P_1$ $P_1$ $P_2$) – commutativity
$\Leftrightarrow$ (`any` $P_1$ $P_2$) – idempotence

### 4.2 Distributivity

The `concat` and `and` operators distribute over `any` and `or` operators:

(`concat` $P_1$ (`any` $P_2$ $P_3$))
$\Leftrightarrow$ (`any` (`concat` $P_1$ $P_2$) (`concat` $P_1$ $P_3$))
(`and` $P_1$ (`or` $P_2$ $P_3$))
$\Leftrightarrow$ (`or` (`and` $P_1$ $P_2$) (`and` $P_1$ $P_3$))

Deciding which form is less costly to match is similar to the problem of distributing the join over the union in relational query optimization, since `concat` and `and` result in joins and `any` and `or` result in a union of resulting sets (see Section 5).

### 4.3 Folding identical shapes in `concat`

Identical shapes inside the `concat` operator are folded using the `exact` operator. For example:

(`concat` $P_1$ $P_2$ $P_2$ ... $P_2$ $P_3$)
$\Leftrightarrow$ (`concat` $P_1$ (`exact` n $P_2$) $P_3$)

where $n$ is the number of occurrences of $P_2$ in the original shape definition, and $P_1$ and $P_3$ do not have a common suffix/prefix with $P_2$. This transformation allows the index structure presented in Section 5 to be used to evaluate the subshape (`exact` $n$ $P_2$).

### 4.4 Multiple Occurrences Operators

The shape expressions involving a multiple Occurrences Operator (MOO) can often be reduced to simpler expressions. The transformation rules fall into three categories, depending on how the MOO has been used: composed with another MOO, inside `concat`, or inside `any`.

**Composition.** When a MOO, $\texttt{M}_1$, is composed with another MOO, $\texttt{M}_2$, the result depends on what $\texttt{M}_1$ is:

$$(\{\texttt{exact}|\texttt{atleast}\}\ n\ (\texttt{M}_2\ m\ P)) \Leftrightarrow (\texttt{M}_2\ m\ P) \text{ if } n = 1, \quad \emptyset \text{ if } n > 1.$$

$$(\texttt{atmost}\ n\ (\texttt{M}_2\ m\ P)) \Leftrightarrow (\texttt{any}\ (\texttt{exact}\ 0\ (\texttt{M}_2\ m\ P))\ (\texttt{M}_2\ m\ P)) \text{ if } n \geq 1.$$

In the rule for the `atmost` operator, the shape arguments to `any` in the right-hand side of the rule correspond to 0 and 1 occurrences of the `atmost` argument in the match.

**Inside `concat`.** When the `concat` operator is applied to two MOOs, $\texttt{M}_1$ and $\texttt{M}_2$, on the same shape, the result is $\emptyset$. The only exception is when $\texttt{M}_2$ matches the null sequence, in which case the result is the same as yielded by $\texttt{M}_1$. $\texttt{M}_2$ can match the null sequence either because it is `atmost` or because the specified number of occurrences is 0.

$$(\texttt{concat}\ (\texttt{M}_1\ n\ P)\ (\texttt{M}_2\ m\ P)) \Leftrightarrow (\texttt{M}_1\ n\ P) \text{ if } (\texttt{M}_2{=}\texttt{atmost} \text{ or } m{=}0), \text{ and } \emptyset \text{ otherwise.}$$

**Inside `any`.** The operators `atmost` and `atleast` can match a range of number of occurrences of the specified shape, whereas `exact` matches only the specified number of occurrence. Therefore, their behavior differs inside `any`. Two `atmost` (or `atleast`) over the same shape are equivalent to one `atmost` (or `atleast`) with the number of occurrences equal to the maximum (or minimum) of the original ones.

$$(\texttt{any}\ (\texttt{atleast}\ n\ P)\ (\texttt{atleast}\ m\ P)) \Leftrightarrow (\texttt{atleast}\ min(n,m)\ P)$$

$$(\texttt{any}\ (\texttt{atmost}\ n\ P)\ (\texttt{atmost}\ m\ P)) \Leftrightarrow (\texttt{atmost}\ max(n,m)\ P)$$

If two `exact` over the same shape specify the same number of occurrences, they can be reduced to one `exact`; otherwise, the shape expression remains unchanged.

When different MOOs are used inside `any`, we have the following rules (the order in which different MOOs are written inside `any` is not important because `any` is commutative):

$$(\texttt{any}\ (\texttt{exact}\ n\ P)\ (\texttt{atleast}\ m\ P)) \Leftrightarrow (\texttt{atleast}\ m\ P) \text{ if } m \leq n, \quad (\texttt{atleast}\ n\ P) \text{ if } n = m \Leftrightarrow 1$$

$$(\texttt{any}\ (\texttt{exact}\ n\ P)\ (\texttt{atmost}\ m\ P)) \Leftrightarrow (\texttt{atmost}\ m\ P) \text{ if } m \geq n, \quad (\texttt{atmost}\ n\ P) \text{ if } m = n \Leftrightarrow 1$$

$$(\texttt{any}\ (\texttt{atmost}\ n\ P)\ (\texttt{atleast}\ m\ P)) \Leftrightarrow (\texttt{atleast}\ 0\ P) \text{ if } m \leq n+1$$

The above rules are the consequence of the following rewritings of `atleast` and `atmost`:

$$(\texttt{atleast}\ n\ P) \Leftrightarrow (\texttt{any}\ (\texttt{exact}\ n\ P)\ (\texttt{exact}\ (n{+}1)\ P) \ldots (\texttt{exact}\ (p{\Leftrightarrow}1)\ P)\ (\texttt{exact}\ p\ P))$$

$$(\texttt{atmost}\ n\ P) \Leftrightarrow (\texttt{any}\ (\texttt{exact}\ 0\ P)\ (\texttt{exact}\ 1\ P) \ldots (\texttt{exact}\ (n \Leftrightarrow 1)\ P)\ (\texttt{exact}\ n\ P))$$

where $p$ is the length of the interval over which the matching is being performed.

### 4.5 The "in" operator

When composed with each other, the operators `precisely`, `noless` and `nomore` have the same properties as the MOOs. When used inside `and` or `or` operators, they have the same properties as MOOs when used inside `concat` or `any` operators, respectively.

When the length specified for the `in` operator is less than the guaranteed minimum length of the shape or the interval length where the match is to be performed, then the result is empty. The guaranteed minimum length can often be computed when the shape expression involves `noless` or `precisely`.

It might be tempting, but `inorder` cannot be rewritten using the other `in` operators because it is the only one in the `in` family that allows gaps but not overlap. For example, the following transformations are not valid:

$$(\texttt{or}\ (\texttt{inorder}\ P_1\ P_2)\ (\texttt{inorder}\ P_2\ P_1)) \not\Leftrightarrow (\texttt{and}\ P_1\ P_2)$$

$$(\texttt{inorder}\ P\ \ldots\ P) \not\Leftrightarrow (\texttt{precisely}\ n\ P)$$

## 5 Indexing

A straightforward method to evaluate a shape query will be to scan the entire database and match the

specified shape against each sequence. We propose a storage structure and show how it is used for speeding up the implementation of $\mathcal{SDL}$.

### 5.1 The Storage Structure

The proposed hierarchical storage structure, which also acts as an index structure, consisting of four layers. The top layer is an array indexed by a *symbol name* from the alphabet. Its size is $ns$ where $ns$ is the number of symbols in the alphabet. Its elements point to one instance of the second layer. An instance of the second layer is an array indexed by the *start period* of the first occurrence of the symbol in the sequence, whose elements point to one instance of the third layer. The size of an array of this layer is $np$ where $np$ is the maximum number of time periods in some time sequence. One instance of the third layer is an array indexed by the *maximum number of occurrences* of the associated symbol. Each element of this array points to a sorted list of object_ids. Consider an array at this layer, being pointed to from the $k$th element of a second-layer array. This array will have $np \Leftrightarrow k$ elements, starting from the $k$th position, because a symbol can occur at most $np \Leftrightarrow k$ times. Thus, the number of elements in a third-layer array depends on its parent in the second-layer. We use NULL, as a special value, to mark elements corresponding to empty combinations, e.g., when a given symbol does not start at a specific position in any of the sequences in the database. Having created this structure, we no longer need the original data.

Figure 2 illustrates this structure. The specific entries in this structure are for the sequence $\mathcal{H}$ given in Figure 1.

The size of the first three layers of the structure is independent of the number of sequences in the database, whereas the fourth layer depends on the number of sequences. In the worst case, the first three layers will have $ns(1 + np + np \times (np + 1)/2)$ entries, which can be approximated to $ns \times np^2/2$. This case arises when all the elements of all the arrays are non-NULL. In the worst case, there can be a total of $np$ entries in the fourth layer for a sequence whose transition sequence does not contain any identical symbol in two contiguous positions. In the best case, there will be one entry. If sequences have on average $k$ identical contiguous symbols, the total number of entries in the index will roughly equal $np \times (ns \times np/2 + nseq/k)$. The original data sequences can be stored as sequences of tuples $(s, k')$, where $k'$ is the number of contiguous occurrences of the symbol $s$, requiring $2 \times np \times nseq/k$ entries. We generally expect $np$ to be much smaller than $nseq$. Thus, if we were to store sequences using the index storage structure, we can save storage as long as $k < (2 \times nseq)/(ns \times np)$. For $ns = 10, np = 50, nseq = 1000$, $k$ up to 10 can save storage. In addition, the index can speed up query processing.

### 5.2 The Mapping Problem

There may be more than one transition sequence corresponding to a time sequence. For example, the time sequence `(0 0 0)` can be mapped either to `(zero zero)` or to `(zero stable)`. One way to deal with this problem is to store both mappings in the index. However, this may lead to an exponential explosion in the number of mappings. Instead, we store only one form in the index as explained below.

Assume the existence of a set $\mathcal{P}$ of *primitive elementary shapes* that are disjoint (i.e. every transition is in at most one of the primitive shapes). Thus, there is no ambiguity with regard to the members of the set $\mathcal{P}$. Further assume that every elementary shape is the "union" of some subset of $\mathcal{P}$ (i.e. every transition in the given elementary shape is in exactly one of the primitive elementary shape in the subset of $\mathcal{P}$ corresponding to the given elementary shape). In this case, the transformation rule `E` $\Leftrightarrow$ (`any` $\mathtt{P_1}$ ... $\mathtt{P_n}$) eliminates the elementary shape `E` in favor of the corresponding primitive elementary shapes $\mathtt{P_1}$ ... $\mathtt{P_n}$ for which there is no ambiguity.

Since there might not already be a set of primitive elementary shapes, it might be necessary to add new primitive elementary shapes. In general, this requires an exponential number of new primitive elementary shapes since there would need to be a new primitive elementary shape for every possible non-empty subset of the original elementary shapes. Fortunately, there is a natural sufficient condition that requires only a linear blowup in the number of new primitive elementary shapes. If every primitive shape can be associated with an interval of real numbers, then there is only linear blowup. To see this, imagine $n$ elementary shapes. These give rise to $2n$ endpoints. These endpoints define at most $2n+1$ disjoint consecutive intervals. (There may be fewer than $2n+1$ intervals since some of the endpoints might coincide.) Add a new primitive symbol for each such interval, giving rise

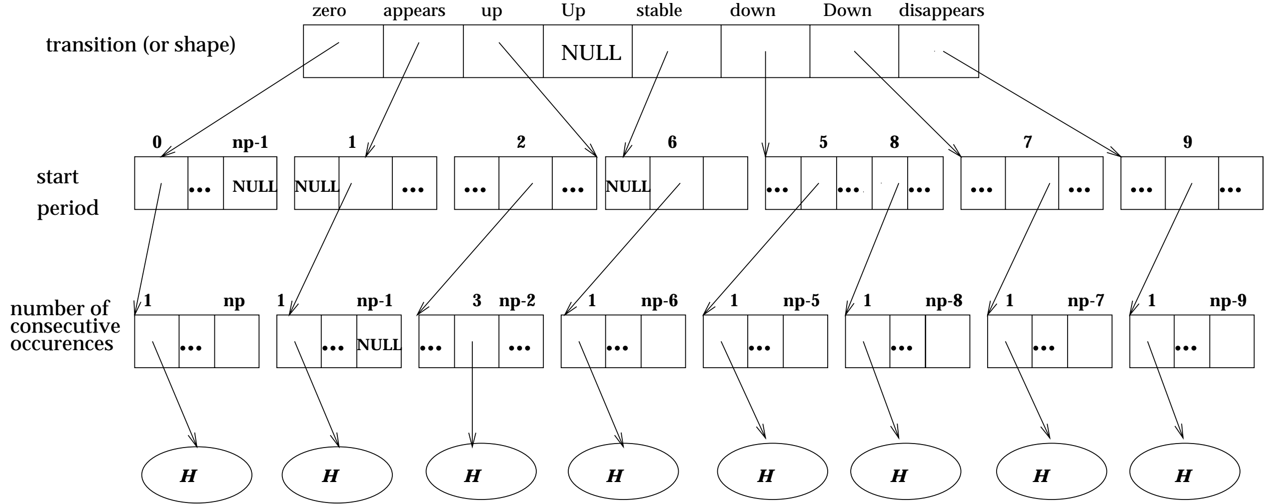


Figure 2: An index structure for $\mathcal{SDL}$ queries.

to $2n+1$ new primitive symbols[1]. Each of the original elementary shapes can clearly be expressed as the "union" of the corresponding new primitive elementary shapes. Intuitively, the fact that each of the original elementary shapes has an associated interval implies that most of the "intersections" between the original elementary shapes is empty and thus require no new primitive shapes, thereby controlling the blowup.

### 5.3 Shape Matching Using the Index

**Notation** In the following, $P$ and $D$ denote an elementary and a derived shape, respectively, $eval(D, [s, e])$ denotes the evaluation of shape $D$ within the interval $[s, e]$, and $p$ denotes the length of the interval, i.e., $p = e \Leftrightarrow s$. The result of $eval$ is a set of tuples $[oid, start, length]$, where $oid$ is the $object_id$, $start$ is the start period, and $length$ the length of the matched subsequence. The notation $shape[P].start[x].occur[y]$ means "get object identifiers that have $y$ occurrences of the shape $P$ starting from $x$", and represents index traversal. The tuples resulting from matching the null sequence have $start = s$ and $length = 0$.

#### 5.3.1 Operations on Elementary Shapes

We first consider the evaluation of elementary shapes and those shapes derived by applying multiples occurrences operators on elementary shapes.

- **Elementary shape**

[1]Extra primitive symbols may be needed to handle constraints on initial and final values.

$$eval(P, [s, e]) = \{[oid : o, start : i, length : 1] \mid \exists x, y \; (o \in shape[P].start[x].occur[y]) \wedge (\max(s, x) \leq i < \min(x + y, e))\}$$

- **exact**

$$eval(exact\ n\ P, [s, e]) = \{[oid : o, start : max(s, x), length : n] \mid \exists\ x, y \; (o \in shape[P].start[x].occur[y]) \wedge (x \leq e \Leftrightarrow n) \wedge (s + n \leq x + y) \wedge (min(e, y + x) \Leftrightarrow max(s, x) = n)\}$$

When $n = 0$, we cannot use directly the index to get subsequences that match the null sequence. Instead, they are computed by the following expression:

$$eval(exact\ 0\ P, [s, e]) = \{[oid : o, start : s, length : 0] \mid [o, s] \notin eval(atleast\ 1\ P, [s, e])[oid, start]\}$$

- **atmost**

$$eval(atmost\ n\ P, [s, e]) = \{[oid : o, start : max(s, x), length : min(e, x + y) \Leftrightarrow max(s, x)] \mid \exists\ x, y \; (o \in shape[P].start[x].occur[y]) \wedge (x < n) \wedge (s < x + y) \wedge (min(e, x + y) \Leftrightarrow max(s, x) \leq n)\} \bigcup eval(exact\ 0\ P, [s, e])$$

- **atleast**

$$eval(atleast\ n\ P, [s, e]) = \{[oid : o, start : max(s, x), length : min(e, x + y) \Leftrightarrow max(s, x)] \mid \exists\ x, y \; (o \in shape[P].start[x].occur[y]) \wedge (x \leq e \Leftrightarrow n) \wedge (s + n \leq x + y) \wedge (min(e, x + y) \Leftrightarrow max(s, x) \geq n)\}$$

When $n = 0$, $eval(exact\ 0\ P, [s, e])$ must be "unioned" to the above expression.

• **precisely, nomore, noless**
The evaluation of (`precisely/nomore/noless` $n$ $P$) within the interval $[s,e]$ is similar to (`exact/atmost /atleast` $n$ $P$) except that $n$ must be equal/greater /smaller than the sum of all $P$ occurrences in $[s,e]$.

### 5.3.2 Operations on Derived Shapes

The evaluation of more complex forms of derived shapes is performed using the index structure inductively.

• **concat**
The result of matching one shape constrains the interval in which the next shape should be searched. The following expression implements it for $n$=2; for $n$>2, the evaluation is performed inductively:

$$eval(concat\ D_1\ D_2, [s,e]) = \\ \bowtie_{(PR_1,PJ_1)} (eval(D_1,[s,e]), \bigcup_{t\in I_1} eval(D_2,[t,e]))$$

Here $I_1$ denotes the interval where the matching of $D_2$ starts. It results from the evaluation of $D_1$, and is given by $I_1 = [min(S_1.start + S_1.length), max(S_1.start + S_1.length)]$. $D_1$ is evaluated first, then $I_1$, then $D_2$, followed by a join operation between resulting sets, $S_1$ and $S_2$, using the predicate $PR_1 = (S_1.oid = S_2.oid) \wedge (S_2.start = S_1.start + S_1.length)$ and projection $PJ_1 = [oid : S_1.oid, start : S_1.start, length : S_1.length + S_2.length]$. The inductive evaluation for the concatenation of $n$ shapes stops either when the result of a join is empty or after all joins have been performed. In the former case, the evaluation returns an empty set. Since $S_i$ elements are sorted on *oid*, the join operations are implemented as *merge-join*.

• **Multiple Occurrences Operators**
We use the same evaluation schema as for the `concat`, replacing $D_i$ by $D$. The `exact` and `atmost` operators have the same stopping condition as `concat`. The `exact` operator returns the result of step $n$ if the result of step $n$+1 is empty, and the empty result otherwise. The `atmost` operator returns the result of step $i$ if $i \leq n$ and the result of step $i$+1 is empty. For `atleast` the evaluation stops when a join returns an empty set. It returns the result of step $i$ if $i \geq n$ and the step $i$+1 returns empty result.

• **any**

$$eval((any\ D_1 \ldots D_n),[s,e]) = \bigcup_{1\leq i\leq n} (eval(D_i,[s,e]))$$

• **in**
The length parameter of `in` defines a *family of intervals* inside interval $[s,e]$ where the match should be performed. Thus, `in` is implemented by the following expression:

$$eval((in\ n\ D),[s,e]) = \bigcup_{s\leq i\leq e-n} (eval(D,[i,i+n]))$$

The `precisely`, `nomore`, and `noless` operators have the same evaluation schema as `exact`, `atmost`, and `atleast`, respectively, but a different definition for the interval, predicate, and projection, because they allow gaps and overlap between their shape arguments. Their definitions for the interval, predicate and projection require an offset of at least one time period between two consecutive shapes. On the other hand, `inorder` does not accept overlap, and its evaluation schema is the same as for `concat` with the exception that its definition of the interval, predicate, and projection requires that two consecutive shapes, $D_1$ and $D_2$, are separated by at least the length of the subsequence matched by $D_1$.

Since we allow gaps and overlap between shapes inside `and`, it is implemented as a join between the set of subsequences that match $D_1$ and $D_2$. The shape order in the sequence does not matter. The `or` operator over two shapes, $D_1$ and $D_2$, is implemented as the union of the set of subsequences that match $D_1$ and the set of sequences that match $D_2$.

## 6 Summary

We presented $\mathcal{SDL}$, a shape definition language for retrieving objects based on shapes contained in the histories associated with the objects. $\mathcal{SDL}$ is designed to be a small, yet powerful, language for expressing naturally and intuitively a rich variety of queries about the shapes found in histories. $\mathcal{SDL}$ is equivalent in expressive power to the regular expressions when finding if a given sequence matches a particular shape. In the case of continuous matching [8], where one finds all the subsequences of a given sequence that match a particular shape, $\mathcal{SDL}$ provides context information that regular expressions are unable to. Thus, $\mathcal{SDL}$ can discard the non-maximal subsequences thereby eliminating useless clutter, whereas the regular expressions cannot provide this service since they are unable to "look-ahead" to provide context information.

A novel feature of $\mathcal{SDL}$ is its ability to perform "blurry" matching where the user gives the overall shape but not all the specific details. $\mathcal{SDL}$ is efficiently implementable — its operators are designed to limit non-determinism, which in turn reduces back-tracking. An $\mathcal{SDL}$ query expression can be rewritten into a more efficient form using transformation rules and its execution can be speeded using our index structure.

**Acknowledgment** We thank Stefano Ceri and John Shafer for useful discussions.

## 7 Appendix A: Formal Semantics for $\mathcal{SDL}$

**Notation** Let $\mathcal{H}$ be a sequence of real values describing a history. Formally, a sequence is a function from an interval into the real numbers where an interval is a finite set of consecutive non-negative integers. An interval is frequently denoted by $[i,j]$. By $length(\mathcal{H})$, we indicate the number of elements in the domain of the function that represents the sequence $\mathcal{H}$. Every element in $\mathcal{H}$ is identified by its position in the sequence. The first element for the whole history is in position 0. We refer to the symbol in position $i$ as $\mathcal{H}[i]$, with $0 \le i < length(\mathcal{H})$.

Let $S \subseteq \mathcal{H}$ be a subsequence of $\mathcal{H}$ defined as follows. Each element in $S$ is identified by its position in the original sequence $\mathcal{H}$ and elements in $S$ are in the same order they are in $\mathcal{H}$. The first element of $S$ is referred to as $first(S)$, while the last as $last(S)$.

The subsequence of $\mathcal{H}$ from position $i$ to position $j$ inclusive is represented as $\mathcal{H}[i,j]$, where $0 \le i \le j < length(\mathcal{H})$. Similarly, $S[i,j]$, where $first(S) \le i \le j \le last(S)$, indicates a subsequence of $S$. The length of $S[i,j]$ is defined as $length(S[i,j]) = j \Leftrightarrow i + 1$. Notice that $S[i,j][k,l] = S[max(i,k), min(j,l)]$.

There exists an alphabet $\mathcal{A}$ of symbols and a mapping that can map the values of any two consecutive elements of $\mathcal{H}$ into the symbols of $\mathcal{A}$. Each symbol corresponds to an elementary shape. An elementary shape induces a class containing all the subsequences of $\mathcal{H}$ of length 2 that satisfy the definition of the corresponding alphabet. We use the notation $s \in P$ to indicate a sequence $s$ belonging to the class induced by $P$'s definition, where $P$ is an elementary shape.

The $\simeq$ operator is an application from a pair *(S, P)*, where $S$ is a sequence and $P$ a shape, to a possibly empty *set of intervals*. This resulting set of intervals contains all subsequences of $S$ that match the shape $P$. Notice that the definition implies that if $[k,l] \in \mathcal{H}[i,j] \simeq P$, then $i \le k \le l \le j$. The interval $[k,k]$ denotes any null sequence since any elementary shape matches only intervals that have a single transition (i.e. are of the form $[k,k+1]$).

**Elementary shapes.** Let $\mathcal{H}$ be a sequence and $P$ one of the symbols in $\mathcal{A}$. Then
$[k,l] \in \mathcal{H}[i,j] \simeq P$ iff $\mathcal{H}[k,l] \in P$ and $i \le k \le l = k+1 \le j$.

**Derived Shape `any`.** Let $\mathcal{H}$ be a sequence and $P_1 \ldots P_n$ some shapes. Then

$$\mathcal{H}[i,j] \simeq (\texttt{any}\ P_1\ P_2\ \ldots\ P_n\ ) = \bigcup_{k=1}^{n} \mathcal{H}[i,j] \simeq P_k.$$

**Derived Shape `concat`.** The syntax of the concatenation operator is:

$(\texttt{concat}\ P_1\ P_2\ \ldots\ P_n)$ for $n \ge 0$.

The following formulas give the semantics:

$\mathcal{H}[i,j] \simeq (\texttt{concat}\ \ ) = \{[k,k] \mid i \le k \le j\}$.

If $n \ge 1$, then $[k,m] \in \mathcal{H}[i,j] \simeq (\texttt{concat}\ P_1\ \ldots\ P_n)$ iff there exists an $l$ such that $[k,l] \in \mathcal{H}[i,j] \simeq P_1$ and $[l,m] \in \mathcal{H}[l,j] \simeq (\texttt{concat}\ P_2\ \ldots\ P_n)$.

**Derived Shapes: `exact`, `atleast`, `atmost`.** The syntaxes are:

$(\texttt{exact}\ n\ P\ )$
$(\texttt{atleast}\ n\ P\ )$
$(\texttt{atmost}\ n\ P\ )$
where $n \ge 0$.

These operators provide richer forms of concatenation. Their semantics is described as follows.

$[k,l] \in \mathcal{H}[i,j] \simeq (\texttt{atleast}\ n\ P\ )$ iff
$\neg\exists m \le k\ ([m,k] \in \mathcal{H}[i,k] \simeq P)$ and
$\neg\exists m \ge l\ ([l,m] \in \mathcal{H}[l,j] \simeq P)$ and
$\exists m \ge n\ ([k,l] \in \mathcal{H}[i,j] \simeq (\texttt{concat}\ P_1\ \ldots\ P_m)$
where $P_1 = \ldots = P_m = P)$

$[k,l] \in \mathcal{H}[i,j] \simeq (\texttt{atmost}\ n\ P\ )$ iff
$\neg\exists m \le k\ ([m,k] \in \mathcal{H}[i,k] \simeq P)$ and
$\neg\exists m \ge l\ ([l,m] \in \mathcal{H}[l,j] \simeq P)$ and
$\exists m \le n\ ([k,l] \in \mathcal{H}[i,j] \simeq (\texttt{concat}\ P_1\ \ldots\ P_m)$
where $P_1 = \ldots = P_m = P)$

$[k,l] \in \mathcal{H}[i,j] \simeq$ (exact $n$ $P$ ) iff
$\neg\exists m \leq k$ ($[m,k] \in \mathcal{H}[i,k] \simeq P$) and
$\neg\exists m \geq l$ ($[l,m] \in \mathcal{H}[l,j] \simeq P$) and
($[k,l] \in \mathcal{H}[i,j] \simeq$ (concat $P_1$ ... $P_n$)
where $P_1 = \ldots = P_n = P$)

**Derived Shape: in.** The syntax is:
(in $n$ $P$) where $n \geq 0$ indicates the length of the sequence in terms of time periods (transitions) for which the condition expressed by the $P$ argument must hold.

$\mathcal{H}[i,j] \simeq$ (in $n$ $P$ ) $= \{[k, k+n] \mid i \leq k \wedge k+n \leq j \wedge [k, k+n] \in \mathcal{H}[k, k+n] \simeq P\}$.

**Derived Shapes: nomore, noless, precisely.** The syntaxes are:
(nomore $n$ $P$ )
(noless $n$ $P$ )
(precisely $n$ $P$ )
where $n \geq 0$.

Even though these forms make sense in general, they are restricted to use within the **in** shape.

$[k,l] \in \mathcal{H}[i,j] \simeq$ (noless $n$ $P$ ) iff $i \leq k \leq l \leq j$ and $card(\mathcal{H}[k,l] \simeq P) \geq n$.

$[k,l] \in \mathcal{H}[i,j] \simeq$ (nomore $n$ $P$ ) iff $i \leq k \leq l \leq j$ and $card(\mathcal{H}[k,l] \simeq P) \leq n$.

$[k,l] \in \mathcal{H}[i,j] \simeq$ (precisely $n$ $P$ ) iff $i \leq k \leq l \leq j$ and $card(\mathcal{H}[k,l] \simeq P) = n$.

**Derived Shape: inorder.** The syntax is:
(inorder $P_1$ ... $P_n$) for $n \geq 0$.

Even though this form makes sense in general, it is restricted to use within the **in** shape.

$[k,m] \in \mathcal{H}[i,j] \simeq$ (inorder $P_1$ ... $P_n$) iff there exist $l_0, k_1, l_1, \ldots, k_n, l_n$ such that $i = l_0 \leq k \leq k_1 \leq l_1 \leq k_2 \leq l_2 \ldots \leq k_n \leq l_n \leq m \leq j$ and $[k_u, l_u] \in \mathcal{H}[l_{u-1}, j] \simeq P_u$ for $1 \leq u \leq n$.

**Derived Shapes: and, or.** The syntaxes are:
(and $P_1$ ... $P_n$ )
(or $P_1$ ... $P_n$ )
where $n \geq 0$.

Even though these forms make sense in general, they are restricted to use within the **in** shape.

$\mathcal{H}[i,j] \simeq$ (or $P_1 \ldots P_n$) $= \mathcal{H}[i,j] \simeq$ (any $P_1 \ldots P_n$).

$$\mathcal{H}[i,j] \simeq (\texttt{and}\ P_1\ P_2\ \ldots P_n) = \bigcap_{k=1}^{n} \mathcal{H}[i,j] \simeq P_k.$$

## References


[1] R. Agrawal, G. Psaila, E. L. Wimmers, and M. Zaït. Querying shapes of histories. IBM Research Report RJ 9962 (87921), IBM Almaden Research Center, San Jose, California, June 1995.

[2] D. J. Berndt and J. Clifford. Using dynamic time warping to find patterns in time series. In *KDD-94: AAAI Workshop on Knowledge Discovery in Databases*, pages 359–370, Seattle, Washington, July 1994.

[3] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts, and detection. In *Proc. of the VLDB Conference*, pages 606–617, Santiago, Chile, September 1994.

[4] R. D. Edwards and J. Magee. *Technical Analysis of Stock Trends.* John Magee, Springfield, Massachusetts, 1966.

[5] S. Gatziu and K. Dittrich. Detecting composite events in active databases using petri nets. In *Proc. of the 4th Int'l Workshop on Research Issues in Data Engineering: Active Database Systems*, pages 2–9, February 1994.

[6] N. Gehani, H. Jagadish, and O. Shmueli. Composite event specification in an active databases: Model & implementation. In *Proc. of the VLDB Conference*, pages 327–338, Vancouver, British Columbia, Canada, August 1992.

[7] J. E. Hopcroft and J. D. Ullman. *Introduction to Automaton Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[8] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A model for sequence databases. In *Proc. of the IEEE Int'l Conference on Data Engineering*, Taiwan, 1995.

[9] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, October 1992.