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. We used $\mathcal{SDL}$ to "mine the mined rules" in a data mining application and found that the capability to query the behavior of the mined rules over a period of time led to the discovery of new and interesting information.

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

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 $\mathcal{A}$

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 [5]. 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 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 [3] [9] [11] [12] [14] 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 [4] [6] [7] concentrate on detecting the endpoints of events rather than concentrating on intervals as $\mathcal{SDL}$ does. The

$\mathcal{SEQ}$ work of [10] focused on building a framework for describing constructs from various existing sequence models. Later in the paper, after we have presented $\mathcal{SDL}$, we will make a more detailed comparison with some of the existing work.

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. We also compare it to existing related languages. Formal proofs of equivalence are given in Appendix C. 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. In Section 6, we report our experience with using this language on real-life data, and conclude by giving directions for future work.

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 [1] :

`(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 $\mathcal{H}$:

(0 0 .02 .17 .35 .50 .45 .43 .15 .03 0)

Figure 1 graphically shows this sequence.

[1] We have adopted a Lisp-like notation for our expression language for two reasons. First, it gives us the possibility to incrementally extend the language with new features while maintaining uniformity. Second, the basic notion of list or sequence that characterizes Lisp also underlies our view of a shape of a history.

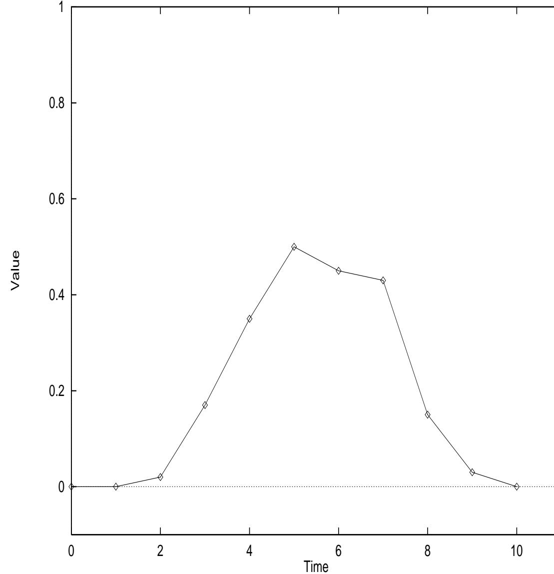

Figure 1: Time Sequence $\mathcal{H}$ = (0 0 .02 .17 .35 .50 .45 .43 .15 .03 0)

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 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.

Building the Alphabet Table. A straightforward way of building the alphabet table is to ask the user to provide *lb, ub, iv* and *fv* values for each symbol. However, the user may have difficulty in providing bounds for each symbol. A better approach will be to let the user give examples of values for transition symbols and infer the alphabet table from these examples.

Given a fixed number of classes and example entities belonging to each class (called training set), a classifier[13] generates characteristic functions to describe each class. We can synthesize the alphabet table by treating it as a classification problem. There will be as many classes as the number of symbols in the alphabet. Besides the class label, each example tuple in the training set

will have three attributes: *iv, fv* and *val.* The first two are categorical attributes and will have one of two possible values: `zero` and `nonzero`. The third attribute is a numerical attribute.

The training set is generated as follows. Show graphically the user several histories and let the user interactively label each segment of the history with a symbol that best describes the transition. For each transition so labeled, compute the values of the aforementioned three attributes. The attribute iv (fv) will have `zero` or `nonzero` value depending on whether the initial (final) value of the transition was zero or non-zero. The value for *var* is simply the difference between the final and the initial value of the transition.

Having so generated the training set, a simple classifier synthesizes the alphabet table as follows. A symbol is assigned `zero` or `nonzero` value for iv (fv) if there is no exception to these values for the attribute iv (fv) in the training set for this symbol (or the exceptions are below some threshold); otherwise, `anyvalue` is assigned. The values for *lb* and *ub* for a symbol are determined by computing bounds on the values of the attribute *var* for the symbol in the training set. The user can then fine tune the alphabet table as necessary.

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 `up`s) has reversed (indicated by two `stable`s or `down`s, 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 $\{[k,k] \| 0 \leq k \leq 1 \vee 6 \leq k \leq 10\}$.

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 `up`s. 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:

(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 [10]) 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 Appendix C. 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 Appendix C.

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. Rather than specifying the shape precisely, the user places certain restrictions that the shape must satisfy. 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. This query does not place restrictions on where these `down`s occur, it merely limits the entire subsequence to a total of no more than 2 `down`s. Many other characteristics of the shape of the given length can be supplied and the full power of $\mathcal{SDL}$ is available for specifying these characteristics. 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 Appendix C 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 Appendix B 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. In fact, the permutation example can be generalized. In Appendix B, we give a natural class of blurry queries and show that for every member of that class, any equivalent regular expression has at least exponential size. (See Theorem 8.3 in Appendix B for a precise statement of the result.)

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` $P_i(x_1, \ldots, x_i)$(`any` (`concat` $x_1 P_{i-1}(x_2, \ldots, x_i)$) ... (`concat` $x_i P_{i-1}(x_1, \ldots, x_{i-1})$))) 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 [8]). 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 conjunc-

tions 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. In short, $\mathcal{SDL}$ permits the readable and compact expression of shapes that can be implemented efficiently and are important to data mining applications!

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).

3.4. Comparison to other work

As we mentioned in Section 1, much of the earlier related work has a different design focus. The differing design points influence which expressions are easy to write, understand, optimize, and evaluate. For instance, the work of [7] provides a mechanism for expressing queries that also has the full power of regular expressions. However, this work focuses more on finding the endpoints of "events" rather than concentrating on intervals as $\mathcal{SDL}$ does. Suppose one wants to find all subsequences of a given sequence that consist of a single `up` immediately followed by a single `down`. The $\mathcal{SDL}$ expression for this query is (`concat up down`). This can be expressed as $sequence$(`up`, `down`) in the notation of [7]. (If one only used the base operations of [7], this could be expressed as $relative(\texttt{up}, first) \wedge$ `down` where $first$ is $!relative(any, any)$.) The answer would consist of the endpoints of matched sequences and a typical answer might be $\{6\}$ whereas in $\mathcal{SDL}$, the answer would be a set of intervals and the corresponding typical answer would be $\{[4,6]\}$. If desired, correspondence tuples in [7] could be used to describe the corresponding intervals.

The $\mathcal{SEQ}$ work of [10] is focused on building a model for describing queries rather than a language for expressing them. As such it is more interested in describing systems such as $\mathcal{SDL}$ than it is in giving a syntax for expressing queries. While the current $\mathcal{SEQ}$ model lacks the power to do continuous matching on all regular expressions, it does provide some interesting optimization techniques.

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:

$$
\begin{aligned}
(\texttt{any}\ P_1\ (\texttt{any}\ P_2\ P_1)) &\Leftrightarrow (\texttt{any}\ P_1\ P_2\ P_1) \text{ -- associativity}\\
&\Leftrightarrow (\texttt{any}\ P_1\ P_1\ P_2) \text{ -- commutativity}\\
&\Leftrightarrow (\texttt{any}\ P_1\ P_2) \quad \text{ -- idempotence}
\end{aligned}
$$

4.2. Distributivity

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

$$
\begin{aligned}
(\texttt{concat}\ P_1\ (\texttt{any}\ P_2\ P_3)) &\Leftrightarrow (\texttt{any}\ (\texttt{concat}\ P_1\ P_2)\ (\texttt{concat}\ P_1\ P_3))\\
(\texttt{and}\ P_1\ (\texttt{or}\ P_2\ P_3)) &\Leftrightarrow (\texttt{or}\ (\texttt{and}\ P_1\ P_2)\ (\texttt{and}\ P_1\ P_3))
\end{aligned}
$$

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:

$$
(\texttt{concat}\ P_1\ P_2\ P_2\ \ldots\ P_2\ P_3) \Leftrightarrow (\texttt{concat}\ P_1\ (\texttt{exact}\ \mathrm{n}\ P_2)\ P_3)
$$

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:

$$\begin{array}{ll} (\texttt{or}\ (\texttt{inorder}\ P_1\ P_2)\ (\texttt{inorder}\ P_2\ P_1)) & \not\Leftrightarrow (\texttt{and}\ P_1\ P_2) \\ (\texttt{inorder}\ P\ \ldots\ P) & \not\Leftrightarrow (\texttt{precisely}\ n\ P) \end{array}$$

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 kth element of a second-layer array. This array will have $np \Leftrightarrow k$ elements, starting from the kth 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

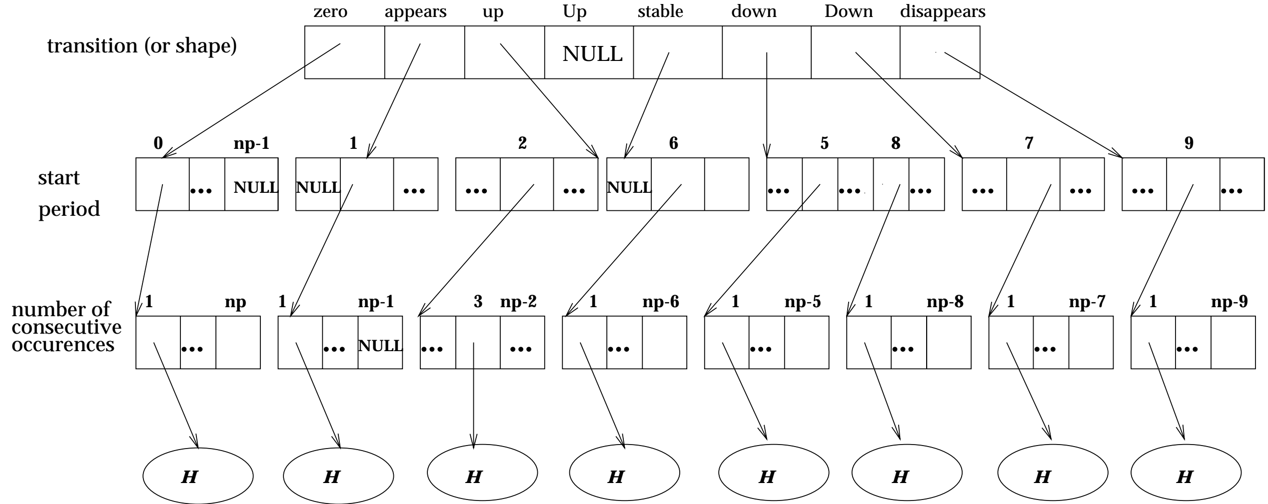

Figure 2: An index structure for $\mathcal{SDL}$ queries.

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 $\mathtt{E} \Leftrightarrow (\mathtt{any}\ \mathtt{P}_1\ \ldots\ \mathtt{P}_n)$ eliminates the elementary shape `E` in favor of the corresponding primitive elementary shapes $\mathtt{P}_1\ \ldots\ \mathtt{P}_n$ 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 to $2n+1$ new primitive symbols[2]. 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**

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

- **exact**

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

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**

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

- **atleast**

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

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

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. Conclusion

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 [10], 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.

Experience. A prototype of the shape query system described here has been implemented in C++ on the AIX system as part of the Quest project at IBM. We tested it against two large datasets in a data mining application. The first dataset was from a mail-order company and consisted of five years of transaction history. The second dataset was from a market research company that provides marketing information to the retail industry and consisted of three years of point-of-sales data. The first dataset had roughly 2.9 million transactions, whereas the second dataset had more than 6.8 million transactions.

In both the cases, we divided data on monthly basis and mined all association rules[1] for each dataset. For rules discovered, we saved three parameter histories for each rule in the rulebase: support, confidence, and the product of support and confidence values. We ran several complex shape queries on these histories. In all cases, we got interactive response for the queries. The mining algorithms such as [1] are excellent in taking gigabytes of data and reducing it into hundreds or thousands of rules. However, the information generated is still too large and there is need to "mine

the mined rules". Having the capability to see the behavior of the mined rules over a period over time and query the behavior led to discovery of new and interesting information. It also provided useful cues for further exploration and querying of data.

Future Work. The work presented in this paper can be extended along several dimensions. We mention a few here:

- *Integration with a Relational Database System.* Our current prototype is a stand-alone implementation and focuses only on the new features introduced by the shape query language. In a production system, the shape predicates must be integrated with the current facility provided by a query language such as SQL.
- *Further reduction in non-determinism.* The `any` construct is the only one that introduces non-determinism. Should `any` be eliminated in favor of `first`? For example (`first` P_1 P_2) would not match P_2 if P_1 matched. Such a system would not have any non-determinism. Alternatively, perhaps `first` should be added anyway since it produces a kind of conditional matching.
- *Recursive parameterized shapes.* Currently, parameterized shape definitions are not allowed to be recursive. Could this restriction be weakened? Lifting it entirely would increase the expressive power and that would have implementation implications.

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 $0 \leq i < length(\mathcal{H})$.

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 $0 \leq i \leq j < length(\mathcal{H})$. Similarly, $S[i, j]$, where $first(S) \leq i \leq j \leq last(S)$, 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 $i \leq k \leq l \leq j$. 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 $i \leq k \leq l = k+1 \leq j$.

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

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

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

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

The following formulas give the semantics:

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

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

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

$\texttt{(exact } n\ P\ \texttt{)}$

$\texttt{(atleast } n\ P\ \texttt{)}$

$\texttt{(atmost } n\ P\ \texttt{)}$

where $n \geq 0$.

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

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

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

$[k,l] \in \mathcal{H}[i,j] \simeq \texttt{(exact } n\ P\ \texttt{)}$ 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
$([k,l] \in \mathcal{H}[i,j] \simeq \texttt{(concat } P_1\ \ldots\ P_n\texttt{)}$ 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` $P_1 \ \ldots \ P_n$) 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` $P_1 \ \ldots \ P_n$) iff there exist $k_1, l_1, \ldots, k_n, l_n$ 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` $P_1 \ \ldots \ P_n$)

(`or` $P_1 \ \ldots \ P_n$)

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` $P_1 \ldots P_n$).

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

8. Appendix B: Exponential Results

It is well known that adding conjunction to regular expressions does not increase expressive power but does allow exponential compaction of some expressions (see Chapter 3 exercises in [8]). Permutation expressions are an example of this well-known exponential compaction. The proof that permutation expressions require at least exponential size (Theorem 8.6) can be adapted to show that (under some separability assumptions) *every* blurry shape in a very large and natural class requires exponential size to express using regular expressions! This natural class of blurry queries includes the conjunction of `precisely,noless` shapes (Theorem 8.3).

We start with a little bit of notation. Let $\mathcal{A}$ be a set of disjoint elementary shapes. (See Section 5.2 for a discussion of making the elementary shapes disjoint.) $\mathcal{A}$ forms the set of characters from which strings are built. Juxtaposition is used to denote the concatenation of two strings. $L(s)$ denotes the length of the string s. Each string implicitly defines a shape that is the concatenation of its constituent elementary shapes. Instead of histories, we use an equivalent string representation since whether a shape matches a history depends only on its corresponding string (which is unique by the disjointness assumption.) A string based treatment (rather than a history based treatment) facilitates comparison with regular expression notation. (Note that the length of the string is one less than the length of the corresponding history.) We use $N_P(s)$ to denote the number (counting multiplicities) of substrings of s that match P. To be a bit more formal, let $\mathcal{H}[0, L(s)]$ be any history that has s as its associated string. (The rest of the discussion is independent of the choice of history.) $N_P(s) = card(\mathcal{H}[0, L(s)] \simeq P)$. To put this in string terms, $N_P(s) = card(\{[k,l) | 0 \leq k \leq l \leq L(s) \ \wedge \ s[k,l) \text{ matches } P\}$ where $s[k,l)$ denotes the substring starting at k and ending just before l. Thus, $s[k,k)$ represents the empty substring starting at k. Since `atleast`, `atmost` and `exact` have greedy semantics, matching has a contextual component to it; so two substrings that have exactly the same characters in the same order might have different matching behavior because they occur in different contexts.

We call a symbol in a regular expression "primitive" iff it is a member of the alphabet. In other words, the "combining" symbols (such as "|", concatenation, parentheses, and "*") are not counted as primitive symbols. Clearly, showing that exponentially many primitive symbols are needed is stronger than showing exponential size is required since the size of a regular expression is at least as large as the number of (occurrences of) primitive symbols within it.

8.1. Main Result

When a divider (see Definition 8.1) for a shape occurs in a history, the divider has the effect of dividing the history into two pieces for the purpose of counting the number of matches to the shape. In addition, it should be the case that a divider (even when concatenated with a portion of itself) does not match the shape.

Definition 8.1. *Let P be a shape and s be a string. Then s is a* divider *for P iff*

1. *$N_P(s_1 s s_2) = N_P(s_1 s) + N_P(s s_2)$ for all strings s_1, s_2, and*
2. *if s' is any prefix of s, then $N_P(s s') = 0$.*

Note that the entire sequence s is a prefix of itself so property 2 in Definition 8.1 implies that $N_P(ss) = 0$. If a_1, a_2 are distinct characters, then a_2a_1 is an example of a divider for the shape (concat (atleast 2 a_1) (atleast 2 a_2)).

We use δ to denote the "equality tester" function defined as $\delta(i,j) = \begin{cases} 1 & \text{if i = j} \\ 0 & \text{otherwise.} \end{cases}$

Definition 8.2. *Let u, v be positive integers. Let p, q be non-negative integers. Shapes $((P_1, \ldots, P_p), (Q_1, \ldots, Q_q))$ are called (u, v)-0 iff there exist non-empty strings $s_0, s_1, \ldots, s_p$ such that the following all hold (for all $j, j' \in \{1, \ldots, p\}$, and $k \in \{1, \ldots, q\}$)*

1. *$L(s_0) \leq u$ and s_0 is a divider for every shape P_j, Q_k.*
2. *$L(s_j) \leq v$ and $N_{P_j}(s_0 s_{j'} s_0) = \delta(j, j')$ and $N_{Q_k}(s_0 s_j s_0) = 0$*

The idea behind part 1 is that s_0 is a divider for all the shapes and u is an upper bound to the length of that divider. The idea behind part 2 is that the string s_j serves to distinguish each P_j shape from all the other $P_{j'}$ shapes as well as the all Q_k shapes by providing a string that matches only P_j. Furthermore, v is an upper bound to the length of every string that is used to distinguish a P_j. The Q_k shapes do not need distinguishing strings of their own because, in the proof of the Exponential Blowup Theorem, they only occur inside of the `nomore` construct and never force a string to appear in a match. It is worth remarking that if P_j and $P_{j'}$ are equivalent shapes (for $j \neq j'$), then it is never possible to separate them since neither has a string to distinguish it from the other.

We next state the main result of this Appendix. Consider the blurry query (in m (and (precisely j_1 P_1)...(precisely j_p P_p) (noless j_{p+1} P_{p+1})...(noless $j_{p+p'}$ $P_{p+p'}$))). Theorem 8.3 shows that, provided certain separability conditions are met, the size of an equivalent regular expression grows *exponentially* in the number of conjuncts! Note that `nomore` subshapes are allowed to appear but do not add to the exponential blowup.

Theorem 8.3 (Exponential Blowup Theorem). *For all non-negative integers $p, p', q, k_1, \ldots, k_q$, and all positive integers $u, v, j_1, \ldots, j_{p+p'}$, and all (u, v)-separable shapes $((P_1, \ldots, P_{p+p'}), (Q_1, \ldots, Q_q))$, and all integers $m \geq 2(u + v)(j_1 + \ldots + j_{p+p'}) + 2u$, every regular expression equivalent to*

(in m (and (precisely j_1 P_1)...(precisely j_p P_p)
(noless j_{p+1} P_{p+1})...(noless j_p $P_{p+p'}$)
(nomore k_1 Q_1)...(nomore k_q Q_q)))

has at least $\frac{m}{2u} \cdot 2^{p+p'}$ occurrences of primitive symbols.

Proof: This theorem follows easily from Theorem 8.7 by noting that in this theorem, $j_1, \ldots, j_{p+p'}$ are required to be positive (rather than merely non-negative as in Theorem 8.7). Hence $2^{p+p'} \leq (1 + j_1) \ldots (1 + j_{p+p'})$. $\square$

In Theorem 8.3, any shape P'_0 equivalent to the given one is sufficient since any regular expression equivalent to P'_0 is equivalent to the blurry shape given in Theorem 8.3. In particular, the order

of the conjuncts can be changed so that the precisely, noless, and nomore subshapes can be intermixed without affecting the conclusion of Theorem 8.3. The restriction that j_i be positive is not particularly burdensome since (precisely 0 P) can be translated to (noless 0 P) and (atleast 0 P) can be dropped without affecting the result. Note that if or is used instead of and to combine subshapes, there is no exponential blowup since regular expressions for each of the disjuncts can be combined easily using the "|" regular expression disjunction operator.

8.2. Separability Considerations

One of the key issues in separating shapes is the ability to find dividers. The next proposition gives an easy sufficient condition for finding dividers.

Proposition 8.4. *Let a_0 be an elementary shape. If P is a shape built from the elementary shapes other than a_0 using the constructs* (concat $P_1 \ldots P_n$), (any $P_1 \ldots P_n$), (atleast n P_0), *and* (exact n P_0) *where $n \geq 1$, then a_0 is a divider for P.*

Proof: First note that P never matches the empty string since $n \geq 1$ and the atmost construct is omitted. It is easy to check by induction on the structure of P that a string s in the either the context of $s_0 s s_1 a_0 s_2$ or the context of $s_2 a_0 s_0 s s_1$ matches P iff it matches in the context of $s_0 s s_1$ and that no string containing a_0 ever matches P. The result clearly follows. □

Of course, separability is not necessary for the exponential blowup in Theorem 8.3. For example, suppose the query is (in m (and ...(noless 3 P)(nomore 3 P)...)). Separability clearly fails since there is no string to distinguish the noless shape from the nomore shape. If this is the only source of non-separability, then the query (in m (and ...(precisely 3 P)...)) has exponential blowup since Theorem 8.3 applies to the new shape and it is clearly equivalent to the original one.

Even though there are many cases where separability fails but the regular expression still grows exponentially, the separability condition of Theorem 8.3 is not superfluous. For example, consider the blurry query P_0 = (in m (and (precisely k_1 P_1)...(precisely k_n P_n))) where $P_1, \ldots, P_n$ are all equivalent shapes. (Note that the shapes $P_1, \ldots, P_n$ might have very different forms since equivalent shapes need not have the same form. Even though equivalence of shapes is decidable (since each shape is equivalent to a regular expression), there is no easy syntactic test for equivalence.) It is clear that for $n \geq 2$, the shapes $(P_1, \ldots, P_n)$ are not separable if they are equivalent; therefore, Theorem 8.3 does not apply in this case. If $k_i \neq k_j$ for some i, j, then P_0 is never satisfied; in this case, P_0 is equivalent to (any) which has no primitive symbols and never matches anything. If all the k_i are equal, then P_0 is equivalent to (in m (precisely k_1 P_1)) and there is no dependence on the number of conjuncts (n) in the shape P_0 since all the conjuncts are redundant. For example, in the case where there are only two elementary shapes a_0, a_1, the query (in m (and (precisely 1 a_1)...(precisely 1 a_1))) which is equivalent to the query (in m (precisely 1 a_1)) can be expressed with a regular expression with at most m^2 primitive symbols since there are exactly m different strings (one for each possible position of a_1 amidst $m-1$ occurences of a_0) all of length exactly m that match (in m (precisely 1 a_1)).

8.3. Proof of Main Result

Our next major goal is to prove Theorem 8.7 upon which Theorem 8.3 is based. Let's begin by reviewing the proof that every permutation expression requires at least exponential size to express in regular expression notation.

The next lemma gives us a way distinguishing occurences of primitive symbols within regular expressions. Note that the "!" marker always follows a character and the η function picks out an occurences of that character in the regular expression. The occurences so chosen "participates" in the matching of the string at the point of the given character.

Lemma 8.5. *For every regular expression r, there is a function η_r that returns an occurrence of a primitive symbol in r such that the following all hold:*

1. *the domain of η_r is the set of all items of the form $s_0!s_1$ where s_0 is a non-empty string and the concatenation $s_0 s_1$ matches r*
2. *$\eta_r(s_0!s_1)$ is an occurrence (in r) of the last character in s_0*
3. *if $\eta_r(s_0!s_1) = \eta_r(s_0'!s_1')$, then $s_0 s_1'$ matches the regular expression r as well.*

Proof: First we inductively construct an NFA based on r that will be denoted by $N(r)$. Every state will be tagged either by ϵ or by one of the characters in the alphabet. The tag indicates that every arc into that state will have as its label the tag for that state. Furthermore, there will be a one-to-one correspondence between the occurrences of primitive symbols in r and the states tagged with the primitive symbols. Inductively, all NFAs will have a start state and a single accept state and both states will be tagged with ϵ. If a_i is a primitive symbol in r, create the NFA $N(a_i)$ with three states: a start state (tagged with ϵ), an accept state (tagged with ϵ), and an intermediate state (tagged with a_i). Let there be an ϵ arc from the intermediate state to the accept state and an arc labeled with a_i from the start state to the intermediate state. Notice that throughout the rest of the proof, no other states tagged by a_i will be created nor will any other copies of this state be made. If the regular expression is (`concat` r_1 r_2), create $N($`concat` r_1 $r_2)$ by drawing an ϵ arc from the accept state of $N(r_1)$ to the start state of $N(r_2)$, let the start state of $N($`concat` r_1 $r_2)$ be the start state of $N(r_1)$, and let the accept state of $N($`concat` r_1 $r_2)$ be the accept state of $N(r_2)$. If the regular expression is $r_1 \mid r2$, create a new start state and a new accept state (both tagged with ϵ) and add ϵ arcs from the new start state to the start states of $N(r_1)$ and $N(r_2)$ and add ϵ arcs from the accept states of $N(r_1)$ and $N(r_2)$ to the new accept state. If the regular expression is r_1^*, form $N(r_1^*)$ simply by adding an ϵ arc from the accept state of $N(r_1)$ to the start state of $N(r_1)$. Let a_0 be the last primitive symbol in s_0. Define $\eta_r(s_0!s_1)$ to be that occurrence of a_0 that corresponds to the state tagged with a_0 that is entered after matching the s_0 portion of the string $s_0 s_1$. Since $N(r)$ is a non-deterministic finite automaton, this choice of state is not necessarily unique; any state that leads to acceptance when the remainder of the string (i.e., s_1) is scanned will do. It is clear that the first two properties are met. Assume that $\eta_r(s_0!s_1) = \eta_r(s_0'!s_1)$. It is clear that after seeing either s_0 or s_0' a state is entered from which either s_1 or s_1' can complete the journey to an accept state. Hence the third property is true as well. □

Theorem 8.6. *Every permutation (on n characters) regular expression has at least $n \cdot 2^{n-1}$ primitive symbols.*

Proof: Let r be regular expression that is permutation expression for the n primitive symbols $a_1, \ldots, a_n$. We show that there are at least 2^{n-1} occurences of every symbol $a_1, \ldots, a_n$ from which the result clearly follows. For example, we show that there at least 2^{n-1} occurences of a_1. Given a subset S of $\{a_2, \ldots, a_n\}$, let $P(S)$ be any (arbitrarily chosen) permutation expression on the elements of S. Given a subset S, define $F(S) = \eta_r(P(S)a_1!P(S'))$ where $S' = \{a_2, \ldots, a_n\} \Leftrightarrow S$. By Lemma 8.5, it is clear that $F(S)$ picks out an occurrence of a_1 in r. Since there are 2^{n-1} subsets of $\{a_2, \ldots, a_n\}$, it is clearly sufficient to show that $F(S) \neq F(T)$ if $S \neq T$. Suppose that $F(S) = F(T)$. Then by Lemma 8.5, $P(S)a_1P(T')$ matches r. Since r matches only permutations, it follows that T' is the complement of S from which it follows that $T = S$. Hence, $F(S) \neq F(T)$ if $S \neq T$ from which the result follows. □

Theorem 8.7. *For all non-negative integers $p, p', q, j_1, \ldots, j_{p+p'}, k_1, \ldots, k_q$,*
and all positive integers u, v,
and all (u,v)-separable shapes $((P_1, \ldots, P_{p+p'}), (Q_1, \ldots, Q_q))$,
and all integers $m \geq 2(u+v)(j_1 + \ldots + j_{p+p'}) + 2u$,
every regular expression equivalent to

(in m (and (precisely j_1 P_1)...(precisely j_p P_p)
(noless j_{p+1} P_{p+1})...(noless $j_{p+p'}$ $P_{p+p'}$)
(nomore k_1 Q_1)...(nomore k_q Q_q)))

has a primitive symbol that has at least $\frac{m}{2u} \cdot ((1+j_1)\ldots(1+j_{p+p'}))$ occurrences.

Proof: Let P_0 be the blurry shape in the statement of the theorem.
Let r be a regular expression that is equivalent to P_0.
Let $s_0, s_1, \ldots, s_{p+p'}$ be the strings (in Definition 8.2) that (u,v)-separate the shapes.
Let $a_0 = s_0$ and $a_d = s_d s_0$ for $d = 1, \ldots, p+p'$.
Let $m_0 = L(s_0) + j_1 L(a_1) + \ldots + j_{p+p'} L(a_{p+p'})$.
Note that $m_0 \leq u + j_1(u+v) + \ldots + j_{p+p'}(u+v) \leq \frac{m}{2}$.
Let $j_0 = \lfloor \frac{m - m_0}{L(s_0)} \rfloor$.
Let $t_0 = m \Leftrightarrow m_0 \Leftrightarrow j_0 L(s_0)$. Clearly, $0 \leq t_0 < L(s_0)$. Let t be the first t_0 characters in s_0.

Let $\mathcal{V} = \{0, \ldots, j_0\} \times \ldots \times \{0, \ldots, j_{p+p'}\}$. Hence every element of $\mathcal{V}$ is a vector of $1 + p + p'$ integers. Addition on $\mathcal{V}$ is defined as componentwise addition. For each $x \in \mathcal{V}$, we use $\tilde{x}$ to denote the complement of x under component addition. That is, $x + \tilde{x} = (j_0, \ldots, j_{p+p'})$.

For each $x \in \mathcal{V}$, let $S(x)$ be the string $a_0^{x_0} \ldots a_{p+p'}^{x_{p+p'}}$ where s^d represents d copies of the string s. It is clear that $L(S(x)) = x_0 L(a_0) + \ldots + x_{p+p'} L(a_{p+p'})$.

Our next goal is to show for all $x, y \in \mathcal{V}$ that $N_{P_d}(s_0 S(x) S(y) t) = x_d + y_d$ for $d = 1, \ldots, p+p'$ and that $N_{Q_d}(s_0 S(x) S(y) t) = 0$ for $d = 1, \ldots, q$. For the moment, let R be any of the relevant shapes (i.e. one of the shapes $P_1, \ldots, P_{p+p'}, Q_1, \ldots, Q_q$). All strings a_d have the form $\sigma_d s_0$ for some string σ_d. Hence, for any string s' and $d = 0, \ldots, p+p'$, we can use the fact that s_0 is a divider for R to see that $N_R(s_0 a_d s') = N_R(s_0 \sigma_d s_0 s') = N_R(s_0 \sigma_d s_0) + N_R(s_0 s') = N_R(s_0 a_d) + N_R(s_0 s')$. This fact can be used inductively to show that $N_R(s_0 S(x) S(y) t) = (x_0 + y_0) N_R(s_0 a_0) + \ldots + (x_{p+p'} + y_{p+p'}) N_R(s_0 a_{p+p'}) + N_R(s_0 t)$. By property 2 of dividers and since a_0 is a prefix of s_0, $N_R(s_0 a_0) = N_R(s_0 t) = 0$. By property 2 of separability, it follows that $N_{P_{d'}}(s_0 a_d) = N_{P_{d'}}(s_0 s_d s_0) = \delta(d', d)$ and $N_{Q_{d'}}(s_0 a_d) = N_{Q_{d'}}(s_0 s_d s_0) = 0$. Putting these facts together shows that $N_{P_d}(s_0 S(x) S(y) t) = x_d + y_d$ for $d = 1, \ldots, p+p'$ and that $N_{Q_d}(s_0 S(x) S(y) t) = 0$ for $d = 1, \ldots, q$.

These facts make it clear that $s_0S(x)S(\tilde{x})t$ satisfies all the `precisely`, `noless`, `nomore` conditions necessary to satisfy P_0 so it suffices to check that $s_0S(x)S(\tilde{x})t$ has the right length.

$$\begin{aligned}
& L(s_0S(x)S(\tilde{x})t) \\
= \; & L(s_0) + (x_0L(a_0) + \ldots + x_{p+p'}L(a_{p+p'})) + (\tilde{x}_0L(a_0) + \ldots + \tilde{x}_{p+p'}L(a_{p+p'})) + L(t) \\
= \; & L(s_0) + (j_0L(a_0) + j_1L(a_1) + \ldots + j_{p+p'}L(a_{p+p'})) + L(t) \\
= \; & j_0L(s_0) + (L(s_0) + j_1L(a_0) + \ldots + j_{p+p'}L(a_{p+p'})) + t_0 \\
= \; & j_0L(a_0) + m_0 + (m \Leftrightarrow m_0 \Leftrightarrow j_0L(s_0)) \\
= \; & m.
\end{aligned}$$

Therefore, $s_0S(x)S(\tilde{x})t$ matches r.

Let c_0 be the last character of s_0. We can define a function F that for each $x \in \mathcal{V}$ produces an occurrence of c_0 in r by defining $F(x) = \eta_r(s_0S(x)!S(\tilde{x})t)$ where η is the function given in Lemma 8.5. By part 2 of Lemma 8.5, $F(v)$ is an occurrence (in r) of c_0. Our next goal is to show that F is injective (i.e. one-to-one). Assume that $F(x) = F(y)$ with the goal of showing that $x = y$. Hence, $\eta_r(s_0S(x)!S(\tilde{x})t) = F(x) = F(y) = \eta_r(s_0S(y)!S(\tilde{y})t)$. By part 3 of Lemma 8.5, $s_0S(x)S(\tilde{y})t$ matches r. Therefore, for $d = 1, \ldots, p+p'$, we have that $y_d + \tilde{y}_d = j_d \leq N_{P_d}(s_0S(x)S(\tilde{y})t) = x_d + \tilde{y}_d$ and so $y_d \leq x_d$. A symmetric argument shows that $x_d \leq y_d$. Thus, for $d = 1, \ldots, p+p'$, we have that $x_d = y_d$. By length restrictions, $L(s_0) + L(S(y)) + L(S(\tilde{y})) + L(t) = L(s_0S(y)S(\tilde{y})t) = m = L(s_0S(x)S(\tilde{y})t) = L(s_0) + L(S(x)) + L(S(\tilde{y})) + L(t)$; from which it follows that $L(S(y)) = L(S(x))$. Since $x_d = y_d$ for $d = 1, \ldots, p+p'$ and $L(S(x)) = L(S(y))$, it follows that $x_0 = y_0$ and hence that $x = y$. Therefore, r has at least as many occurrences of c_0 as there are elements in the domain of F. But $\mathcal{V}$ is the domain of F and clearly $\mathcal{V}$ has exactly $(1+j_0)\cdot((1+j_1)\cdot\ldots\cdot(1+j_{p+p'}))$ elements. We can bound $1+j_0$ as follows: $1+j_0 = 1 + \lfloor\frac{m-m_0}{L(s_0)}\rfloor \geq \lceil\frac{m-m_0}{L(s_0)}\rceil \geq \frac{m-m_0}{L(s_0)} \geq \frac{m/2}{u} = \frac{m}{2u}$ from which the result follows. □

8.4. Some Exponential Upper Bounds

The proof that regular expressions are exponentially bigger than the corresponding blurry shapes (Theorem 8.3) is based on the proof that permutation expressions require exponential size to express using regular expressions (Theorem 8.6). Even this "exponential blowup" result slightly understates the relative advantage of blurry shapes since the straightforward permutation expression requires factorial size in regular expression notation. It is only through a certain amount of cleverness (with a corresponding potential loss of clarity) that the permutation regular expression can be reduced to exponential size (Theorem 8.8).

Theorem 8.8. *For every $n \geq 1$, there is a permutation (on n characters) regular expression of size at most $8 \cdot 4^n$.*

Proof: This is proven by induction on n. Let $f(n)$ be the size of the smallest permutation expression on n characters. We prove by induction on n that $f(n) \leq 8 \cdot 4^n$. For $n = 1$, the result is clear since the expression a_1 is a 1-permutation expression and has size 1. Therefore, $f(1) = 1 \leq 32 = 8 \cdot 4^1 = 8 \cdot 4^n$.

For the inductive step, we use the well-known divide and conquer technique by dividing n in half and writing permutation expressions for each half. Assume that $m < n$. Use $P(b_1, \ldots, b_m)$ to denote a permutation expression on the characters $b_1, \ldots, b_m$. A permutation expression for

$a_1, \ldots, a_n$ can be obtained as
$(\texttt{concat}\ (P(\{a_1, \ldots, a_m\}))\ (P(\{a_{m+1}, \ldots, a_n\})))$
$\quad | \ldots |\ (\texttt{concat}\ (P(\{a_{n-m+1}, \ldots, a_n\}))\ (P(\{a_1, \ldots, a_{n-m}\})))$
where each m element subset and its $n \Leftrightarrow m$ element complement participates as a `concat` pair. Let's count the symbols in this expression. Since it is sensible to be lavish when counting in an upper bound proof, we will include parentheses in the counting. Each `concat` subexpression has the form (`concat` (...)(...)) and has size $7 + f(m) + f(n \Leftrightarrow m)$. Thus, ignoring for the moment the "|" symbols, the size of the expression is $(f(m) + f(n \Leftrightarrow m) + 7) \cdot \binom{n}{m}$. There are $\binom{n}{m} \Leftrightarrow 1$ occurrences of the symbol "|". Counting this as 3 symbols (to lavishly include an extra set of parentheses), the "|" symbols add at most $3 \cdot \binom{n}{m}$ to the size of the permutation expression. This shows that $f(n) \leq (f(m) + f(n \Leftrightarrow m) + 10) \cdot \binom{n}{m}$ provided that $0 < m < n$. For $n = 2$, the calculation proceeds as follows: $f(n) = f(2) \leq (f(1) + f(1) + 10) \cdot \binom{2}{1} = 12 \cdot 2 = 24 \leq 8 \cdot 16 = 8 \cdot 4^n$.

Before continuing with the inductive calculation, it is helpful to recall some basic combinatoric facts. It is easy to see by induction that, for $m \geq 2$, $\binom{2m}{m} \leq \frac{3}{2} \cdot 4^{m-1}$ since $\binom{4}{2} = \frac{3}{2} \cdot 4^1$ and $\binom{2(m+1)}{(m+1)} = \binom{2m+2}{m+1} = (\frac{(2m+2)(2m+1)}{(m+1)(m+1)}) \cdot \binom{2m}{m} \leq (2 \cdot 2) \cdot (\frac{3}{2} \cdot 4^{m-1}) = \frac{3}{2} \cdot 4^{(m+1)-1}$. It is also easy to by induction that, for $m \geq 1$, $\binom{2m+1}{m} \leq 3 \cdot 4^{m-1}$.

The inductive calculation of $f(n)$ for $n \geq 3$ breaks down into two cases depending on whether n is even or odd.
If $n = 2m$, then $2 \leq m < n$ and

$$\begin{aligned}
f(n) &\leq (f(m) + f(n \Leftrightarrow m) + 10) \cdot \tbinom{n}{m} \\
&= (f(m) + f(m) + 10) \cdot \tbinom{2m}{m} \\
&\leq (8 \cdot 4^m + 8 \cdot 4^m + \tfrac{10}{128} \cdot 8 \cdot 4^m) \cdot (\tfrac{3}{2} \cdot 4^{m-1}) \\
&= 8 \cdot (2 + \tfrac{10}{128}) \cdot 4^m \cdot (\tfrac{3}{2} \cdot 4^{m-1}) \\
&= 8 \cdot (3 + \tfrac{15}{128}) \cdot 4^{2m-1} \\
&\leq 8 \cdot (4) \cdot 4^{2m-1} \\
&= 8 \cdot 4^{2m} \\
&= 8 \cdot 4^n
\end{aligned}$$

If $n = 2m + 1$, then $1 \leq m < n$ and

$$\begin{aligned}
f(n) &\leq (f(m) + f(n \Leftrightarrow m) + 10) \cdot \tbinom{n}{m} \\
&= (f(m) + f(m+1) + 10) \cdot \tbinom{2m+1}{m} \\
&\leq (8 \cdot 4^m + 8 \cdot 4^{m+1} + \tfrac{10}{32} \cdot 8 \cdot 4^m) \cdot (3 \cdot 4^{m-1}) \\
&= 8 \cdot (1 + 4 + \tfrac{10}{32}) \cdot 4^m \cdot (3 \cdot 4^{m-1}) \\
&= 8 \cdot (15 + \tfrac{30}{32}) \cdot 4^{2m-1} \\
&\leq 8 \cdot (4^2) \cdot 4^{2m-1} \\
&= 8 \cdot 4^{2m+1} \\
&= 8 \cdot 4^n
\end{aligned}$$

In either case, $f(n) \leq 8 \cdot 4^n$ as desired. □

Theorem 8.9 shows that, for every possible number of conjuncts, a situation can be found in which the size of the regular expression equivalent to a blurry query is at most polynomially bigger than the exponential size required by Theorem 8.3. While this does not preclude better lower bounds if the alphabet is held fixed or m is allowed to grow without limits, it does suggest that the exponential lower bound given in Theorem 8.3 is not overly conservative.

Theorem 8.9. *For every positive integer p, there exists positive integers $u, v, j_1, \ldots, j_p$, an alphabet $\mathcal{A}$, (u,v)-separable shapes $((P_1, \ldots, P_p), ())$, a positive integer $m \geq 2(u+v)(j_1 + \ldots + j_p) + 2u$, and a regular expression r of size at most $[(\frac{m}{2u}) \cdot 2^p]^8$ that is equivalent to* (`in` m (`and` (`precisely` j_1 P_1)...(`precisely` j_p P_p))).

Proof: Let $u = v = j_1 = \ldots = j_p = 1$. Let $\mathcal{A}$ consist of $p+1$ disjoint elementary shapes $a_0, \ldots, a_p$. Let the shape P_i be a_i for $i = 1, \ldots, p$. Then the strings $a_0, a_1, \ldots, a_p$ separate $((P_1, \ldots, P_p), ())$. Since every string a_i has length 1, the strings $(1,1)$-separate $(P_1, \ldots, P_p)$. Let $m = 4p + 2$. By Theorem 8.8, we can find a regular expression r' of size at most $8 \cdot 4^m$ that is a permutation expression of the characters $\{a_0^1, \ldots, a_0^{m-p}, a_1, \ldots a_p\}$. Let r be the regular expression that results from replacing (in r') each occurrence of a_0^i by a_0. It is clear that r also has size $8 \cdot 4^m$. It is easy to see that r is equivalent to the shape $P_0' =$ (`in` m (`and` (`precisely` 1 a_1)...(`precisely` 1 a_n))) since any match of P_0' must have exactly one occurrence of each a_i for $i \geq 1$ and exactly $m \Leftrightarrow p$ occurences of a_0 (the only remaining primitive symbol) to fill the string out to length exactly m. Since $P_i = a_i$ and $j_1 = \ldots = j_p = 1$, it is easy to see that P_0' (and hence r) is equivalent to the shape (`in` m (`and` (`precisely` j_1 P_1)...(`precisely` j_p P_p))). We can obtain the desired upper bound to the size of r as follows: $8 \cdot 4^m = 8 \cdot 4^{4p+2} = 8 \cdot 16 \cdot 4^{4p} = 2^7 \cdot 2^{8p} \leq 3^8 \cdot 2^{8p} = [3 \cdot 2^p]^8$ $= [(\frac{4 \cdot 1 + 2}{2 \cdot 1}) \cdot 2^p]^8 \leq [(\frac{4p+2}{2u}) \cdot 2^p]^8 = [(\frac{m}{2u}) \cdot 2^p]^8$. □

8.5. Regular Expressions with Exponentiation

It is well-known that exponentiation does not increase the expressive power of regular expressions but does allow some compaction. *Regular expressions augmented with exponentiation* permits expressions of the form r^k for some exponent k. The meaning of r^k is $r \ldots r$ where there are k copies of r. For example, a^{249} is much more compact than 249 copies of a concatenated together. It turns out that even augmenting regular expressions with exponentiation does not stop the exponential growth of regular expressions equivalent to a blurry shape! Before proving this in Theorem 8.11, it is helpful to begin with a lemma.

Lemma 8.10. *If r' is a regular expression augmented with exponentiation such that every string that matches r' has length at most m, then there is a regular expression r that is equivalent to r' and has at most m times as many primitive symbols as r'.*

Proof: Form r from r' by replacing each occurrence of r_0^i by (`concat` $r_0 \ldots r_0$) where there are i occurences of r_0 in the `concat` construct. Notice that if r_0 matches at least one string of length k, then both versions of r_0^i match at least one string of length $i \cdot k$. Also note that this expansion creates i copies of every primitive symbol within r_0. It follows that if there are i copies of a primitive symbol in the final result, then there is a string of length at least i. Hence, after the above expansion process, no occurrence of a primitive symbol in r' can be expanded to more than m copies so the result follows. □

Theorem 8.11. *For all non-negative integers $p, p', q, k_1, \ldots, k_q$,*
and all positive integers $u, v, j_1, \ldots, j_{p+p'}$,
and all (u,v)-separable shapes $((P_1, \ldots, P_{p+p'}), (Q_1, \ldots, Q_q))$,
and all integers $m \geq 2(u+v)(j_1 + \ldots + j_{p+p'}) + 2u$,

every regular expression with exponentiation equivalent to

```
(in m (and   (precisely j_1 P_1)...(precisely j_p P_p)
             (noless j_{p+1} P_{p+1})...(noless j_p P_{p+p'})
             (nomore k_1 Q_1)...(nomore k_q Q_q)))
```

has at least $\frac{1}{2u} \cdot 2^{p+p'}$ *occurrences of primitive symbols.*

Proof: Suppose not. Let P_0 be the shape

```
(in m (and   (precisely j_1 P_1)...(precisely j_p P_p)
             (noless j_{p+1} P_{p+1})...(noless j_p P_{p+p'})
             (nomore k_1 Q_1)...(nomore k_q Q_q)))
```

and let r' be a regular expression augmented with exponentiation equivalent to P_0 that has size strictly less than $(\frac{1}{2u}) \cdot 2^{p+p'}$ primitive symbols. Since every match to r' has length exactly m, Lemma 8.10 applies. Hence there is a regular expression r equivalent to r' that has strictly less than $m \cdot (\frac{1}{2u}) \cdot 2^{p+p'}$ primitive symbols. Thus, r is a regular expression equivalent to P_0 with strictly less than $(\frac{m}{2u}) \cdot 2^{p+p'}$ primitive symbols. This contradicts Theorem 8.3. Hence, the result follows. □

9. Appendix C

In this appendix, a formal comparison is made between the expressive power of regular expressions and the expressive power of $\mathcal{SDL}$ with regard to regular matching. As discussed in Section 5.2, it suffices to consider the case where all the elementary shapes are disjoint. Call this set of disjoint elementary shapes $\mathcal{A}$.

The next theorem shows that the $\mathcal{SDL}$ language is at least expressive as regular expressions for regular matching. Since $\mathcal{SDL}$ has `any`, `concat`, and `atleast`, this is not a surprising result. The main difficulty comes about since the `atleast` operator has different semantics in $\mathcal{SDL}$ than it does for regular expressions. The proof that the set of strings accepted by a deterministic finite automaton can be described by a regular expression is adapted in Theorem 9.1 to prove a similar result for shapes. This shape is constructed in such a way so as to ensure that the `atleast` operator is used only in places where its greedy $\mathcal{SDL}$ semantics coincide with the standard non-greedy regular expression semantics.

Theorem 9.1. *For every regular expression r, there exists a shape s in $\mathcal{SDL}$ such that a history matches r iff it matches s.*

Proof: Given a regular expression r, let D be a deterministic finite automaton that matches the same set of strings as r does. Let $\mathbf{O}^a_b$ = `(any` $s_1 \ldots s_m$ `)` where $s_1, \ldots, s_m$ are the primitive symbols that allow a transition from state a to state b. If $U = \{a, b_1, \ldots, b_m\}$ is a set of states, let $U' = \{b_1, \ldots, b_m\}$ and define by induction on m, the shapes $\mathbf{l}^a(U)$ and $\mathbf{L}^a(U)$ as follows:
$\mathbf{l}^a(U)$ = `(any` $\mathbf{O}^a_a$ `(concat` $\mathbf{O}^a_{b_1}$ $\mathbf{L}^{b_1}(U')$ $\mathbf{O}^{b_1}_a$ `)` ... `(concat` $\mathbf{O}^a_{b_m}$ $\mathbf{L}^{b_m}(U')$ $\mathbf{O}^{b_m}_a$ `) )`
and $\mathbf{L}^a(U)$ = `(atleast 0` $\mathbf{l}^a(U)$ `)`.
It is easy to see that $\mathbf{l}^a(U)$ matches exactly those non-empty strings that, when starting from state a, terminate in state a while visiting only the states in $U \Leftrightarrow \{a\}$ during intermediate stops. It is also easy to see that $\mathbf{L}^a(U)$ matches exactly those strings that, when starting from state a, terminate in state a while visiting only the states in U. Notice that the only place that `atleast` occurs is

in $\mathbf{L}^a(U)$. Hence the first symbol of a non-empty string that matches $\mathbf{L}^a(U)$ must be a primitive symbol that transit from state a to some state in U.

Denote by B the set of all states. Given a sequence of states $(b_1, \ldots, b_m)$ where no state is repeated, define the shape
$\mathbf{P}(b_1, \ldots, b_m) =$ `(concat` $\mathbf{L}^{b_1}(\{b_1\})\ \mathbf{O}^{b_1}_{b_2}\ \ldots\ \mathbf{L}^{b_{m-1}}(\{b_1, \ldots, b_{m-1}\})\ \mathbf{O}^{b_{m-1}}_{b_m}\ \mathbf{L}^{b_m}(B)$ `)`.
Notice that since $\mathbf{O}^{b_i}_{b_{i+1}}$ represents a transition from state b_i to a state outside of $\{b_1, \ldots, b_i\}$, any symbol that matches the shape $\mathbf{O}^{b_i}_{b_{i+1}}$ is not a prefix of any match of $\mathbf{L}^{b_i}(\{b_1, \ldots, b_i\})$. Therefore, $\mathbf{P}(b_1, \ldots, b_m)$ matches exactly those strings that when starting in state b_1, the ith state visited (for $i \leq m$) is b_i; since a state might be visited several times, only the first visit counts; after state b_m is reached any states may be visited provided the string terminates in state b_m.

Finally, let $E =$ `(any` $\ldots\ \mathbf{P}(b_1, \ldots, b_m)\ \ldots$ `)` where m is less than or equal to the number of states in the D, b_1 is the start state, and b_m is some accepting state. It is clear that E is a shape that matches exactly those strings accepted by D. □

A possible change to $\mathcal{SDL}$ is to restrict `any` to match only one of the alternatives (say the first listed that matches). The above construction shows that this change does not lose any expressive power since in the two uses of `any` in the proof of Theorem 9.1 (in the definition of $\mathbf{O}^a_b$ and E), no two different alternatives in the `any` clause share a string in their respective match sets.

Even though the general construction given in the proof of Theorem 9.1 involves an explosion in the size of a shape, in practice this is rarely a problem. Given a regular expression, it is quite possible that the corresponding $\mathcal{SDL}$ shape has the same meaning and in this case, there is no explosion at all. Furthermore, even if the regular expression and the corresponding shape do not have the same semantics, there is frequently a simple transformation that produces a shape that has the same meaning as the regular expression. For example, consider the shape (`concat` (`atleast` 0 `up`) `up` `down`). In regular expression semantics, this matches all strings of the form (`up`^{n} `down`) for some $n \geq 1$. However, in shape semantics, the shape never matches any string since every `up` is gobbled up by the `atleast` expression. A simple solution is to move the `up` into the `atleast` clause giving the shape `(concat (atleast` 1 `up ) down )` which has the same meaning as the corresponding regular expression. Slightly complicated versions of such transformations are frequently possible and so the explosion in transforming a regular expression to a shape is rarely a problem.

Theorem 9.1 shows that $\mathcal{SDL}$ shapes are at least as expressive as regular expressions. Conversely, Theorem 9.4 shows that regular expressions are at least as expressive as $\mathcal{SDL}$ shapes. Before proving Theorem 9.4, it is helpful to review the proof of one the most basic and well-known ([8]) theorems about regular expressions, namely that every regular expression has a DFA that accepts exactly the set of strings matched by the regular expression. Since the proof of Theorem 9.4 is different in style than the standard proof, it is a good idea to reprove the basic theorem (with which everyone is familiar) as a warmup exercise. The standard proof of Theorem 9.3 may be found in a textbook covering automaton such as [8]. Before giving our version of the proof, it is helpful to give an example.

Example 9.2. Given the regular expression $r =$ (`concat` (`star` a) (`any` a (`concat` b c))) where a, b, c are members of the alphabet $\mathcal{A}$, we construct a DFA. Every state of the DFA will be a finite set of occurences of primitive symbols. It is helpful to have a new primitive symbol m

that serves as an endmarker. An accepting state one is one that contains the endmarker m. Let r' = (concat (concat (star a_1) (any a_2 (concat b c))) m) where subscripts have been put on the two occurences of the primitive symbol a to allow us to distinguish them in this discussion. The basic idea of a state is that the occurences of primitive symbols that comprise a state are the "next" symbol that could be scanned. After the symbol is scanned, rewriting is performed to find the next possible symbols. Rewriting is performed by moving a pointer indicated with the notation "!". The precise rules for moving the pointer are given in the proof. The initial phase starts with the pointer at the beginning of r' and it is moved until it encounters a primitive symbol. Here is an example of that rewriting process:

!(concat (concat (star a_1) (any a_2 (concat b c))) m)
$\Rightarrow$ (concat !(concat (star a_1) (any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat !(star a_1) (any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat (star !a_1) (any a_2 (concat b c))) m)

So a_1 is in the initial state.

!(concat (concat (star a_1) (any a_2 (concat b c))) m)
$\Rightarrow$ (concat !(concat (star a_1) (any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat !(star a_1) (any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat (star a_1)! (any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat (star a_1) !(any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat (star a_1) (any !a_2 (concat b c))) m)

So a_2 is in the initial state.

!(concat (concat (star a_1) (any a_2 (concat b c))) m)
$\Rightarrow$ (concat !(concat (star a_1) (any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat !(star a_1) (any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat (star a_1)! (any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat (star a_1) !(any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat (star a_1) (any a_2 !(concat b c))) m)
$\Rightarrow$ (concat (concat (star a_1) (any a_2 (concat !b c))) m)

So b is in the initial state.

The initial state is $\{a_1, a_2, b\}$. Consider running the DFA on the string $abcd$. Since the endmarker m is not in the initial state, the empty string is not accepted by the DFA. Next we scan the symbol a by moving the pointer past all the occurences of a that were in the previous state and advance those pointer until new symbols are encountered as follows.

First we move the pointer past a_2.

(concat (concat (star a_1) (any a_2! (concat b c))) m)
$\Rightarrow$ (concat (concat (star a_1) (any a_2 (concat b c))!) m)
$\Rightarrow$ (concat (concat (star a_1) (any a_2 (concat b c)))! m)
$\Rightarrow$ (concat (concat (star a_1) (any a_2 (concat b c))) !m)

So m is the next state.

Next we move the pointer past a_1.

(concat (concat (star a_1!) (any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat !(star a_1) (any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat (star !a_1) (any a_2 (concat b c))) m)

So a_1 is the next state.

(concat (concat (star a_1!) (any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat !(star a_1) (any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat (star a_1)! (any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat (star a_1) !(any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat (star a_1) (any !a_2 (concat b c))) m)
So a_2 is in the next state.
(concat (concat (star a_1!) (any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat !(star a_1) (any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat (star a_1)! (any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat (star a_1) !(any a_2 (concat b c))) m)
$\Rightarrow$ (concat (concat (star a_1) (any a_2 !(concat b c))) m)
$\Rightarrow$ (concat (concat (star a_1) (any a_2 (concat !b c))) m)
So b is in the next state.

The next state consists of $\{m, a_1, a_2, b\}$. Since the endmarker m is in the state, the string a is accepted by the DFA.

Next we scan the symbol b in the string $abcd$ by moving the pointer past the b and continuing.
(concat (concat (star a_1) (any a_2 (concat b! c))) m)
$\Rightarrow$ (concat (concat (star a_1) (any a_2 (concat b !c))) m)
So c is in the next state. The next state is $\{c\}$. Since the endmarker is not in this set, the string ab is not accepted by the DFA.

Next we scan the symbol c in $abcd$ by moving the pointer past the c and continuing.
(concat (concat (star a_1) (any a_2 (concat b c!))) m)
$\Rightarrow$ (concat (concat (star a_1) (any a_2 (concat b c)!)) m)
$\Rightarrow$ (concat (concat (star a_1) (any a_2 (concat b c))!) m)
$\Rightarrow$ (concat (concat (star a_1) (any a_2 (concat b c)))! m)
$\Rightarrow$ (concat (concat (star a_1) (any a_2 (concat b c))) !m)
So m is in the next state. The next state is $\{m\}$. Since m is in this state, the DFA accepts the string abc.

Finally, the DFA scans the symbol d in the string $abcd$ but since there are no occurences of d in the previous state, the next state is empty. Hence, the string $abcd$ is not accepted by the DFA. (Furthermore, any extension of $abcd$ will not be accepted by the DFA since once the state becomes empty it remains empty forever.) The following table summarizes the process for the DFA for the regular expression (concat (concat (star a_1) (any a_2 (concat b c))) m) with endmarker m when acting upon the string $abcd$ and its prefixes.

String Scanned (so far)	DFA state	Comments
ϵ	$\{a_1, a_2, b\}$	(initial state) ϵ rejected.
a	$\{a_1, a_2, b, m\}$	a accepted.
ab	$\{c\}$	ab rejected.
abc	$\{m\}$	abc accepted.
$abcd$	$\{\}$	$abcd$ rejected.

Theorem 9.3 (Well-Known Theorem). *Every regular expression r has a deterministic finite automaton that accepts exactly the set of strings that match the regular expression r.*

Proof:

The first step is to review the definition of matching for regular expressions. This can be defined in terms of a string and a stack of (marked) regular expressions. The purpose of the marking is to prevent expansion of the `star` operator without consuming at least one symbol thereby preventing infinite loops. The stack is indicated using `stack` and a mark is indicated by T. Unmarking everything in the stack is denoted by U. The semantics is as follows where M_0 maps (string,stack) pairs into boolean values. It turns out that it is convenient to introduce a new primitive symbol m_0 as an endmarker.

1. $M_0(s, (\texttt{stack})) = False$
2. $M_0(\epsilon, (\texttt{stack}\ m_0\ r_2 \ldots r_n)) = True$
3. $M_0(\epsilon, (\texttt{stack}\ a\ r_2\ \ldots r_n)) = False$ if a is a primitive symbol
4. $M_0(as, (\texttt{stack}\ a\ r_2 \ldots r_n)) = M_0(s, U(\texttt{stack}\ r_2 \ldots r_n))$ if a is a primitive symbol
5. $M_0(as, (\texttt{stack}\ b\ r_2 \ldots r_n)) = False$ if a, b are different primitive symbols
6. $M_0(s, (\texttt{stack}\ (\texttt{any}\ r'_1 \ldots r'_m)\ r_2 \ldots r_n))$
   $= M_0(s, (\texttt{stack}\ r'_1\ r_2 \ldots r_n)) \vee\ \ldots\ \vee M_0(s, (\texttt{stack}\ r'_m\ r_2 \ldots r_n))$
7. $M_0(s, (\texttt{stack}\ (\texttt{concat}\ r'_1 \ldots r'_m)\ r_2 \ldots r_n)) = M_0(s, (\texttt{stack}\ r'_1 \ldots r'_m\ r_2 \ldots r_n))$
8. $M_0(s, (\texttt{stack}\ (\texttt{star}\ r_1)\ r_2 \ldots r_n))$
   $= M_0(s, (\texttt{stack}\ r_1\ T(\texttt{star}\ r_1)\ r_2 \ldots r_n)) \vee M_0(s, (\texttt{stack}\ r_2 \ldots r_n))$
9. $M_0(s, (\texttt{stack}\ T(\texttt{star}\ r_1)\ r_2 \ldots r_n)) = False$

A string s *matches* a regular expression r iff $M_0(s, (\texttt{stack}\ (\texttt{concat}\ r\ m_0))) = True$.

At first glance, this definition might seem to require unbounded space because of the stack. However, each stack can be encoded with a pointer into the regular expression (with markings as needed). Define the function R to reconstruct the stack as follows (where ! represents the pointer) and $(\texttt{push}\ r_1\ (\texttt{stack}\ r_2 \ldots r_n)) = (\texttt{stack}\ r_1\ r_2 \ldots r_n)$ as follows:

1. $R(\ldots !r \ldots) = (\texttt{push}\ r\ R(\ldots r! \ldots))$
2. $R(\ldots (\texttt{any}\ r_1 \ldots r_{i-1}\ r_i!\ r_{i+1} \ldots r_n) \ldots) = R(\ldots (\texttt{any}\ r_1 \ldots r_n)! \ldots)$
3. $R(\ldots (\texttt{concat}\ r_1 \ldots r_{n-1}\ r_n!) \ldots) = R(\ldots (\texttt{concat}\ r_1 \ldots r_{n-1}\ r_n)! \ldots)$
4. $R(\ldots (\texttt{concat}\, r_1 \ldots r_{i-1}\, (r_i!)\ \ r_{i+1} \ldots r_n) \ldots) = R(\ldots (\texttt{concat}\, r_1 \ldots r_{i-1}\, r_i\ \ (!r_{i+1})\, r_{i+2} \ldots r_n) \ldots)$
5. $R(\ldots (\texttt{star}\ r_1!) \ldots) = R(\ldots !(\texttt{star}\ r_1) \ldots)$
6. $R(\ldots T(\texttt{star}\ r_1!) \ldots) = R(\ldots !T(\texttt{star}\ r_1) \ldots)$
7. $R(r!) = (\texttt{stack})$

For the rest of the proof, the notation $(\ldots !r_0 \ldots)$ will be used to denote a regular expression with the pointer just before r_0 and the notation $(\ldots r_0! \ldots)$ will be used to denote a regular expression with the pointer just after r_0. A regular expression with a single pointer (and possibly with "T" markers) will be referred to as a "nicked" regular expression.

Define a new meaning function M_1 that maps a (string,nicked regular expression) into booleans by the definition $M_1(s, r) = M_0(s, R(r))$ where r is the nicked regular expression. It is easy to check that M_1 satisfies the following properties:

1. $M_1(s, (r!)) = False$
2. $M_1(\epsilon, (\ldots !m_0 \ldots)) = True$
3. $M_1(\epsilon, (\ldots !a \ldots)) = False$ if a is a primitive symbol
4. $M_1(as, (\ldots !a \ldots)) = M_1(s, U(\ldots a! \ldots))$ if a is a primitive symbol
5. $M_1(as, (\ldots !b \ldots)) = False$ if a, b are different primitive symbols
6. a.) $M_1(s, (\ldots !(\texttt{any}\ r_1 \ldots r_m) \ldots))$
   $= M_1(s, (\ldots (\texttt{any}\ !r_1\ r_2 \ldots r_m) \ldots)) \vee\ \ldots\ \vee M_1(s, (\ldots (\texttt{any}\ r_1 \ldots r_{m-1}\ !r_m) \ldots))$
   b.) $M_1(s, (\ldots (\texttt{any}\ r_1 \ldots r_{i-1}\ r_i!\ r_{i+1} \ldots r_m) \ldots)) = M_1(s, (\ldots (\texttt{any}\ r_1 \ldots r_m)! \ldots))$
7. a.) $M_1(s, (\ldots !(\texttt{concat}) \ldots)) = M_1(s, (\ldots (\texttt{concat})! \ldots))$
   b.) $M_1(s, (\ldots !(\texttt{concat}\ r_1 \ldots r_m) \ldots)) = M_1(s, (\ldots (\texttt{concat}\ !r_1\ r_2 \ldots r_m) \ldots))$
   c.) $M_1(s, (\ldots (\texttt{concat}\ r_1 \ldots r_{i-1}\ r_i!\ \ r_{i+1} \ldots r_n) \ldots)) =$
   $M_1(s, (\ldots (\texttt{concat}\ r_1 \ldots r_{i-1}\ r_i\ \ !r_{i+1}\ r_{i+2} \ldots r_n) \ldots))$
   d.) $M_1(s, (\ldots (\texttt{concat}\ r_1 \ldots r_{m-1}\ r_m!) \ldots)) = M_1(s, (\ldots (\texttt{concat}\ r_1 \ldots r_m)! \ldots))$
8. a.) $M_1(s, (\ldots !(\texttt{star}\ r_1) \ldots))$
   $= M_1(s, (\ldots T(\texttt{star}\ !r_1) \ldots)) \vee M_1(s, (\ldots (\texttt{star}\ r_1)! \ldots))$
   b.) $M_1(s, (\ldots (\texttt{star}\ r_1!) \ldots)) = M_1(s, (\ldots !(\texttt{star}\ r_1) \ldots))$
9. a.) $M_1(s, (\ldots !T(\texttt{star}\ r_1)\ \ldots)) = False$
   b.) $M_1(s, (\ldots T(\texttt{star}\ r_1!)\ \ldots)) = False$

The above properties of M_1 could be viewed as a definition of matching. A string s matches a regular expression r iff $M_1(s, !(\texttt{concat}\ r\ m_0)) = True$.

Since the real action happens when a primitive symbol is on the top of the stack, one can view the other steps as rewriting (denoted with the $\Rightarrow$ symbol) to get a primitive symbol on the top of the stack. The inductive clauses 6-9 of M_1 could be taken to define a set of rewrite rules on nicked regular expressions. Let $\Rightarrow$ be the least reflexive, transitive relation such that the following all hold:

6. a.) $!(\texttt{any}\ r_1\ \ldots r_n) \Rightarrow (\texttt{any}\ r_1 \ldots r_{i-1}\ !r_i\ r_{i+1} \ldots r_n)$
   b.) $(\texttt{any}\ r_1 \ldots r_{i-1}\ r_i!\ \ r_{i+1} \ldots r_n) \Rightarrow (\texttt{any}\ r_1 \ldots r_n)!$

7. a.) $!$(concat) $\Rightarrow$ (concat)$!$
   b.) $!$(concat $r_1 \ldots r_n$) $\Rightarrow$ (concat $!r_1$ $r_2 \ldots r_n$)
   c.) (concat $r_1 \ldots r_{i-1}$ $r_i!$ $r_{i+1} \ldots r_n$) $\Rightarrow$ (concat $r_1 \ldots r_{i-1}$ r_i $!r_{i+1}$ $r_{i+2} \ldots r_n$)
   d.) (concat $r_1 \ldots r_{n-1}$ $r_n!$) $\Rightarrow$ (concat $r_1 \ldots r_{n-1}$ r_n)$!$

8. a'.) $!$(star r_1) $\Rightarrow T$(star $!r_1$)
   a".) $!$(star r_1) $\Rightarrow$ (star r_1)$!$
   b.) (star $r_1!$) $\Rightarrow!$(star r_1)

9. $!T$(star r_1) and T(star $r_1!$) are stuck (i.e. no $\Rightarrow$ rules apply)

Let $\Rightarrow_a$ be the rule that corresponds to clause 4 in the definition of M_1; namely, $\Rightarrow_a$ transforms $!a$ into $a!$ and erases all the T markers. Define M_2 on (string,nicked regular expression) pairs to produce a boolean as follows: $M_2(\epsilon, r_0) = True$ iff $r_0 \Rightarrow r_1$ for some r_1 of the form $(\ldots !m_0 \ldots)$; $M_2(as, r_0) = True$ iff there exists r_1, r_2 such that $r_0 \Rightarrow r_1 \Rightarrow_a r_2$ and $M(s, r_2) = True$. It is easy to check that $M_2 = M_1$. Hence M_2 represents a valid definition for matching a regular expression.

We will use M_2 to build a DFA. Given a regular expression r, let $O(r)$ be the set of occurences of primitive symbols in r. For each $o \in O(r)$, let $P(o)$ denote the primitive symbol of which o is an occurrence. Define $I(r) = \{o \in O(r) \mid !r \Rightarrow (\ldots !o \ldots)\}$ and for each $o \in O(r)$, define $N_o(r) = \{o' \in O(r) | (\ldots o! \ldots) \Rightarrow (\ldots !o' \ldots)\}$. To build a DFA for the regular expression r, let $r_0 =$ (concat r m_0) where m_0 is the endmarker. Every state of the DFA is a subset of $O(r_0)$. The initial state is $I(r_0)$. An accepting state is a state that has at least one occurrence of m_0. The transition function δ is as follows: $\delta(a, S) = \cup\{N_o(r_0) | o \in S \wedge P(o) = a\}$. It is clear that this DFA behaves just as M_2 does. Therefore, this DFA accepts a string iff the string matches the regular expression r. $\Box$

The next theorem shows that the regular expressions are at least as expressive as $\mathcal{SDL}$ for expressing full sequence queries.

Theorem 9.4. *For every $\mathcal{SDL}$ shape s, there exists a regular expression r such that a history matches r iff it matches s.*

Proof: (sketch)

First note that it is easy to dispense with the `in` construct since one of the arguments to `in` is the length of the string. Hence there are only a finite (but potentially huge) number of possible strings that match the `in` construct. These can be all listed using the `any` construct. The `in` no longer needs to be considered for the rest of the proof.

Before proving the result, it is helpful to reduce the number of constructs that must be dealt with. Add the new shape `nofollows`. The `nofollows` shape is different than the typical regular expression in that it restricts future matching. In particular, (`concat` (`nofollows` P) Q) indicates that, in order to match, not only must Q match but no prefix can match P.

The `atleast`, `atmost`, and `exact` constructs can be eliminated in favor of the new constructs as follows:
(`atleast` n P) = (`concat` P ... P (`star` P) (`nofollows` P))) where the dots represent n repetitions of P
(`exact` n P) = (`concat` P ... P) (`nofollows` P)) where the dots represent n repetitions of P

$(\texttt{atmost}\ n\ P) = (\texttt{concat}\ (\texttt{any}\ (\texttt{concat}) \ldots (\texttt{concat}\ P\ \ldots\ P))\ (\texttt{nofollows}\ P))$ where the dots indicate all sequences of P's of length at most n.

Using the notation in the proof of Theorem 9.3, the match set for `nofollows` is as follows: $M_0(s, (\texttt{stack}\ (\texttt{nofollows}\ r_1)\ r_2 \ldots r_n)) = True$ iff $M_0(s, (\texttt{stack}\ r_2 \ldots r_n)) = True$ and for all prefixes s' of s, $M_0(s', (\texttt{stack}\ r_1)) = False$. Since we are depending on endmarkers, there has to be a new distinct endmarker added for each `nofollows` construct so that $(\texttt{nofollows}\ r_1)$ is replaced by $(\texttt{nofollows}\ (\texttt{concat}\ r_1\ m_0))$ where m_0 is a unique new endmarker.

Just as in the proof of Theorem 9.3, rewriting (using the $\Rightarrow$ notation) is used to get an alternative definition of matching upon which the DFA is constructed. The `nofollows` construct represents a drag on the matching that may occur. Fix a regular expression r_0 that has already been augmented with endmarkers (a unique one for each `nofollows` clause as well as the endmarker for the entire regular expression). Each state must keep track not only of the current DFA but also of the subsidiary DFAs that have been set in motion by a `nofollows` clause. Every potential state S will be a set of the form $\{(o_1, S_1), \ldots, (o_n, S_n)\}$ where each o_i is an occurrence of a primitive symbol and S_i is itself a state. (Note that since a state might be empty, there is indeed a base case in the definition of potential state.) A potential state S is called *homogeneous* iff (a) if $(o_1, S_1) \in S$, then S_1 is homogenous, (b) if $(o_1, S_1) \in S$ and $(o_2, S_2) \in S_1$, then o_2 occurs within at least one more `nofollows` constructs than o_1 occurs in. and (c), if $(o_1, S_1), (o_2, S_2) \in S$, then o_1, o_2 occur within exactly the same set of `nofollows` constructs. Note that the empty set is homogeneous. Also note that there are only a finite number of homogenous potential states since the "depth" of a homogeneous potential state can be no more than the depth of nesting of `nofollows` constructs. A potential state S is *valid* iff o_2 is not an endmarker whenever $(o_2, S_2) \in S_1$ for some $(o_1, S_1) \in S$. The states of the DFA are the valid, homogeneous potential states. As before, there will be a rewriting process $\Rightarrow$ that takes a (regular expression, homogeneous potential state) pair and produces another. (The rewrite rules are given below.) The rules have the property that as a pointer moves, it always stays within exactly the same set of `nofollows` constructs thereby preserving homogeneity. (Note that the notion of homogeneity is extended to include pointers, not just occurences of primitive symbols.) As before, we define the functions I and N_o, but first we need the function V that turns a homogeneous potential state into a state as follows: $V(S) = \{(o_1, V(S_1)) \mid (o_1, S_1) \in S \wedge \forall (o_2, S_2) \in V(S_1)(P(o_2) \text{is not an endmarker})\}$. Note that V works from the inside out, transforming the innermost potential states first. $I(r) = V(\{(o, D) \mid (!r, \emptyset) \Rightarrow (o, D)\})$. If $o \in O(r)$ is an occurrence of a primitive symbol, $N_o(r, D) = \{(o', D') \mid ((\ldots o! \ldots), D) \Rightarrow ((\ldots !o' \ldots, D'))\}$ where $(\ldots o! \ldots)$ represents r with the pointer just after o and $(\ldots o! \ldots)$ represents some other nicked regular expression with the pointer just before o'. The initial state is $I(r_0)$. Any state with an element of the form (m_0, S_0) where m_0 is an endmarker is an accepting state. The transition function is as follows: $\delta(a, S) = V(\{N_{o_1}(r_0, \delta(a, S_1)) \mid (o_1, S_1) \in S \wedge P(o_1) = a\})$. The reader can check that this DFA accepts a string iff the string matches the regular expression where $\Rightarrow$ is the least reflexive, transitive relation such that the following all hold:

1. $(!(\texttt{any}\ r_1\ \ldots r_n), D) \Rightarrow ((\texttt{any}\ r_1 \ldots r_{i-1}\ !r_i\ r_{i+1} \ldots r_n), D)$
2. $((\texttt{any}\ r_1 \ldots r_{i-1}\ r_i!\ \ r_{i+1} \ldots r_n), D) \Rightarrow ((\texttt{any}\ r_1 \ldots r_n)!, D)$
3. $(!(\texttt{concat}), D) \Rightarrow ((\texttt{concat})!, D)$
4. $(!(\texttt{concat}\ r_1 \ldots r_n), D) \Rightarrow ((\texttt{concat}\ !r_1\ r_2 \ldots r_n), D)$

5. $((\texttt{concat}\ r_1 \ldots r_{i-1}\ (r_i!)\ \ r_{i+1} \ldots r_n), D) \Rightarrow ((\texttt{concat}\ r_1 \ldots r_{i-1}\ r_i\ \ (!r_{i+1})\ r_{i+2} \ldots r_n), D)$
6. $((\texttt{concat}\ r_1 \ldots r_{n-1}\ r_n!), D) \Rightarrow ((\texttt{concat}\ r_1 \ldots r_{n-1}\ r_n)!, D)$
7. $((\texttt{star}\ r_1!), D) \Rightarrow (!(\texttt{star}\ r_1), D)$
8. $(!(\texttt{star}\ r_1), D) \Rightarrow (T(\texttt{star}\ !r_1), D)$
9. $(!(\texttt{star}\ r_1), D) \Rightarrow ((\texttt{star}\ r_1)!, D)$
10. $(!T(\texttt{star}\ r_1), D)$ is stuck (i.e. no $\Rightarrow$ rules apply)
11. $(!(\texttt{nofollows}\ r_1), D) \Rightarrow ((\texttt{nofollows}\ r_1)!, D \cup I(r_1))$

□

Remark 9.5. The reader might find it unsatisfying to see that the `in` construct was handled in one fell swoop by a finiteness argument. This is indeed the easiest proof but sheds little light on the `in` construct itself. In fact, the `in` construct could be handled by using a counter to keep track of how many different substrings have entered that state. A counter in the start state would be incremented every time a symbol was scanned since the newly scanned symbol represents another starting point for a subsequence. Of course, DFAs cannot handle counting since that involves a potentially unbounded memory. However, DFAs can handle bounded counting where once the counter hits a maximum value further incrementing has no effect. Since every `precisely`, `nomore`, `noless` has an integer argument, it is sufficient if all counters are bounded by one plus the maximum of all these integer arguments. The complexity of this construction (which must be folded in with added complexity already captured in the proof of Theorem 9.4) underscores the sophistication of blurry matching!

References.

[1] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993. Special Issue on Learning and Discovery in Knowledge-Based Databases.

[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] A. Califano and I. Rigoutsos. FLASH: A fast look-up algorithm for string homology. In *Proc. of the 1st Int'l Conference on Intelligent Systems for Molecular Biology*, pages 353–359, Bethesda, MD, July 1993.

[4] 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.

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

[6] 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.

[7] 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.

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

[9] M. Roytberg. A search for common patterns in many sequences. *Computer Applications in the Biosciences*, 8(1):57–64, 1992.

[10] 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.

[11] M. Vingron and P. Argos. A fast and sensitive multiple sequence alignment algorithm. *Computer Applications in the Biosciences*, 5:115–122, 1989.

[12] J. T.-L. Wang, G.-W. Chirn, T. G. Marr, B. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Minneapolis, May 1994.

[13] S. M. Weiss and C. A. Kulikowski. *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems.* Morgan Kaufman, 1991.

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