Symbolic Trajectories

RALF HARTMUT GÜTING and FABIO VALDÉS, FernUniversität Hagen
MARIA LUISA DAMIANI, University of Milan

Due to the proliferation of GPS-enabled devices in vehicles or with people, large amounts of position data are recorded every day and the management of such mobility data, also called trajectories, is a very active research field. A lot of effort has gone into discovering "semantics" from the raw geometric trajectories by relating them to the spatial environment or finding patterns, for example, by data mining techniques. A question is how the resulting "meaningful" trajectories can be represented or further queried.

In this article, we propose a systematic study of *annotated trajectory databases*. We define a very simple generic model called *symbolic trajectory* to capture a wide range of meanings derived from a geometric trajectory. Essentially, a symbolic trajectory is just a time-dependent label; variants have sets of labels, places, or sets of places. They are modeled as abstract data types and integrated into a well-established framework of data types and operations for moving objects. Symbolic trajectories can represent, for example, the names of roads traversed obtained by map matching, transportation modes, speed profile, cells of a cellular network, behaviors of animals, cinemas within 2km distance, and so forth. Symbolic trajectories can be combined with geometric trajectories to obtain annotated trajectories.

Besides the model, the main technical contribution of the article is a language for pattern matching and rewriting of symbolic trajectories. A symbolic trajectory can be represented as a sequence of pairs (called units) consisting of a time interval and a label. A pattern consists of unit patterns (specifications for time interval and/or label) and wildcards, matching units and sequences of units, respectively, and regular expressions over such elements. It may further contain variables that can be used in conditions and in rewriting. Conditions and expressions in rewriting may use arbitrary operations available for querying in the host DBMS environment, which makes the language extensible and quite powerful.

We formally define the data model and syntax and semantics of the pattern language. Query operations are offered to integrate pattern matching, rewriting, and classification of symbolic trajectories into a DBMS querying environment. Implementation of the model using finite state machines is described in detail. An experimental evaluation demonstrates the efficiency of the implementation. In particular, it shows dramatic improvements in storage space and response time in a comparison of symbolic and geometric trajectories for some simple queries that can be executed on both symbolic and raw trajectories.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Logical Design, Languages—*Data models, Query languages*; H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*

General Terms: Design, Languages, Algorithms, Performance

Additional Key Words and Phrases: Trajectories, moving objects, semantic trajectories, pattern matching, rewriting

---

This work was partially supported by the EU COST Action IC0903 "Knowledge Discovery from Moving Objects."
Authors' addresses: R. H. Güting, Databases for New Applications, FernUniversität Hagen, 58084 Hagen, Germany; email: rhg@fernuni-hagen.de; F. Valdés, Databases for New Applications, FernUniversität Hagen, 58084 Hagen, Germany; email: fabio.valdes@fernuni-hagen.de; M. Luisa Damiani, Department of Computer Science, University of Milan, 39, Via Comelico, Milan 20135, Italy; email: damiani@di.unimi.it.

DOI: http://dx.doi.org/10.1145/2786756

ACM Reference Format:
Ralf Hartmut Güting, Fabio Valdés, and Maria Luisa Damiani. 2015. Symbolic trajectories. ACM Trans. Spatial Algorithms Syst. 1, 2, Article 7 (July 2015), 51 pages.
DOI: http://dx.doi.org/10.1145/2786756

1. INTRODUCTION

Due to the widespread use of GPS-enabled devices such as smartphones and car navigation systems, the recording of position data has become very easy, and huge amounts of such data are collected every day. In response to this, the research field of trajectory data management, also termed moving objects databases, has been very active in the last 15 years [Zheng and Zhou 2011; Güting and Schneider 2005].

A trajectory describes the movement of an entity, for example, a person, a vehicle, or an animal, over time. At a low level of abstraction, it is a sequence of positions with timestamps, corresponding to the way data are recorded by devices. At a higher level of abstraction, it is a continuous function from time into 2D space that may be represented by an abstract data type.

There is a large body of research on trajectories of this kind (later called *raw* or *geometric* trajectories). For example, they have been integrated into DBMS data models and query languages in the form of abstract data types with suitable operations [Güting et al. 2000], index structures have been developed [Pfoser et al. 2000; Nguyen-Dinh et al. 2010], modeling has addressed movement in networks [Speicys et al. 2003] and uncertainty [Trajcevski et al. 2004], similarity measures for trajectories have been studied [Chen et al. 2005], and techniques for visual analysis have been explored [Andrienko and Andrienko 2013]. Two prototype database systems implementing the model of Güting et al. [2000] exist, SECONDO [Güting et al. 2010] and Hermes [Pelekis et al. 2008].

A lot of effort has gone into data mining on large sets of trajectories [Giannotti and Pedreschi 2008]. Whereas the general goal is to discover any kind of interesting phenomena on such datasets, an important aspect is to associate some meaning with a trajectory as a whole or parts of it. For example, for a tourist, rather than being at some geographic coordinate in France for an interval of time, we would like to understand that he or she is visiting the Louvre or having dinner at a restaurant. For a car, we want to be aware that it was in a traffic jam during a certain period. For an animal observation, we would like to understand that this is migration behavior, or a bird flying in a swarm. For a person moving around, we would like to know whether he or she is walking, going by bicycle, or using a bus.

Obviously such "semantic" information is derived from the original geometric trajectory and will be represented in some symbolic form. It can be viewed as a *trajectory annotation* or the derived symbolic version of the trajectory can afterward be handled independently. We can distinguish three kinds of trajectory annotations:

(1) Semantic information obtained by data mining
(2) Relating a trajectory to the spatiotemporal environment
(3) Properties that can be derived directly from the geometric trajectory

Figure 1 shows some examples of data mining annotations. Several research works have tried to determine transportation modes [Zheng et al. 2010a; Reddy et al. 2010; Stenneth et al. 2011]. For taxis, one is interested in the periods when they are occupied or free. More generally, a goal is activity recognition and the discovery of meaningful places. Activities may be walking, driving, or sleeping, for example. Significant locations can be home, workplace, bus stops and parkings typically used, homes of friends, and stores the person frequently shops in [Liao et al. 2005; Liu et al. 2006].

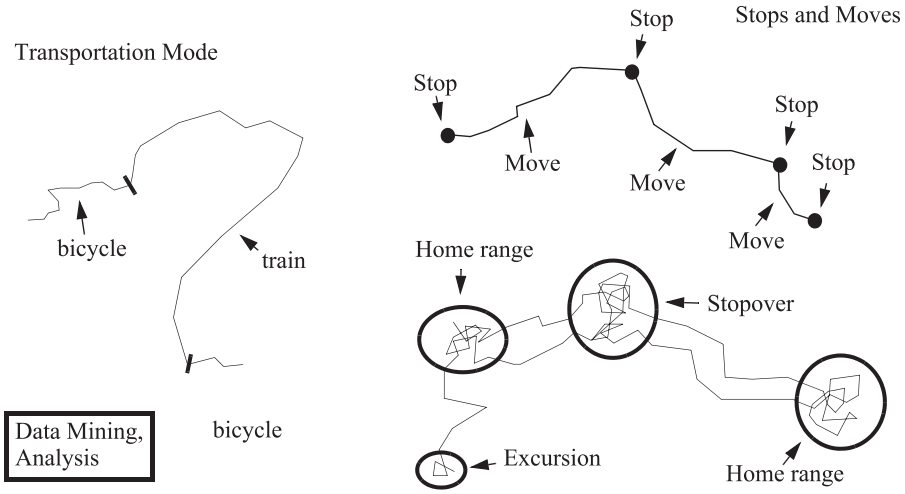


Fig. 1. Trajectory annotation with results of data mining.

One approach to defining "semantic trajectories" is to view a trajectory as a sequence of stops and moves [Spaccapietra et al. 2008]. Stops are the meaningful places that a person has visited and moves are the transitions between them.

In the field of animal ecology, one is interested in the behavior of animals as expressed by their movements. A major challenge is to discover their home ranges and migration patterns. There may be occasional excursions from the home range, to be distinguished from migration, which may contain stopovers as shorter breaks in the migration trip [Li et al. 2011; Kie et al. 2010; Urbano et al. 2010].

An important step in analysis is to relate geometric trajectories to their environment in space and/or time (e.g., temperature, weather). One fundamental task is to find for a vehicle the roads that have been used, called map matching [Quddus et al. 2007; Newson and Krumm 2009]. Another important abstraction is to describe a trajectory relative to a partition of the plane (e.g., states or districts of a country) as a sequence of labels of regions traversed. This has been the basis of earlier work on pattern matching languages for trajectories [du Mouza and Rigaux 2005; Vieira et al. 2010], discussed in Section 8 in detail. Yet another symbolic representation concerns the cell towers of mobile phone providers that a cell phone has registered with during the movement. This kind of data is often obtained without geometric trajectories, collected from providers. Movement and activity analysis can be based on these symbolic representations [Eagle and Pentland 2006; Ahas et al. 2008], possibly deriving geometric trajectories from the known locations of cell towers [Wu et al. 2014].

Finally, some properties such as direction of movement [Frank 1996], speed, acceleration, and altitude can be derived by introducing a classification on the value ranges in a symbolic form, which makes them available to new types of queries as developed in this article.

The data mining community, interested in finding meaningful aspects of trajectories, has recognized the need for representing the results of analysis, and there has been a trend to consider so-called "semantic trajectories" [Spaccapietra et al. 2008, 2013]. Initially, the modeling focused on stops and moves; later this was a bit generalized. A detailed discussion is provided in Section 8. Generally, this line of research has focused on the process of deriving semantic information and has looked at representations from the point of view of conceptual modeling, providing extended entity-relationship

models. For querying, they need to be mapped to corresponding sets of relations. These representations are not suitable for simple and elegant querying.

The database problem of representing annotated trajectories (or just the semantic annotation) in a way that allows for easy, flexible, and efficient querying (e.g., including indexing techniques) has so far been addressed only in a very limited way.

The purpose of this article is threefold:

(i) We propose *annotated trajectory databases* as a new direction of research, focusing on the database rather than the conceptual modeling aspects.
(ii) We define a class of simple generic models for trajectory annotations, called *symbolic trajectories*. They can represent all annotations discussed earlier and so provide a data model for annotated trajectory databases.
(iii) We propose a new query language for symbolic and annotated trajectories based on pattern matching.

(i) The work of this article is just a first step into the study of annotated trajectory databases. There are many more interesting issues to be explored. Some of them are mentioned in Section 9.

(ii) A symbolic trajectory is in its basic form just a time-dependent label, that is, a function from time into label values. Labels are just short character strings. Such a function can be represented as a sequence of pairs $<(i_1, l_1), \ldots, (i_n, l_n)>$, where i_j is a time interval and l_j a label. Time intervals are disjoint (possibly adjacent) and the pairs in the sequence are ordered by time. For example, a simple symbolic trajectory would be[1]

```
< ([8:30 - 8:45], walk), ([8:45 - 9:13], train), ([9:13 - 9:19], walk) >
```

We follow the framework of Güting et al. [2000] and represent a symbolic trajectory as an abstract data type called *moving*(*label*), or *mlabel* for short. The framework has data types such as *moving*(*point*) / *mpoint* to represent a geometric trajectory, *moving*(*real*) / *mreal* to represent a time-dependent real (e.g., speed, heading, or the distance between two moving objects), and so forth. The new type is seamlessly integrated into the framework and inherits generic operations (e.g., **atinstant** to evaluate it at some instant of time, **deftime** to get the total time interval when it is defined).

These data types can all be used as attribute types in a relational model; hence, one can construct a relation describing moving objects (each tuple representing one moving object) with attributes describing geometric together with symbolic information. For example, we may have a relation schema

```
Vehicles (Id: int, Trip: mpoint, RoadName: mlabel, Speed: mlabel)
```

where the road name is obtained from map matching and the speed from a classification of speeds. Hence, this relation represents a set of annotated trajectories. Queries on the symbolic part can be used to retrieve the tuples with the geometric trajectories, hence the complete annotated trajectory.

A crucial aspect of this model is that the symbolic trajectories include the time interval for each symbolic value. This allows one to come back to the geometric trajectory and identify the parts with certain properties.

Beyond the basic type *moving*(*label*), three more types for symbolic trajectories are provided that allow one to have time-dependent sets of labels, (symbolic references to) places, and sets of places. They are motivated when they are introduced.

[1]This representation is simplified. Time intervals do contain absolute dates. More precise notations for time intervals are defined later in the article.

(iii) The main technical contribution of this article is a language for pattern matching and rewriting of symbolic trajectories. Matching is used to retrieve symbolic trajectories fulfilling a given pattern. Rewriting can be used to translate a symbolic trajectory into some other form, classify it into certain categories, or retrieve the parts of a symbolic trajectory matching a pattern.

The basic idea of the language is illustrated next:

```
* (_ taxi) (_ bus) *
```

This pattern matches symbolic trajectories containing adjacent pairs (called *units*) where a transfer occurs from taxi to bus. A pattern to match a unit is denoted as $(x\ y)$, x a time interval specification, y a label specification; the symbol * matches any sequence of units. In contrast, the pattern

```
* (monday taxi) X (_ bus) * // duration(X.time) > 20 * minute
```

requires that the transfer occurs on a Monday and that the bus trip takes more than 20 minutes. This pattern contains a variable X and a condition.

Patterns can be extended to rules that can be used in rewriting:

```
* W (monday taxi) X (_ bus) * // duration(X.time) > 20 * minute
=> W X
```

This rule returns for each input trajectory all symbolic trajectories that can be obtained from it by selecting two adjacent units matching the pattern

```
(monday taxi) (_ bus)
```

Hence, one can retrieve all transitions from taxi to bus occurring in the possibly long symbolic trajectory.

A few works exist that provide simple symbolic representations designed to support querying or data mining (e.g., du Mouza and Rigaux [2005], Vieira et al. [2010, 2011], Zheng et al. [2013], and Zhang et al. [2014]). Among them, du Mouza and Rigaux [2005], [Vieira et al. 2010], and Vieira et al. [2011] provide a pattern query language. These works are discussed in detail in the related work section (Section 8). Compared to our proposal, they have severe shortcomings:

—All these proposals address special cases and provide ad hoc representations. For example, du Mouza and Rigaux [2005] address sequences of labels arising from traversal of a partition of the plane; labels are not associated with precise time intervals. The proposal in Vieira et al. [2010, 2011] does not provide a data model at all; it basically uses raw trajectories and derives time intervals and labels internally in a data structure. Again, labels are fixed to describe traversed regions of the plane. The model of Zheng et al. [2013] does not have time at all and otherwise is a sequence of locations annotated with sets of activities. The model of Zhang et al. [2014] is a sequence of timestamped locations annotated with a semantic label where timestamps are instants of time, not time intervals. All these models are not suitable for representing and querying generic annotations.
—The only approach that associates precise time intervals with symbolic information is Vieira et al. [2010, 2011]. Even in this approach, the handling and querying of time intervals is quite limited. Precise time intervals are important to come back from a query on symbolic annotations to the geometric trajectory, for example, to extract from it the pieces retrieved by the symbolic query.
—The data models used for geometric trajectories are primitive (sequences of timestamped points) and ignore the state of the art [Güting 2009].

—None of these approaches is concerned about connecting the particular query mechanism offered (patterns, knn queries) with querying on geometric trajectories. The querying mechanisms offered do not integrate with any querying environment of the DBMS handling the geometric trajectories. However, in annotated trajectory databases, it is crucial to have a consistent framework for querying both symbolic and geometric trajectories (as demonstrated in Section 4).
—None of the approaches is presented in a system context where the system and the proposal is publicly available.
—None of these approaches provides rewriting of trajectories into other forms.
—All of these approaches provide a closed language; there is no extensibility.

In contrast, the novel contributions of this article are the following:

—We introduce the concept of symbolic trajectories as a generic representation of meanings for trajectories. Symbolic trajectories can be used as annotations of geometric trajectories as they contain precise time information. They are formalized as abstract data types and integrated into an existing comprehensive framework of data types for moving objects, inheriting generic operations from the framework.
—A language for pattern matching and rewriting of symbolic trajectories is defined, providing rigorous formal definitions for the syntax and semantics. In contrast to earlier work,
  —it is not restricted to special cases (e.g., labels of a sequence of areas traversed) but handles symbolic trajectories in full generality,
  —it not only refers to labels but also provides sophisticated specification of temporal conditions,
  —it provides not only pattern matching but also rewriting and classification of trajectories,
  —the language is not closed but connects to the full power of the querying environment, allowing one to use any available operation for conditions or assignments in rewriting.
—Data model and language have been fully implemented in the DBMS prototype Secondo. Efficient implementation of the language is presented in detail.
—An experimental evaluation based on the well-known BerlinMOD benchmark provides detailed insight about the efficiency of the implementation. It demonstrates the advantages of using symbolic trajectories over raw trajectories for the aspects covered by the symbolic trajectory representation.
—The implementation is freely available for experiments and practical use together with the Secondo DBMS prototype. It has already been demonstrated in Valdés et al. [2013] and Damiani et al. [2014b].

The rest of the article is structured as follows: Symbolic trajectories are introduced in Section 2. Section 3 presents the pattern language including formal definitions of syntax and semantics. Section 4 presents example queries from two real-life annotated trajectory databases to demonstrate usability. Section 5 compares our approach to the idea of "querying by regular expressions" and systematically discusses the features going beyond that. Section 6 explains in detail data structures and algorithms for implementing the model and illustrates them by examples. Section 7 first provides an experimental evaluation of the main query operations for pattern matching, rewriting, and classification. In a second set of experiments, raw trajectories from the BerlinMOD benchmark [Düntgen et al. 2009] are mapped to symbolic trajectories (for names of roads traversed) and the two representations are compared with respect to storage consumption and query time. Section 8 discusses related work, and Section 9 concludes with an outlook on future work.

2. SYMBOLIC TRAJECTORIES

The model of symbolic trajectories proposed in this article fits into and extends a comprehensive framework for representing and querying moving objects in databases [Erwig et al. 1999; Güting et al. 2000; Forlizzi et al. 2000]. The general idea is to provide a collection of abstract data types to describe moving objects and the operations applicable to them. For example, *moving point* (or *mpoint*, for short) is a data type to represent a time-dependent location in the Euclidean plane, *line* is a spatial data type describing a continuous curve in the plane, and *mreal* is a type to represent time-dependent real values. Operation **trajectory** maps a moving point to a *line* value, and operation **distance**, applied to two *mpoint* values, returns their time-dependent distance as an *mreal*. An *mpoint* m may represent the trip of a car, **trajectory**(m) would be the path in the plane taken, and **distance**(m_1, m_2) may be the time-dependent distance between two cars.

This is embedded into a DBMS data model (e.g., an object-relational model) as follows. The data types can be used as attribute types. Hence, we can have a relation describing car trips with schema:

```
Vehicles (Id: string, Trip: mpoint)
```

The operations can be used in queries. For example, one can find pairs of vehicles that have been closer to each other than 100 meters by a query

```
SELECT v1.Id, v2.Id
FROM Vehicles as v1, Vehicles as v2
WHERE minimum(distance(v1.Trip, v2.Trip)) < 0.1
```

using a further operation **minimum** that maps an *mreal* into a *real*.

A symbolic trajectory is in the most simple form just a time-dependent symbol, called a *label*, where a label is simply a character string. Hence, conceptually, a symbolic trajectory is a function

$$f : A_{instant} \rightarrow A_{label},$$

where *instant* is the data type representing time, and *label* is the type of labels. For a data type t, let A_t denote its domain, that is, the set of possible values.

The framework of Güting et al. [2000] provides a type constructor called *moving*. Given a type α, *moving*(α) is the type whose values are partial continuous functions from $A_{instant}$ into A_α. Obviously, this is exactly what we need as a conceptual model for a symbolic trajectory, which is therefore represented as a type *moving*(*label*), *mlabel* for short.

There are three further variants: a time-dependent *set of labels*, a time-dependent *place*, or a time-dependent *set of places*. We will introduce corresponding data types *moving*(*labels*), *moving*(*place*), and *moving*(*places*) with abbreviations *mlabels*, *mplace*, *mplaces*, respectively.

A place is a pair consisting of a label and an integer such as (cinema, 114), where the integer component is a reference to some repository of geometries. Geometries can be of data types *point*, *line*, or *region*. Hence, a place is a symbolic representation of an entity in space with a reference to its precise geometric location or extent.

The motivation for these additional data types is the following. First, places (*mplace*) are useful in the case of relating raw trajectories into the environment; we not only can use the names (labels) of entities in the environment but also can formulate conditions on the geometries. For example, for a symbolic trajectory that describes regions traversed, we can require that the first and last region are adjacent. Second, having sets of places is useful in relating a raw trajectory to a set of overlapping regions, for example, or a set of nearest neighbors. This is illustrated in Figure 2, where a trajectory

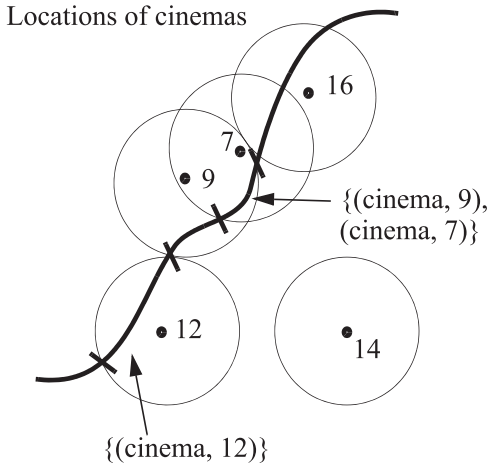


Fig. 2. Trajectory annotation with places for overlapping regions. Black dots represent cinemas; circles show the area within a 500-meter distance.

is mapped to a symbolic form having the set of cinemas within a 500-meter distance.

Third, data types *mlabels* and *mplaces* are closed under set operations $\cup$, $\cap$, and $\setminus$ on symbolic trajectories. For example, let f be a symbolic trajectory representing the sequence of roads traversed, and g a symbolic trajectory representing the transportation mode; then $f \cup g$ is the symbolic trajectory representing for each instant of time both the current road and the transportation mode.

Formally, we introduce four data types, *label*, *labels*, *place*, and *places*. Their semantics are defined by the sets of possible values, that is, the domain, or the carrier sets in algebraic terminology.

Definition 2.1. The carrier sets for the types *label*, *labels*, *place*, and *places* are defined as

$A_{label} := V^* \cup \{\perp\}$, where V is a finite alphabet,
$A_{labels} := 2^{V^*} \cup \{\perp\}$,
$A_{place} := (V^* \times \mathbb{N}) \cup \{\perp\}$,
$A_{places} := 2^{V^* \times \mathbb{N}} \cup \{\perp\}$.

Each carrier set contains an undefined value $\perp$ consistent with definitions in Güting et al. [2000]. The moving constructor in Güting et al. [2000] defines the related time-dependent types *moving*(*label*), *moving*(*labels*), *moving*(*place*), and *moving*(*places*).

Discrete Model. So far we have considered symbolic trajectories within the so-called abstract model, viewing them just as functions of time. For implementation, one needs to provide a so-called discrete model that describes the values of data types in terms of finite representations [Forlizzi et al. 2000]. At this level, a symbolic trajectory is represented as a sequence of pairs (called *units*), where each unit consists of a time interval and a value from the respective type (*label*, *labels*, *place*, *places*).[2] Within the sequence, the time intervals of units are disjoint (but possibly adjacent) and units are ordered by time intervals.

Hence, a symbolic trajectory may be denoted as $u_1 \ldots u_n$ for $n \geq 0$, where u_i is a unit, or as a sequence of pairs $<(i_1, l_1), \ldots, (i_n, l_n)>$, where i_j is a time interval and l_j a label (set of labels, respectively, etc.). Time intervals are represented as four-tuples

[2]We also have related data types to represent single units called *ulabel* and so forth.

Table I. Operations for Interaction of Temporal Values with Values in Domain and Range

Operator	Signature		
atinstant:	$\underline{moving}(\alpha) \times \underline{instant}$	$\rightarrow \underline{intime}(\alpha)$	
atperiods:	$\underline{moving}(\alpha) \times \underline{periods}$	$\rightarrow \underline{moving}(\alpha)$	
initial, **final**:	$\underline{moving}(\alpha)$	$\rightarrow \underline{intime}(\alpha)$	
present:	$\underline{moving}(\alpha) \times \underline{instant}$	$\rightarrow \underline{bool}$	
present:	$\underline{moving}(\alpha) \times \underline{periods}$	$\rightarrow \underline{bool}$	
at:	$\underline{moving}(\alpha) \times \alpha$	$\rightarrow \underline{moving}(\alpha)$	[1D]
at:	$\underline{moving}(\alpha) \times \beta$	$\rightarrow \underline{moving}(\alpha)$	[2D]
atmin,**atmax**:	$\underline{moving}(\alpha)$	$\rightarrow \underline{moving}(\alpha)$	[1D]
passes:	$\underline{moving}(\alpha) \times \beta$	$\rightarrow \underline{bool}$	

(s, e, lc, rc), where s and e are instants with $s < e$, and lc and rc are Booleans denoting whether the interval is left-closed and/or right-closed. This makes it possible to have adjacent but disjoint intervals. See Forlizzi et al. [2000] for full formal definitions.

Hence, an example symbolic trajectory of type *mlabel* is

```
< ( (2013-01-17-9:02:30, 2013-01-17-9:05:51, T, F), "Queen Anne St"),
  ( (2013-01-17-9:05:51, 2013-01-17-9:10:16, T, F), "Wimpole St"),
  ...
  ( (2013-01-17-9:18:44, 2013-01-17-9:20:10, T, F), "Queen Anne St") >
```

It represents the sequence of road names of roads traversed by someone who did a round-walk of about a quarter of an hour on January 17, 2013.

Operations on Symbolic Trajectories. Due to the integration into the model of Güting et al. [2000], we inherit a comprehensive set of generic operations. For example, Table I shows one class of operations that deals with the interaction of temporal values (i.e., functions of time) with values in their domain and range (Güting et al. [2000], Table XV).

We may substitute any of our new types *label*, *labels*, *place*, and *places* for α and obtain a valid signature, for example, **atperiods**: $\underline{moving}(\underline{label}) \times \underline{periods} \rightarrow \underline{moving}(\underline{label})$. This operation restricts a symbolic trajectory of type *mlabel* to a certain set of disjoint time intervals (represented by type *periods*). For detailed explanations of Table I, see Güting et al. [2000].

As a result, we already have an expressive query language for querying trajectories of various kinds including symbolic trajectories. This is illustrated in the following example.

Example 2.2. Assume we have a relation with symbolic trips of people captured as road profiles (road names of roads traversed). The schema is

```
SymTrips (Name: string, Trip: mlabel)
```

We can formulate the following queries (operations used[3] are shown in Table II):

—Find all trips passing through Baker street.

```
SELECT * FROM SymTrips WHERE Trip passes "Baker St"
```

—For these trips, determine the time intervals when they were in Baker street.

[3]Operations **sometimes** and **theInstant** are not defined in Güting et al. [2000]. **sometimes** is shown to be a derived operation in Güting and Schneider [2005], Exercise 4.5. **theInstant** is available in the implementation in the SECONDO system.

Table II. Operations Used in Queries of Example 2.2

Operator	Signature		Syntax
passes:	$\underline{moving(label)} \times \underline{label}$	$\rightarrow \underline{bool}$	`_ # _`
at:	$\underline{moving(label)} \times \underline{label}$	$\rightarrow \underline{moving(label)}$	`_ # _`
deftime:	$\underline{moving(label)}$	$\rightarrow \underline{periods}$	`# ( _ )`
atinstant:	$\underline{moving(label)} \times \underline{instant}$	$\rightarrow \underline{intime(label)}$	`_ # _`
val:	$\underline{intime(label)}$	$\rightarrow \underline{label}$	`# ( _ )`
intersection:	$\underline{moving(label)} \times \underline{moving(label)}$	$\rightarrow \underline{moving(label)}$	`# ( _, _ )`
=:	$\underline{moving(label)} \times \underline{moving(label)}$	$\rightarrow \underline{moving(bool)}$	`_ # _`
sometimes:	$\underline{moving(bool)}$	$\rightarrow \underline{bool}$	`# ( _ )`
theInstant:	$\underline{int} \times \underline{int} \times \underline{int} \times \underline{int} \times \underline{int}$	$\rightarrow \underline{instant}$	`# (_,_,_,_,_)`

```
SELECT Name, deftime(Trip at "Baker St") as AtBaker
FROM SymTrips
WHERE Trip passes "Baker St"
```

—In which road was John on January 17, 2013, at 6:30 a.m.?

```
SELECT val(Trip atinstant theInstant(2013, 1, 17, 6, 30)) as Road
FROM SymTrips
WHERE Name = "John"
```

—Find any pairs of people that have been in the same road at the same time. Provide the parts of the trips where they have been in the same road.

```
SELECT s1.Name, s2.Name, intersection(s1.Trip, s2.Trip) as CommonRoads
FROM SymTrips as s1, SymTrips as s2
WHERE sometimes(s1.Trip = s2.Trip)
```

The operations used in Example 2.2 are shown in Table II with the specific instantiations of the generic signature used in the query. For each operator, its syntax is specified in the last column; here # denotes the operator and _ an argument.

The report version of this article [Güting et al. 2013] describes more formally the integration of the new types into the type system of Güting et al. [2000] and into the mechanisms for defining generic operations; this was omitted here due to space restrictions.

3. PATTERN MATCHING AND REWRITING

The generic operations inherited for symbolic trajectories from Güting et al. [2000] already permit expressive queries. Nevertheless, we are now interested in making the language even more expressive by providing an additional facility to retrieve symbolic trajectories matching a user-defined pattern and to manipulate trajectories by rewriting them into some other form. For example, rewriting allows one to extract certain pieces of interest, to aggregate subsequences of units to some higher-level semantics (expressed by a corresponding label), or even to classify the whole trajectory by assigning an adequate label. Note that rewriting in particular allows one to determine positions where matches occur—with matching alone this is not possible.

We discuss pattern matching and rewriting for the most simple type, $\underline{mlabel}$. It is straightforward to extend this to the other three types of symbolic trajectories.

3.1. Overview

The proposed language provides *patterns* that can match symbolic trajectories. Patterns may contain *variables,* which in the process of matching are bound to either single units or subsequences of a symbolic trajectory. After binding, properties of the bound units or sequences can be accessed via *attributes* of the variables and then be

used in *conditions*. These conditions further restrict what is matched. Conditions may use arbitrary operations available in the host DBMS. The syntax for patterns with conditions is

```
<pattern> // <conditions>
```

and an example is

```
D (_ "Queen Anne St") * A (_ "Queen Anne St")
    // (A.end - D.start) < (20 * minute)
```

This query pattern will match all symbolic trajectories (defined over street names passed), which are round trips starting and ending at Queen Anne Street taking less than 20 minutes. Here the first line contains the pattern, the second the condition. Within the pattern, the unit patterns shown for the first and the last unit contain the label "Queen Anne St." In between, the * matches the intermediate sequence of units. *D* and *A* are variables that will be bound in matching to the first and last unit of a trajectory. Via attributes *A.end* and *D.start,* the condition can be formulated that the duration of the entire trip should be less than 20 minutes.

Moreover, after matching, interesting parts of a symbolic trajectory can be retrieved by a *rewrite rule*, constructing a derived symbolic trajectory. The derived trajectory can be modified by assigning values to fields within units. So, for example, new labels can be written. The syntax for rewrite rules is an extension for that of patterns with conditions

```
<pattern> // <conditions> => <result pattern> // <assignments>
```

and an example is

```
* X (monday "Wimpole St.") * => X   // X.label := "at stadium"
```

This rule finds all trajectories passing on a Monday through Wimpole Street, retrieves the units where this happens (so one can find out the exact times, for example) and resets the label to "at stadium".

In the following subsections, all these concepts will be introduced in more detail. Their semantics are defined formally, to have a precise specification for users and implementors.

3.2. Patterns

In Section 2, we have seen that a symbolic trajectory can be represented as a list of the form

```
< ((s1 e1 lc1 rc1) l1) ... ((sn en lcn rcn) ln) >
```

or, a bit more abstractly,

```
(t1 l1) ... (tn ln)
```

This is a list of units where each unit is a pair consisting of a time interval and a label. The time interval consists of the four components *start*, *end*, *leftclosed*, and *rightclosed*.

A *pattern* describes such a list with some desired structure or contents by approximating this notation. It might look as follows:

```
(_ a) (_ b) * (_ c) *
```

Here a pair in parentheses denotes a unit pattern; that is, it matches a unit. The underscore symbol matches any corresponding element of a unit pair; hence, any time intervals are matches. The label of the pattern matches a unit label if they are equal.

The symbol * matches a sequence of units (0 or more). Hence, the pattern matches a sequence (an *mlabel* value) having first a unit with label a, then a unit with label b, then an arbitrary sequence of units, then a unit with label c, then an arbitrary sequence of units. Beyond the symbol * there are further patterns that can match sequences, for example, alternatives or repeating subsequences:

```
[(_ a) | (_ b)]
[(_ a) (_ b)]*
```

Here the first line denotes a pattern that matches a single unit with label either a or b. The second matches a sequence with alternating labels a and b. In other words, we support notations for regular expressions. For them, square brackets are used, as parentheses are already employed for unit patterns.

We now formalize this, calling it a *simple pattern*. A *pattern* will later be a simple pattern extended by variables.

Definition 3.1. A *simple pattern* is a sequence of *simple pattern elements* $<p_1, \ldots, p_n>$, where each p_i is a *unit pattern* or a *sequence pattern*.

(i) A *unit pattern* has one of the forms (t l), (_ l), (t _), or (), where t is a time interval specification, l is a label specification, and _ is a wildcard symbol. In the most simple cases, $t \in D_{\underline{instant}} \times D_{\underline{instant}}$ and $l \in D_{\underline{label}}$.
(ii) A *sequence pattern* has one of the forms *, +, [p], [p_1 | p_2], [p]+, [p]*, or [p]?, where p, p_1, p_2 are simple patterns.

More complex time or label specifications are addressed in Section 3.6. Unit patterns and sequence patterns of the forms * and + are called *atomic pattern elements* (*atoms*, for short). This notion is relevant in the implementation (Section 6).

Definition 3.2 (*Pattern Matching*).

(i) *Unit Patterns.* Let $u = (u_t, u_l)$ be a unit of type *ulabel* (i.e., a unit with u_l of type *label*).
—(t l) matches $u :\Leftrightarrow u_t \subseteq t \wedge u_l = l$.
—(_ l) matches $u :\Leftrightarrow u_l = l$.
—(t _) matches $u :\Leftrightarrow u_t \subseteq t$.
—() matches u.
When a unit pattern p matches a unit u, then it also matches the single unit sequence $U =< u >$.
(ii) *Sequence Patterns.* Let $U = <u_1, \ldots, u_n>, n \geq 0$ be a sequence of units, each u_i of type *ulabel*.
—* matches U.
—+ matches $U :\Leftrightarrow n > 0$.
—[p] matches $U :\Leftrightarrow p$ matches U.
—[p_1 | p_2] matches $U :\Leftrightarrow p_1$ matches $U \vee p_2$ matches U.
—[p]+ matches $U :\Leftrightarrow$ there exists a partitioning of U into subsequences $U_1 \ldots U_m, m \geq 1$ such that $U = U_1 \circ \cdots \circ U_m$ and $\forall i \in \{1, \ldots, m\}$, p matches U_i, where $\circ$ denotes concatenation.
—[p]* matches $U :\Leftrightarrow$ [p]+ matches $U \vee n = 0$.
—[p]? matches $U :\Leftrightarrow$ [p] matches $U \vee n = 0$.
(iii) Let $U =< u_1, \ldots, u_n >, n \geq 0$ be a sequence of units, each u_i of type *ulabel*. Let $P = p_1 \ldots p_m$ be a simple pattern.
P matches $U :\Leftrightarrow$ there exists a partitioning of U into subsequences $U_1 \ldots U_m$ such that $U = U_1 \circ \ldots \circ U_m$ and $\forall i \in \{1, \ldots, m\}$, p_i matches U_i.

Example 3.3. In Section 2, we have seen an example trajectory in a database of personal trips:

```
< ( (2013-01-17-9:02:30, 2013-01-17-9:05:51, T, F), "Queen Anne St"),
  ( (2013-01-17-9:05:51, 2013-01-17-9:10:16, T, F), "Wimpole St"),
  ...
  ( (2013-01-17-9:18:44, 2013-01-17-9:20:10, T, F), "Queen Anne St") >
```

The pattern

```
(_ "Queen Anne St") * (_ "Queen Anne St")
```

could be used to retrieve all round trips starting and ending in Queen Anne Street, including the one shown previously.

3.3. Variables

We now add variables to patterns. Their purpose is twofold: (1) to allow us to specify further conditions on subsequences matched by pattern elements, and (2) to control rewriting. Variables are written as words starting with a capital letter (as in Prolog). Usually we denote them by just one letter. They can be associated with unit patterns or sequence patterns and accordingly be bound to units or sequences of units. Once they are bound, we can access properties of the unit or the sequence via attributes of the variables. We write variables in front of the patterns to which they are associated. For example:

```
X (_ a) Y (_ b) Z * (_ c) *
```

This pattern has five elements. If the pattern matches an *mlabel*, then variable X is bound to the first unit, variable Y is bound to the second unit, and variable Z is bound to the sequence of units between the two units with labels b and c. No variables exist for the last two elements of the pattern.

Because variables are bound to distinct subsequences—even if the labels are equal, the time intervals differ—it does not make sense to have the same variable more than once in a pattern. We therefore require that all variables occurring in a pattern are distinct.

Definition 3.4. Let V be a domain of variable names. A *pattern* is a sequence of pattern elements $P =< e_1, \ldots, e_n >$, where each e_i is either a pair (v_i, p_i) of a variable $v_i \in V$ and a simple pattern element p_i or just a simple pattern element p_i. In the first case, e_i is called a *variable element*, otherwise a *free element*. In a variable element (v, p), v is called a *unit variable* (a *sequence variable*) if p is a unit pattern (a sequence pattern). All variables are distinct, that is, $i \neq j \Rightarrow v_i \neq v_j$.

For a pattern P, $simple(P)$ denotes the corresponding simple pattern $< p_1, \ldots, p_n >$ and $var(P)$ denotes the set of variables occurring in P.

Definition 3.5. A *binding* is a set of pairs $B = \{(v_1, U_1), \ldots, (v_k, U_k)\}$, where each pair consists of a variable v_i and a sequence of units U_i of type *ulabel*. All variables are distinct. $var(B) = \{v_1, \ldots, v_k\}$ denotes the set of variables occurring in B.

Definition 3.6 (*Pattern Matching with Binding*). Let $U = <u_1, \ldots, u_n>, n \geq 0$ be a sequence of units, each u_i of type *ulabel*. Let $P = <e_1, \ldots, e_m>$ be a pattern and $simple(P) = <p_1, \ldots, p_m>$.

P *matches* U *with binding* $B :\Leftrightarrow$ there exists a partitioning of U into subsequences $U_1 \ldots U_m$ such that $U = U_1 \circ \ldots \circ U_m$ and $\forall i \in \{1, \ldots, m\}$, p_i matches U_i. The binding

is $B = \bigcup_{i=1}^{m} B_i$, where

$$B_i = \begin{cases} \{(v_i, U_i)\} & \text{if } e_i = (v_i, p_i) \\ \emptyset & \text{if } e_i = p_i. \end{cases}$$

As mentioned before, once a variable is bound to a unit or a sequence of units, we can access properties via attributes of the variable. The following definition determines these attributes and their contents.

Definition 3.7 (*Attributes*). Let B be a binding and $b = (v, U) \in B$.

(i) v is a unit variable and $U =< (u_t, u_l) >$, $u_t = (s, e, lc, rc)$. Then v has attributes
—*label* of type *label* with value u_l
—*time* of type *periods* with value u_t
—*start* of type *instant* with value s
—*end* of type *instant* with value e
—*leftclosed* of type *bool* with value lc
—*rightclosed* of type *bool* with value rc

(ii) v is a sequence variable and $U =< (t_1, l_1), \ldots, (t_n, l_n) >$, $n \geq 1$, with $t_i = (s_i, e_i, lc_i, rc_i)$. Then v has attributes
—*labels* of type *labels* with value $\{l_1, \ldots, l_n\}$
—*time* of type *periods* with value $\bigcup_{i=1}^{n} t_i$
—*card* of type *int* with value n
—*start* of type *instant* with value s_1
—*end* of type *instant* with value e_n
—*leftclosed* of type *bool* with value lc_1
—*rightclosed* of type *bool* with value rc_n

(iii) v is a sequence variable and $U =<>$ (the empty sequence). Then v has the same attributes as in case (ii) and $v.labels = \emptyset$, $v.card = 0$, and all other values are undefined, for example, $v.time = \perp$.

For an attribute *attr* of variable v we denote by $type(v.attr, B)$ its type and by $val(v.attr, B)$ its value.

Example 3.8. Continuing the previous examples, we now show the complete trajectory:

```
< ( (2013-01-17-9:02:30, 2013-01-17-9:05:51, T, F), "Queen Anne St"),
  ( (2013-01-17-9:05:51, 2013-01-17-9:10:16, T, F), "Wimpole St"),
  ( (2013-01-17-9:10:16, 2013-01-17-9:13:48, T, F), "Welbeck Way"),
  ( (2013-01-17-9:13:48, 2013-01-17-9:18:44, T, F), "Welbeck St"),
  ( (2013-01-17-9:18:44, 2013-01-17-9:20:10, T, F), "Queen Anne St") >
```

The pattern

```
(_ "Queen Anne St") T * A (_ "Queen Anne St")
```

matches the previous trajectory with binding

```
{(T, < ( (2013-01-17-9:05:51, 2013-01-17-9:10:16, T, F), "Wimpole St"),
       ( (2013-01-17-9:10:16, 2013-01-17-9:13:48, T, F), "Welbeck Way"),
       ( (2013-01-17-9:13:48, 2013-01-17-9:18:44, T, F), "Welbeck St") > ),
 (A, < ( (2013-01-17-9:18:44, 2013-01-17-9:20:10, T, F), "Queen Anne St") > )}
```

Attribute values for variable T are

```
T.labels = {"Wimpole St", "Welbeck Way", "Welbeck St"}
T.time = (2013-01-17-9:05:51, 2013-01-17-9:18:44, T, F)
```

```
T.card = 3
T.start = 2013-01-17-9:05:51, T.end = 2013-01-17-9:18:44
T.leftclosed = true, T.rightclosed = false
```

3.4. Patterns with Conditions

Patterns with variables can now be used to specify additional conditions on the matching of such a pattern with a symbolic trajectory. Conditions are Boolean expressions over attributes of the variables, constants, and database objects using arbitrary operations available on the respective data types. We write a pattern with conditions in the form

```
<pattern with variables> // <condition 1>, ..., <condition q>
```

Example 3.9. For example, we can restrict the previous round-trip query to trips taking no more than 20 minutes:

```
D (_ "Queen Anne St") * A (_ "Queen Anne St")
    // (A.end - D.start) < (20 * minute)
```

where *minute* is a database object representing a duration of 1 minute. We can also find round trips starting and ending at some arbitrary street and passing through no more than 10 different streets:

```
D () T * A () // D.label = A.label, T.card <= 9
```

We now formalize these concepts.

Definition 3.10 (*Databases, Constants, and Operations*).

(i) A database is a set of triples $DB \subseteq \{(n, t, v) \mid n \in N, t \in T, v \in dom(t)\}$, where N is the set of allowed object names, T the set of available data types, and $dom(t)$ the domain of values of type $t \in T$. Object names are distinct (i.e., there are no distinct triples with the same object name).
(ii) A domain of constants is a set of triples $C = \{(c, t, v) \mid c \in C_d, t \in T, v \in dom(t)\}$, where C_d is a set of constant denotations. Distinct triples have different constant denotations.
(iii) A set of operations is given as a family of sets $\Sigma = \{\Sigma_{wt} | w \in T^*, t \in T\}$. For an operator $\sigma \in \Sigma_{wt}$, $w = t_1 \dots t_n$ are the argument types and t is the result type. The operator's evaluation function is $f_\sigma : dom(t_1) \times \cdots \times dom(t_n) \to dom(t)$.

Definition 3.11 (*Expressions Over P*). Let P be a pattern and B a binding for the variables in $var(P)$. Let DB be a database, C a domain of constants, and Σ a set of operations. The set of expressions over P denoted $E(P)$ is defined next. Further, for an expression $e \in E(P)$, its evaluation for binding B is defined as well, denoted $eval(e, B)$.

(i) $(o, t, v) \in DB \Rightarrow o$ is an expression of type t, and $eval(o, B) = v$.
(ii) $(c, t, v) \in C \Rightarrow c$ is an expression of type t, and $eval(c, B) = v$.
(iii) $v \in var(P) \wedge attr$ is an attribute of v of type $t \Rightarrow v.attr$ is an expression of type t, and $eval(v.attr, B) = val(v.attr, B)$.
(iv) For $m \geq 0, e_1, \dots, e_m$ are expressions of types $t_1, \dots, t_m$, respectively, and $\sigma \in \Sigma_{t_1,\dots,t_m,t} \Rightarrow \sigma(e_1, \dots, e_m)$ is an expression of type t, and $eval(\sigma(e_1, \dots, e_m), B) = f_\sigma(eval(e_1, B), \dots, eval(e_m, B))$.

Definition 3.12. A *pattern with conditions* is a pair (P, C), where P is a pattern and C a set of expressions of type $\underline{bool}$ over P.

Definition 3.13 (*Pattern Matching for Patterns with Conditions*). Let $U = <u_1, \ldots, u_n>, n \geq 0$ be a sequence of units, each u_i of type *ulabel*. Let (P, C) be a pattern with conditions. U *matches* (P, C) *with binding* $B :\Leftrightarrow U$ matches P with binding B and $\forall c \in C : eval(c, B) = true$.

3.5. Rewriting

Patterns with variables allow us to rewrite a given trajectory into some other form.

Result Patterns. For rewriting, we first introduce rules of the form

```
<pattern> => <result pattern>
<pattern> // <conditions> => <result pattern>
```

At this point, a result pattern is a subsequence of the variables occurring in the pattern. For example:

```
X (_ a) Y (_ b) * (_ c) * => X Y
```

For any *mlabel* value matching this pattern, the rule returns an *mlabel* value consisting just of the first two units. The result is in any case of type *mlabel* even if only a single unit variable is mentioned in the result pattern as in

```
(_ a) (_ b) Z * (_ c) * => Z
```

It is obvious that resulting *mlabel* values have a correct structure because the resulting sequence of units is just a subsequence of the original sequence of units. This is always correct (i.e., no unit time intervals overlap and units are ordered by time).

Assignments and New Variables. The values bound to variables in result patterns can also be changed. This is possible through *assignments*. Rewrite rules get the general forms:

```
<pattern> => <result pattern> // <assignments>
<pattern> // <conditions> => <result pattern> // <assignments>
```

An assignment has the form

```
<var>.<attr> := <expr>
```

where the type of the attribute and the type of the expression must be the same. For example, a complete rule with conditions and assignments may look as follows:

```
X (_ a) Y (_ b) Z * (_ c) *
       // Z.card > 2, X.start > theinstant(2011, 1, 1)
=> X Y
       // X.label := "u", Y.label := "v"
```

Assignments are allowed only to attributes of unit variables. This is because the attributes of sequence variables describe in general aggregations over the entire matched sequence of units (*labels*, *time*, *card*), so one cannot assign values to them. For the remaining four attributes (*start*, *end*, *leftclosed*, *rightclosed*) corresponding to fields of the first and last matched unit, it would be possible but does not make much sense.

On the other hand, it would certainly be interesting to abstract from a given subsequence and represent it by a single unit with some other label. This is possible by introducing *new variables* not occurring in the pattern. They are by definition unit variables and their attributes have to be set by assignments (except for *leftclosed* and *rightclosed* for which defaults *leftclosed* = *true*, *rightclosed* = *false* apply). New

variables may be inserted at arbitrary positions into a result pattern, but like the other variables, each new variable may occur only once.

Example 3.14. Continuing the previous examples of a personal trip database, assume we wish to classify trips into different "semantic" categories. Say, short trips starting and ending at Queen Anne St. are to be classified as "short walk." This can be done by a rewrite rule:

```
D (_ "Queen Anne St") * A (_ "Queen Anne St")
       // (A.end - D.start) < (20 * minute)
=> X
       // X.label := "short walk", X.start := D.start, X.end := A.end
```

To achieve a simple semantics of assignments, the order in which they are written should not matter. That is, they can be evaluated in any order with the same result. We therefore require that no two assignments to the same attribute of the same variable occur.

There is a slight complication because the attribute *time* overlaps with the four attributes *start*, *end*, *leftclosed*, and *rightclosed*. Attribute *time* is of type *periods*, which represents a set of intervals. In an assignment, one would use a *periods* value with a single interval. The representation of this interval contains the four fields mentioned. Therefore, an assignment to *time* and an assignment to *start* would be conflicting. Hence, we further require that in a set of assignments there is no assignment to the *time* attribute if there is an assignment to one of the other four.

Of course, the use of assignments may lead to incorrect descriptions of result sequences. In such a case, the result sequence is simply undefined. In practice, the user will receive an error message.

The formalization of rewrite rules and their semantics is quite involved. To enhance readability, it is omitted here and can be found in Appendix A.

3.6. Unit Patterns in More Detail

Now that the general structure and semantics of our language for pattern matching and rewriting are clear, we go into more detail on the possible time and label specifications within a unit pattern. So far, we have only shown the most simple form in Definition 3.1. A unit pattern has the general form

```
(<time specification> <label specification>)
```

where each of the two components may be replaced by a wild card "_".

For the first element, the *time specification*, one of the following *time symbols* can be entered, each defining a time interval or a set of time intervals:

—a year, month, or day, written as 2010, 2010-07, or 2010-07-05, respectively
—an hour, minute, or second on a particular day, for example, 2010-07-05-14:30
—a range of dates, for example, 2010∼2011, 2010-07∼2011-03
—a range of times, for example, 2010-07-05-14:30∼2010-07-09-14
—a half-open range, for example, 2005-05∼ or ∼2010-12-06
—a day of the week, that is, one of {sunday, monday, tuesday, . . . , saturday}
—a month of the year, that is, one of {january, . . . , december}
—a time of day such as {morning, afternoon, evening, night}
—a time of the day given by a time interval such as 14:30∼16, 17∼
—the name of a database object of type *periods*
—a set of such specifications

Note that semantic descriptions such as *monday* may be viewed as defining an infinite set of time intervals.

A unit will match such a pattern if its time interval (viewed as an infinite set of instants) is a subset of the set of time intervals specified by the time symbol. For a set of specifications, the unit must fulfill all of them.

The second component of a unit pattern, the *label specification*, can be

—a single label,
—a set of labels denoted $\{label_1, \ldots, label_n\}$, or
—the name of a database object of type *labels*.

Hence, a label specification in general defines a set of labels. A unit will match such a pattern if its label is contained in the set.

3.7. Querying with Patterns and Rules

3.7.1. Pattern Matching and Rewriting. To make pattern matching and rewriting available for querying, we introduce a data type *pattern* and two operators **matches** and **rewrite**. Type *pattern* is used to represent patterns or rules in an efficient data structure. An auxiliary operator **topattern** converts a pattern or rule specified as text into this form:

topattern: *text* → *pattern* _ #

The operator is responsible for parsing the pattern or rule, checking for correctness, and converting it to a corresponding data structure. It is applied in postfix notation, that is, written after the text argument.

Example 3.15. We can store our example rewrite rule as a value of type *pattern*:

```
LET short_walk = 'D (_ "Queen Anne St") * A (_ "Queen Anne St")
  // (A.end - D.start) < (20 * minute) => X
  // X.label := "short walk", X.start := D.start, X.end := A.end' topattern
```

The two main operators accept patterns or rules either as a text or as a *pattern* value. An advantage of using the representation as a *pattern* is that in processing a large set of symbolic trajectories in a query, the overhead of constructing an efficient representation of the pattern (including a nondeterministic finite automaton, see Section 6) occurs only once.

matches: *mlabel* × (*pattern* | *text*) → *bool* _ # _
rewrite: *mlabel* × (*pattern* | *text*) → *set*(*mlabel*) #(_ , _)

The **matches** operator returns true if the pattern matches the *mlabel* value (Definition 3.13). **rewrite** returns one rewritten version of the argument for each way the pattern matches, in total the set $apply(R, U)$ for rule R and *mlabel* value U (Definition A.4 in Appendix A). If the pattern does not match, the result set is empty.

Example 3.16. Suppose we have a relation with personal trips of schema:

```
Trips(Id: int, Trip: mlabel)
```

(1) We can find all trips matching the pattern of the short walk by a query:

```
SELECT * FROM Trips
WHERE Trip matches 'D (_ "Queen Anne St") * A (_ "Queen Anne St")
       // (A.end - D.start) < (20 * minute)'
```

(2) We can also rewrite such trajectories according to the "short walk" rule created in Example 3.15:

```
SELECT Id, rewrite(Trip, short_walk) as Class FROM Trips
```

Here we assume that the SQL environment allows one to define in a select-clause one new attribute through a function returning a set of values and that for each such value, one result tuple is created, copying the values of the other attributes mentioned in the select-clause.

3.7.2. Classification. An interesting application of pattern matching is to classify a large number of symbolic trajectories into certain categories, where each category is specified by some pattern. For example, in a database of personal trips, some categories might be as follows:

—home to work by car
—home to work by bicycle
—piano lesson
—short morning walk
—visit Peter
—downtown shopping
—. . .

We assume that categories are specified in a relation with schema

```
(Description: text, Pattern: text)
```

Given such a table and a set of symbolic trajectories, the problem is now to determine for each trajectory the matching patterns. In principle, one might check all pairs (trajectory, pattern), but it is possible to improve this by checking one trajectory in a single step against all patterns, by preprocessing the set of patterns into a single data structure (a combined finite automaton, see Section 6). As a result of the classification, each symbolic trajectory will be associated with the description entries of matching patterns.

To support classification, we introduce a data type *classifier* and an operation **classify**. The data type is used to keep the efficient data structure for the set of patterns. An operation **toclassifier** constructs it from a table with specifications:

toclassifier: *set* (*tuple* ([Desc: *text*, Pattern: *text*])) → *classifier* _ #

Operation classify takes a symbolic trajectory and a classifier and returns all matching descriptions:

classify: *classifier* × *mlabel* → *set*(*text*) #(_ , _)

Example 3.17. Let a table *Categories* be given with schema

```
Categories (Desc: text, Pattern: text)
```

which describes the various kinds of trips occurring in our personal trip database, and let relation *Trips* be given as before. We first create a classifier from the table:

```
LET descriptions = (SELECT * FROM Categories) toclassifier
```

We can then classify the trips by the query

```
SELECT Id, classify(Trip, descriptions) as Class FROM Trips
```

4. EXAMPLE APPLICATIONS

In this section, we illustrate the use of the pattern language by formulating a somewhat larger number of queries related to two applications. Both are based on real-world annotated trajectory datasets. The first is GeoLife with people's movements annotated by transportation mode. The second uses roe deer movements annotated by the results

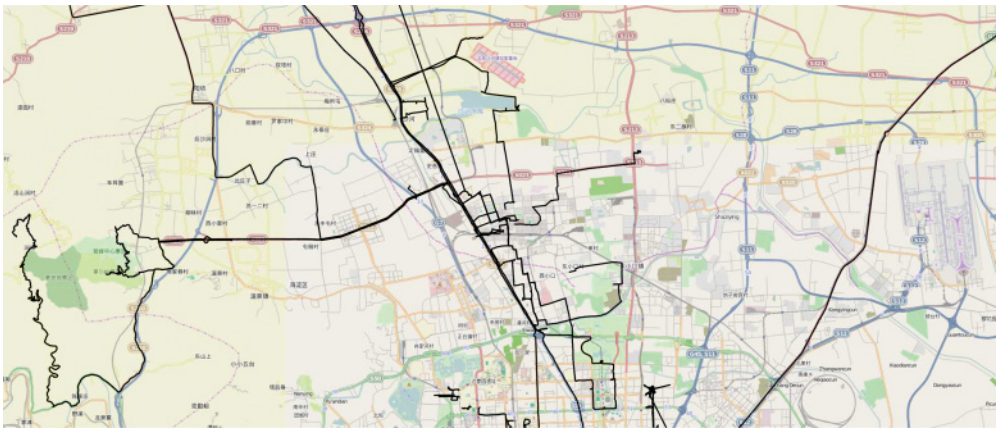


Fig. 3. SECONDO GUI screenshot, showing some of the trajectories from Example 4.2.

of a data mining procedure discovering home ranges and so forth) as sketched in the introduction. The example queries show that it is easy to formulate questions using the pattern language (after a bit of learning) and that it is crucial to combine pattern queries with other operations on moving objects from Güting et al. [2000] available on the host DBMS.

4.1. Example Queries Using Human GPS Data with Transportation Modes

The queries shown in this subsection are based on human movement data from the Microsoft GeoLife project [Microsoft Geolife 2015]. Its 182 participants collected their GPS data over a period of over 3 years. Some of them (about one out of three) also annotated their movement data with their applied mode of transportation (walk, bus, train, etc.), so that we were able to derive more than 4,400 symbolic trajectories along with the corresponding raw data (1.2GBytes as a whole). All examples except the first one use the pattern matching approach as well as other SECONDO operators. The relation has the following schema:

```
GeoLife (Pid: string, Tid: string, Trip: mpoint, Alt: mreal, Trans: mlabel)
```

Here *Pid* and *Tid* denote the person and trip identifier, respectively; *Trip* is the geometric trajectory, *Alt* the altitude (as an *mreal*, a time-dependent real), and *Trans* the transportation mode.

Example 4.1. How many trajectories include at least two bus trips on the same day?

```
SELECT count(*) FROM GeoLife WHERE Trans matches '* X (_ bus) * Y (_ bus) *
  // day_of(X.start) = day_of(Y.end), (Y.end - X.start) < duration(1 0)'
```

Here the *day_of* operation returns the day of the month; therefore, the second condition is necessary. *duration(1 0)* denotes a duration of 1 day, 0 milliseconds.

Example 4.2. How many persons did at least one Sunday morning bike trip of more than 1 hour? (A subset of the trajectories resulting from this query is displayed in Figure 3.)

```
SELECT count(distinct(Pid)) FROM GeoLife WHERE Trans matches
  '* X ({sunday, morning} bike) *
  // get_duration(X.time) > duration(0 3600000)'
```

Example 4.3. Retrieve the id and the maximum speed of all trajectories that include at least five different transportation modes.

```
LET wgs1984 = create_geoid("WGS1984")

SELECT Tid, maximum(speed(Trip, wgs1984)) as MaxSpeed FROM GeoLife
WHERE Trans matches 'X * // no_components(X.labels) >= 5'
```

Here the geoid is needed to get the speed in reasonable units. Operator *speed* applied to an *mpoint* returns the time-dependent speed as an *mreal* value; *maximum* returns the maximal value as a number.

Example 4.4. How many persons did not travel by airplane at all?

```
SELECT count(distinct(Pid)) FROM GeoLife
WHERE not(Trans matches '* (_ airplane) *')
```

Example 4.5. How many trajectories begin and end with the same transportation mode and end less than 1km away from where they start?

```
SELECT count(*) FROM GeoLife WHERE Trans matches 'X () * Y ()
  // X.label = Y.label'
  AND distance(gk(val(initial(Trip))), gk(val(final(Trip)))) < 1000.0
```

Here *initial* and *final* return the first and last (instant value) pair of an *mpoint*; from that, *val* extracts the value. So we obtain the initial and final position. *gk* converts to Gauss-Krüger coordinates so that the *distance* operator returns a result in meters.

Example 4.6. Retrieve all transitions from bus to subway, together with the corresponding *Pid*.

```
SELECT Pid, rewrite(Trans, '* X (_ bus) Y (_ subway) * => X Y') as Transition
FROM GeoLife
```

Example 4.7. For each person, compute the total duration of his or her train journeys.

```
SELECT Pid, sum(get_duration(deftime(rewrite(Trans, '* X (_ train) * => X'))))
  as TrainJourneyDuration
FROM GeoLife
GROUP BY Pid
```

Here *deftime* returns the (set of) definition time interval(s) of the symbolic trajectory returned by rewriting. *get_duration* returns the duration of these time intervals.

4.2. Example Queries Based on Roe Deer GPS Data

In the following, we present queries referring to a set of roe deer GPS data. These data were recorded in the region of Trento, Italy, and have already been used in Damiani et al. [2014a]. More precisely, they include the positions of 26 roe deers, as well as the temperature, altitude, slope, and so forth at each instant of GPS recording (33MBytes as a whole). As a major result of the mentioned paper, the authors derived a symbolic trajectory (type *mlabel*) for each animal, representing either a home range (labels H0, H1, and H2), an excursion (label E0), or a stopover (labels S0 and S1). While the first two queries shown in this subsection only use pattern matching on symbolic trajectories, the other ones combine this technique with further SECONDO operators on the remaining data. The schema of the relation is as follows:

```
Animals (Name: text, Trip: mpoint, Alt: mreal, Temp: mreal,
  Slope: mreal, Sym: mlabel)
```

4.2.1. Using Mlabel.

Example 4.8. How many animals started at their respective home range H0 in November and stopped at another place?

```
SELECT count(*) FROM Animals
WHERE Sym matches '(november "H0") * X () // X.label # "H0" '
```

Example 4.9. How many animals passed all three home ranges H0, H1, and H2?

```
SELECT count(*) FROM Animals
WHERE Sym matches 'X * // X.labels contains tolabels("H0", "H1", "H2")'
```

Example 4.10. For all animals that passed a stopover S0, show their name and the highest altitude and slope they met.

```
SELECT Name, maximum(Alt) as MaxAlt, maximum(Slope) as MaxSlope
FROM Animals WHERE Sym matches '* (_ "S0") *'
```

Example 4.11. Show the initial instant of the observation for all animals that stopped at H1 or H2 in the year 2007.

```
SELECT Name, inst(initial(Trip)) as Start FROM Animals
WHERE Sym matches '* [(2007 "H1") | (2007 "H2")]'
```

Example 4.12. Show the minimum temperature for the trajectories of all animals that spent more than 30 consecutive days at the excursion E0. (Note that the dataset contains trajectories with repeated labels in subsequent units.)

```
SELECT Name, minimum(Temp) as MinTemp FROM Animals WHERE Sym matches
  '* X [(_ "E0")]+ * // (X.end - X.start) > duration(30 0)'
```

Example 4.13. For each animal, retrieve the latest stopover time and altitude corresponding to that instant.

```
SELECT Name, max(inst(final(rewrite(Sym,'* X [(_ "S0") | (_ "S1")] * => X'))))
  as LastStop, val(Altitude atinstant LastStop) as Altitude
FROM Animals GROUP BY Name
```

Example 4.14. Find temporally connected morning excursions (label E0), combine them to one unit, and rename the label to "morning excursion."

```
SELECT rewrite(Sym, '* X [(morning "E0")]+ * => A // A.time := X.time,
  A.label := "morning excursion" ') as MorningExcursions
FROM Animals
```

4.2.2. Using Mplace. Next, we combine the animals' movement data with the underlying geometries of the applied regions. Hence, a further attribute *Sym*2 having the data type *mplace* is added to the relation. Each unit of such a trajectory contains a label (which can be queried in a pattern as for an *mlabel*) and a reference to a region, accessible via *X.extent* in a condition, where *X* is a variable occurring in the pattern. If *X* is a sequence variable, *X.extent* denotes the union of all regions from the units bound to *X*.

Example 4.15. For which animals does the home range H0 have an area of more than 10 square kilometers?

```
SELECT Name FROM Animals
WHERE Sym2 matches '* X (_ "H0") * // size(gk(X.extent)) > 10 * km2'
```

Here the conversion to Gauss-Krüger coordinates by operator *gk* leads to a size in square meters, and km2 is a database object with value 1,000,000.

Example 4.16. Find all animals where the union of the traversed regions has at most five faces.

```
SELECT Name FROM Animals
WHERE Sym2 matches 'X * // no_components(X.extent) <= 5'
```

Example 4.17. Extract all trajectory parts where the associated region overlaps a skiing area (database object "skiing" of type *region*), and set the label to "skiing area."

```
SELECT rewrite(Sym2, '* X () * // X.extent overlaps skiing => X
  // X.label := "skiing area" ' FROM Animals
```

Another application example is detailed in Valdés et al. [2013], where the authors analyze the trajectories of a person over a long period. Furthermore, the report of Güting et al. [2013] provides examples based on personal trips annotated by either significant locations passed or names of roads traversed. It includes discussions on how to construct such symbolic trajectories.

5. COMPARISON TO QUERYING BY REGULAR EXPRESSIONS

Our pattern language includes regular expressions. Actually, they are NOT the main feature of the language. The main features are wildcards * and + to represent uninteresting parts of symbolic trajectories, unit patterns, variables for unit, and sequence patterns and conditions. Regular expression constructs such as $[p_1 \mid p_2]$, $[p]+$, $[p]*$, or $[p]?$ have been added much later than the main features in order to be sure to cover the expressive power of regular expressions.

Unfortunately, the inclusion of regular expressions and perhaps the use of wildcard symbols *, + may lead to the wrong impression that our pattern language is more or less the same as regular expressions. We address this concern in this section. First, we recall the definition of regular expressions (e.g., Aho et al. [2006]).

Let Σ be an alphabet (a finite set of symbols). A subset L of Σ^* is called a language. The symbol ϵ denotes the empty sequence of symbols. On languages L, L_1, L_2, concatenation L_1L_2 and star operation L^* are defined.

Definition 5.1. A *regular expression* r and the language $L(r)$ denoted by it are given by the following rules:

(i) ϵ is a regular expression denoting the language $\{\epsilon\}$.
(ii) For each $a \in \Sigma$, a is a regular expression denoting the language $\{a\}$.
(iii) Let r and s be regular expressions denoting the languages $L(r)$ and $L(s)$. Then,
  (1) $(r)|(s)$ is a regular expression denoting $L(r) \cup L(s)$,
  (2) $(r)(s)$ is a regular expression denoting $L(r)L(s)$,
  (3) $(r)^*$ is a regular expression denoting $(L(r))^*$.

Querying by a regular expression r means to retrieve all symbolic trajectories matching r. Hence, we define what it means for a regular expression to match a symbolic trajectory. A direct correspondence can only be established for the most simple type *mlabel*.

Definition 5.2. Let r be a regular expression and $S =< (i_1, l_1), \ldots, (i_n, l_n) >$ a symbolic trajectory of type *mlabel*, where i_j is a time interval and $l_j \in \Sigma$.
r *matches* $S :\Leftrightarrow < l_1 \ldots l_n > \in L(r)$.

For querying with regular expressions to be feasible, we need a wildcard symbol to match the parts we are not interested in. We assume that *regular definitions* for classes of symbols are available and define $any = a_1|a_2|\ldots|a_m$ for $\Sigma = \{a_1, \ldots, a_m\}$.

THEOREM 5.3. *Any query on a set of symbolic trajectories that can be expressed by a regular expression can be expressed by a pattern (as defined in Section 3).*

PROOF. For each form of a regular expression r we show its translation into an equivalent pattern r', denoting it as $r \rightarrow r'$.

—For each $a \in \Sigma$, $a \rightarrow (\_\ a)$
—$(r)|(s) \rightarrow [r'|s']$
—$(r)(s) \rightarrow r's'$
—$(r)^* \rightarrow [r']^*$

We have not yet addressed the symbol ϵ. It can be used to express either that the entire sequence should be empty (if the entire regular expression is equal to ϵ) or that a part of it is optional. In the first case, a corresponding pattern of our language is $X* //X.card = 0$. In the other case, $(r)|\epsilon \rightarrow [r']?$ and $\epsilon|(r) \rightarrow [r']?$ □

On the other hand, there are numerous features of our pattern language that allow one to formulate queries that cannot be expressed by regular expressions. Whereas abundant examples can be found in the preceding section, we discuss the issue here more systematically.

Time. Symbolic trajectories are not just sequences of symbols but include time intervals. The pattern language can express conditions on time intervals. It has full access to time intervals associated with units or subsequences. Any kind of computation on time intervals or instants is possible due to the availability of operations of the host DBMS and its extensibility (missing operations can be added). Semantic concepts of time such as weekdays, month names, and times of day are available. All of this is not available in regular expression queries.

Note that time intervals cannot be handled as symbols of an alphabet for two reasons. First, for all practical purposes, the set of symbols is infinite. Second, matching via equality of symbols, the only possibility in regular expressions, is not interesting at all as we can never guess precise time intervals occurring in symbolic trajectories. Hence, the idea of using regular expressions over an alphabet that is the product of a label alphabet and a time interval alphabet does not work.

Labels. Assuming that the set of labels in a symbolic trajectory database is equivalent to an alphabet in regular expressions is questionable. For some applications such as street names, the set of labels is huge, whereas alphabets are usually limited. The straightforward translation to finite automata does not work for huge alphabets. Moreover, our approach has more complex types of labels such as sets of labels, places, and sets of places. Finally, regular expressions can only match by equality, whereas in the pattern language any kind of matching is possible (e.g., subset or substring relationships). For example, it is useful to match street names by just giving substrings.

Variables. Our language allows one to associate variables with units or (sub)sequences of a symbolic trajectory. The available information in matched units or sequences is available in attributes of the variables that have well-defined data types of the host system and so are available for operations. This is the basis for

formulating conditions and for rewriting, discussed next. — Variables are not available in regular expressions.

Conditions. Our language permits one to formulate conditions over units or sequences of a symbolic trajectory. For example, different positions in a trajectory can be related, requiring that the same (unknown) label occurs here and there, or that the duration of a subsequence is more than 10 times that of a preceding unit. Counting queries are possible, for example, referring to the length of a sequence or the number of times a label occurs. All of this is not possible in regular expression queries.

DBMS Operations and Extensibility. Conditions and assignments (used in rewriting discussed in the next paragraph) are not formulated in a limited ad hoc language but can use the full power of the host DBMS operations. For example, one can formulate conditions about durations even though durations have not been defined in this article. One can formulate conditions about the size of the convex hull of places occurring in a subsequence matched by some variable. Moreover, the extensibility of the DBMS can be used to add any operation that is later discovered to be important. Hence, we have an open rather than a closed language. — None of this exists in regular expression queries.

Rewriting. Our language allows one to extract matched parts of a symbolic trajectory and transmit them to an output sequence. It is even possible to overwrite attributes of the result sequence by assignments. This is useful for different purposes. One example is to locate the parts of a symbolic trajectory matching a particular pattern, an indispensable feature if the trajectories are large. Another example is to apply a transformation to the input sequence such as classification. — With regular expressions, one cannot extract (find) locations but only match the entire sequence.

THEOREM 5.4. *The following queries can be expressed by the pattern language defined in this article but not by regular expressions:*

(1) *Find trajectories present on a Monday afternoon in 2007.*
```
* ({2007, monday, afternoon} _) *
```
(2) *Find trajectories passing a street whose name contains "Luther."*
```
* X () * // X.label contains "Luther."
```
(3) *Find trajectories passing through the same street in the morning and in the afternoon.*
```
* X (morning _) * Y (afternoon _) * // X.label = Y.label
```
(4) *Find trajectories containing five different transportation modes.*
```
X * // no_components(X.labels) = 5
```
(5) *Find trajectories covering a large area (precisely: where the convex hull area of all places visited is larger than 50km^2).*[4]
```
X * // size(convex_hull(gk(X.locations))) > (50 * km2)
```
(6) *Find all trips that passed through Central Park and return all the times when they were there.*
```
* X (\_ "Central Park") * => X
```

PROOF. See the preceding discussion. □

As a consequence of Theorems 5.3 and 5.4, our language is strictly more expressive than querying by regular expressions.

[4]X.locations is an attribute of type *points* for sequence variables over symbolic trajectories of types *mplace* or *mplaces*. The gk operator transforms geographic to Gauss-Krüger coordinates, which leads to results in square meters. km2 is a database object with value 1,000,000.

6. IMPLEMENTATION

We have implemented the model of symbolic trajectories and the pattern language within the SECONDO DBMS prototype. In this section, we present the algorithms that are used to realize the main operators **matches**, **rewrite**, and **classify** that have been integrated into SECONDO. Since these operators share several common computation steps, it is not appropriate to detail them separately. Instead, we first explain the major algorithms of the **matches** operator in execution order. Based on these results, the remaining operators, whose complexity is beyond **matches**, are described.

For a better understanding of the proposed concepts, we adopt the symbolic trajectory from Example 3.8 and call it M_0. Moreover, we define the patterns P_0 and P_1 (with rewrite rule) as

```
X * Y [({thursday, morning} "Queen Anne St") | (_ "Welbeck St")]+ Z [()]?
       // (Y.end - X.start) < 20 * minute
```

and

```
X * Y [({thursday, morning} "Queen Anne St") | (_ "Welbeck St")]+ Z [()]?
      // (Y.end - X.start) < 20 * minute
=> A Y
      // A.time := X.time, A.label := "start of trip"
```

respectively, both serving as continuous examples throughout this section. The pattern P_0 refers to all trips passing through Queen Anne St. on a Thursday morning or Welbeck St. (at any time) at least once, either exactly before reaching the last unit or at the end of the trajectory, where the duration of X and Y adds up to less than 20 minutes.[5] The pattern P_1 contains the same pattern elements and condition as P_0 and will extract one result trajectory for each (multiple) occurrence of Queen Anne St. or Welbeck St. in the original trajectory, with one unit prepended having the time interval of the trip before passing Queen Anne St. or Welbeck St. (if existing) and the label "start of trip." As we will show later, there are three different result trajectories.

The unit items of the example trajectory M_0 and the patterns P_0 and P_1 are addressed by m_i (the ith unit of M_0, starting from 0) and p_i (the ith atom of P_0 and P_1, starting from 0), respectively. The sizes of M_0 (number of units) and P_0 (number of atoms) are denoted by $|M_0|$ and $|P_0|$, respectively.

In the remainder of this section, first a short overview of the parsing process is given. Subsequently, the major algorithms that are invoked by the operators **matches**, **rewrite**, and **classify** are presented in this order. We also show the algorithms' behavior if applied to the continuous example and analyze their computation cost.

6.1. Parsing

The translation of the input string into its internal representation is done with the help of the tools Flex and Bison and consists of two steps. Initially, the input is treated as if there were no regular expression symbols like `[..|..]+` and `[..]?` in the pattern, and all atoms, conditions, and assignments are processed. After that, we create a new string `regEx` containing only those regular expression parts and an integer for each of the atoms, starting from 0. For P_0, `regEx` reads `0(1|2)+3?`, where `+` and `?` refer to the regular expressions directly preceding them, respectively. Besides, square brackets are transformed into parentheses. This string is then transformed into an NFA by an existing SECONDO operator, which implements the McNaughton-Yamada-Thompson algorithm [Aho et al. 2006, p. 159] to convert a regular expression to an NFA, resulting in the NFA depicted in Figure 4 in graphical (left) and tabular form (right).

[5] `minute` is defined as a SECONDO object of type *duration*, having a length of 60,000ms.

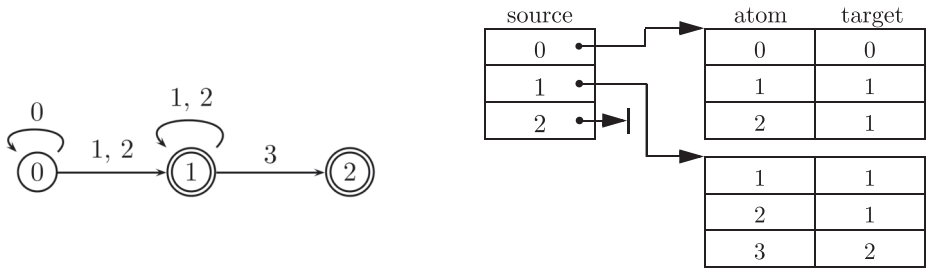


Fig. 4. The NFA after parsing P_0 in graphical form (left) and as a vector of mappings (right).

The computation cost of the parsing phase is linear in the number of atoms, and thus it hardly affects the overall runtime.

Throughout this section, a transition t from an NFA state s_0 to a state s_1 is denoted by $s_0 \xrightarrow{i} s_1$, where i is the position of the transition-triggering atom. In this context, s_0, i, and s_1 are referred to as $t.source$, $t.atom$, and $t.target$, respectively. Finally, let δ_s be the set of transitions going out from the state s. Regarding P_0 or P_1, the set δ_0 equals $\{0 \xrightarrow{0} 0, 0 \xrightarrow{1} 1, 0 \xrightarrow{2} 1\}$, while δ_2 is empty.

6.2. Matching without (Complex) Conditions

In the following, we detail how the NFA transitions are applied for the matching process, in case there are no complex conditions. If a condition can be evaluated immediately, we call it *easy*. More precisely, this is the case if and only if it contains only one variable and this variable refers to a nonwildcard atom. A noneasy condition is called *complex* and must be evaluated in a separate process (cf. Section 6.3). Note that the condition of P_0 is complex.

ALGORITHM 1: *matchesWithoutCondition*

Input: P – a pattern with p atoms, including an NFA δ with n states and a set of final states F;
M – a symbolic trajectory (type moving label) of size m.
Output: *true*, if and only if a final state of the NFA is active after processing the mlabel;

```
1 S ← {0};
2 for i = 0 to m − 1 do                                   // loop over trajectory
3     T ← ∅;
4     foreach s ∈ S do T ← T ∪ δ_s;                       // collect possible transitions
5     if T = ∅ then return false;
6     S ← ∅;
7     foreach t ∈ T do                                    // loop over possible transitions
8         if match(m_i, p_{t.atom}) then S ← S ∪ t.target;
9 return (S ∩ F ≠ ∅);
```

At the beginning of Algorithm 1, we define the set of currently active states S to contain only 0, which is always the initial state for a single NFA. Inside the main loop over the symbolic trajectory, first all transitions starting from any of the states in S are collected in the set T. If T remains empty (either because there is no possible transition or because no state is active), no transition is available, and the algorithm stops and reports a mismatch. Otherwise, we collect the new states by applying those transitions from T whose corresponding atom matches the current unit m_i. More exactly, the

function *match* on the one hand compares the user-specified information from the atom to the unit (cf. Definition 3.2), and on the other hand checks whether the easy conditions corresponding to the atom are fulfilled. After the main loop, *true* is returned if and only if at least one of the final states is active.

To illustrate the algorithm's behavior, we apply it to P_0 and M_0. Since $|M_0|$ equals 5, the outer loop performs five iterations that are detailed subsequently.

Iteration 0. Starting from state 0, the transitions $T = \{0 \xrightarrow{0} 0, 0 \xrightarrow{1} 1, 0 \xrightarrow{2} 1\} = \delta_0$ are feasible. Since the unit m_0 is matched by the atoms p_0 and p_1, the states 0 and 1 become active, that is, $S = \{0, 1\}$.

Iteration 1. In this step, we retrieve $T = \delta_0 \cup \{1 \xrightarrow{1} 1, 1 \xrightarrow{2} 1, 1 \xrightarrow{3} 2\} = \delta_0 \cup \delta_1$. The *match* function returns *true* only for p_0 and p_3, so $S = \{0, 2\}$.

Iteration 2. Since there is no transition from state 2, T equals δ_0. Only p_0 matches, and consequently we obtain $S = \{0\}$.

Iteration 3. Again, $T = \delta_0$. Now the states 0 and 1 become active, since p_0 and p_2 match m_3.

Iteration 4. In the final step, T equals $\delta_0 \cup \delta_1$ as in the first iteration. The last unit is matched by p_0, p_1, and p_3, so $S = \{0, 1, 2\}$ at the end.

The result is *true*, since a final state is active after the loop over M_0.

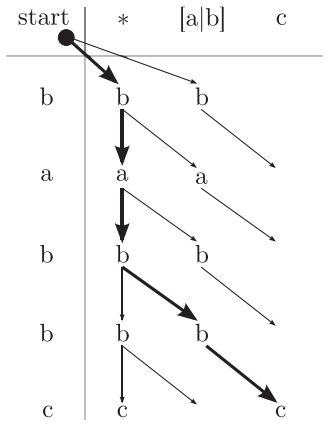
Obviously, the complexity of Algorithm 1 is linear in m, the number of units of the moving label. Let p be the number of atoms of the considered pattern and n the number of states of the generated NFA. For each unit, the additional cost is linear in the average number of active states $\varnothing|S|$ plus the average number of possible transitions $\varnothing|T|$, which both may be as high as n and p, respectively—assuming that the function *match* is executed in constant time, which is true disregarding the number of time and/or label specifications inside an atom. Consequently, the worst-case runtime complexity amounts to $O(m(n+p))$. However, both n and p do not assume high values, and provided a sensible pattern definition, $\varnothing|S|$ and $\varnothing|T|$ remain below their theoretic maxima.

6.3. Matching with (Complex) Conditions

If a pattern with conditions is processed, the complex ones have to be evaluated if and only if the sequence of atoms matches the symbolic trajectory. Although Algorithm 1 reports whether this occurs, it is impossible to decide whether a condition is true or false as long as the binding of the variables is unknown (see Definition 3.5). For the conditions to be verified, we have to find one binding that fulfills each condition. The approach for the computation of these bindings entails recording a matching history during the execution of Algorithm 1, that is, which unit was matched by which pattern element and which are the candidates for matching the next unit. This is done by applying an adjusted version of the algorithm.

6.3.1. Recording the Matching History. Before the outer loop starts, a two-dimensional array of integer sets A, which is later used to retrieve all possible variable bindings, is initialized with the dimensions $m \times e$, where e is the number of pattern elements (cf. Definition 3.4). In addition, we denote the number of the pattern element containing the atom $t.atom$ as $t.elem$. Now consider line 8 of Algorithm 1. In addition to the command after **then**, as long as $i < m - 1$ holds, we collect the set T' of all feasible transitions from $t.target$, where $t \in T$, and insert $t'.elem$, where $t' \in T'$, into the set $A_{i,t.elem}$; that is,

$$\textbf{if } (i < m-1) \textbf{ then } A_{i,t.elem} = A_{i,t.elem} \cup \{t'.elem | t' \in T', t'.source = t.target, t \in T\}.$$



start: $\{0,1\}$			
	pattern elements		
unit	0	1	2
0	$\{0,1\}$	$\{2\}$	$\emptyset$
1	$\{0,1\}$	$\{2\}$	$\emptyset$
2	$\{0,1\}$	$\{2\}$	$\emptyset$
3	$\{0,1\}$	$\{2\}$	$\emptyset$
4	$\emptyset$	$\emptyset$	$\{-1\}$

Fig. 5. The matching history for $M =$ `babbc` and $P =$ `* [a|b] c` as a graph (left) and as a table (right).

In other words, all possible successive pattern element numbers are collected for each atom matching a unit, so the elements of such a set represent pointers to matching candidates for the next unit. During the final iteration, that is, when i equals $m-1$, the value -1 is stored in $A_{m-1,t.elem}$ if and only if a final state is reached.

For a better understanding of this approach, we present a simple example for which we assume that every unit is expressed by a lowercase letter and an atom is either such a single letter or a `+` or a $*$, the latter two having the same meaning as before. Now consider the symbolic trajectory `babbc`, consisting of five units, and the pattern `X * Y [a|b] Z c`. In the following, we discuss the left-hand side of Figure 5 from top to bottom. Since the first atom is a $*$, the first unit could match any of the elements $*$ and `[a|b]`. Hence, we have connections from the start to the columns of the first two pattern elements in the first row. The first unit is a `b`, so it matches the $*$ and the `b` from the pattern, and possible successive matches are (again) the elements $*$, `[a|b]` (after $*$), and `c` (for `b`), respectively; thus, we have three connections to the second row. This procedure is applied similarly for the remaining rows, and also for the final unit `c`, there are two possible matchings, one with $*$ and one with `c`. However, only the latter leads to a complete matching. The bold arrows show the only possible path representing a complete matching, immediately leading us to a binding of the variables. More exactly, since the first three units match $*$, the variable X is bound to the unit set $\{0,1,2\}$. Similarly, the unit sets $\{3\}$ and $\{4\}$ are associated to Y and Z, respectively.

On the right-hand side of Figure 5, the matching history is displayed in tabular form. The pattern elements and the units are represented by their positions, and the set of numbers inside each cell signifies possibly matching pattern elements of the successive unit. For the final unit, only -1 is stored in case of a match. This way of presentation is close to the actual implementation as a two-dimensional array of integer sets.

Now we return to the continuous example M_0 and P_0, for which in Table III we present the two-dimensional array A_0 that is obtained after executing the adjusted version of Algorithm 1. To ease interpretation of the table, we have added abbreviations for the units (left) as well as for the pattern elements (above table). Entries in the table are only the integer sets, but we have added the abbreviations for the matching units that have led to these entries and will be elements of the bindings to be computed. Hence, Table III now is a combination of the representations of the left and right parts of Figure 5.

The computation cost of the extended version of Algorithm 1 includes the initialization cost for A and the additional transition search; more exactly, the complexity is

Table III. The Matching History for M_0 and P_0 as a Two-Dimensional Array of Integer Sets

Start: {0, 1}				
		Pattern Elements		
		`X *`	`Y [({th, m} QA) | (_ WelSt)]+`	`Z [()]?`
	Unit	0	1	2
QA	0	QA	QA	
		{0, 1}	{1, 2}	∅
Wim	1	Wim		
		{0, 1}	∅	∅
WelW	2	WelW		
		{0, 1}	∅	∅
WelSt	3	WelSt	WelSt	
		{0, 1}	{1, 2}	∅
QA	4	QA	QA	QA
		∅	{−1}	{−1}

$O(mp + m(n + p^2)) = O(m(p + n + p^2))$ for the worst case. Thus, the runtime is still linear in the trajectory size.

6.3.2. Computation of Bindings. A binding is implemented as a mapping from a string (i.e., a variable) to a pair of integers (representing the start and the end of a sequence of units). In the following, we show how to deduce bindings from the recently computed two-dimensional array A. As our objective is to find one binding fulfilling every condition—not necessarily all bindings—the computation is aborted in case of success. Consider Algorithm 2. For each pattern element j that enables a transition t starting at state 0 (i.e., $t \in \delta_0$), Algorithm 3 is executed, receiving the parameters P, 0, j, and B where the latter is still empty. As soon as a suitable binding is found and the condition evaluation is completely processed, *true* is returned. Details concerning the evaluation of conditions are discussed later.

ALGORITHM 2: *bindingExistsFrame*

Input: P – a pattern with e elements, including an NFA δ;
B – an empty binding, i.e., a mapping from a string to a pair of integers
Output: *true*, if and only if a binding is found that fulfills every condition;

```
1 foreach j ∈ {t.elem | t ∈ δ0} do
2     if bindingExists(P, 0, j, B) then return true;        // abort if successful
3 return false;
```

Algorithm 3 starts at a certain position in the two-dimensional array A and recursively tries to find a path through A that leads to a final state. If a condition-fulfilling binding is found (line 9), the process is aborted. At the beginning, if the current pattern element is assigned a variable v, either the latter is added to B if it does not yet occur in B (in this case, v is bound only to the current unit; line 6) or the binding of v is extended by one (line 4). The recursion is processed in lines 11 and 12, where the successive unit number $i + 1$, the following pattern element number k (according to the transition), and the adjusted binding B are passed. The aforementioned binding modifications are undone if no success is reported (lines 13–15).

Subsequently, we apply this procedure to our continuous example. In Table IV, each line corresponds to an invocation of Algorithm 3. The recursion depth is represented by i, the number of the current unit. The current pattern element is referred to by j. Note

Table IV. Execution of Algorithm 3 for P_0 and M_0

i	j	B at Call	B Updated	Final	Proceed	Undo B
0	1	$\emptyset$	$\{Y \mapsto [0,0]\}$	no	yes	no
1	2	$\{Y \mapsto [0,0]\}$	$\{Y \mapsto [0,0], Z \mapsto [1,1]\}$	no	no	yes
1	1	$\{Y \mapsto [0,0]\}$	$\{Y \mapsto [0,1]\}$	no	no	yes
0	0	$\emptyset$	$\{X \mapsto [0,0]\}$	no	yes	no
1	1	$\{X \mapsto [0,0]\}$	$\{X \mapsto [0,0], Y \mapsto [1,1]\}$	no	no	yes
1	0	$\{X \mapsto [0,0]\}$	$\{X \mapsto [0,1]\}$	no	yes	no
2	1	$\{X \mapsto [0,1]\}$	$\{X \mapsto [0,1], Y \mapsto [2,2]\}$	no	no	yes
2	0	$\{X \mapsto [0,1]\}$	$\{X \mapsto [0,2]\}$	no	yes	no
3	1	$\{X \mapsto [0,2]\}$	$\{X \mapsto [0,2], Y \mapsto [3,3]\}$	no	yes	no
4	2	$\{X \mapsto [0,2], Y \mapsto [3,3]\}$	$\{X \mapsto [0,2], Y \mapsto [3,3], Z \mapsto [4,4]\}$	yes	no	no

ALGORITHM 3: *bindingExists*

Input: P – a pattern with e elements;
i – the current unit number;
j – the current pattern element number;
B – a binding, i.e., a mapping from a string to a pair of integers.
Output: *true*, if a complete binding fulfilling every condition is found starting from unit i and atom j; *false* otherwise.

```
1  v ← p_elem(j).getVar();
2  inserted ← false;
3  if ¬(v is empty) then
4  |  if B contains v then B(v).right ← B(v).right + 1;        // extend existing binding
5  |  else                                                     // add new variable to binding
6  |  |  B(v) ← (i, i);
7  |  |  inserted ← true;
8  if {−1} ∈ A_{i,j} then                                      // complete match
9  |  if conditionsMatch(P, B) then return true;              // abort if successful
10 else
11 |  foreach k ∈ A_{i,j} do
12 |  |  if bindingExists(P, i + 1, k, B) then return true;   // abort if successful
13 if ¬(v is empty) then
14 |  if inserted then erase v from B;
15 |  else B(v).right ← B(v).right − 1;
16 return false;
```

that we iterate over each integer set $A_{i,j}$ in decreasing order, since reaching a higher pattern element increases the probability of arriving at a final state.

A complete binding can only be achieved for $i = m-1$ (which is 4 in this example), as shown in the bottom row of Table IV. The next step consists of checking whether this binding fulfills every condition.

As the results of the condition evaluation are unpredictable, we have to determine the computation cost for Algorithm 2 under the assumption that every path through the two-dimensional array A has to be pursued. Let a be the number of paths leading through A; then we obtain a runtime complexity of $O(a \cdot T(C))$, where $T(C)$ is the computation cost of the conditions' evaluation, being detailed in Section 6.3.3. The worst case arises given a maximal number of transitions from each state (e.g., this holds for a pattern of the form $* * \cdots *$) and if all conditions must always be evaluated (i.e., if the last condition is always false and the others are always true). Consequently,

each integer set $A_{i,j}$ contains e elements, so a may increase to $O(m^{e-1})$. This can be established as follows:

If e equals 1, there can be only one path through A. For $e = 2$, the number of possible paths increases to $m + 1$, since there are $m + 1$ ways to bind the first variable (i.e., empty, first unit, first two units, ..., whole trajectory). Consequently, every further pattern element increments the exponent by one, resulting in a total number of paths of $O(m^{e-1})$.

6.3.3. Evaluation of Conditions. In order to prepare the evaluation of conditions, we create a SECONDO operator tree for every condition during the parsing process. Such an operator tree includes one pointer for each expression of a variable and an attribute, the latter determining the type of the pointed data. Valid data types in this context are *label*, *place*, *periods*, *instant*, *bool*, *int*, *labels*, and *places*. Consequently, for the condition `Y.end - X.start < 20 * minute` from P_0, two pointers to *instant* values are required. Each of the operator trees enables SECONDO to verify whether the condition is syntactically and semantically correct and, particularly, whether its result is a Boolean value. For example, the input `A.start = B.label` would be rejected due to incompatible attributes, and `A.card + 3` is invalid since the resulting data type is not *bool*.

In the following, we detail the function *conditionsMatch*, which is invoked in line 9 of Algorithm 3. It loops over the conditions, returning *false* in case of a negative result, and *true* if all conditions are fulfilled. Before a condition can be evaluated, we need to update the data referenced by the condition pointer(s). For each expression of the form $v.attr$ (cf. Definition 3.7), the binding B combined with the symbolic trajectory M provides the appropriate values.

For our continuous example, we consider the end of the time interval of unit 3 (`2013-01-17-09:18:44`) and the start of the time interval of unit 0 (same day, `09:02:30`), according to the binding $\{X \mapsto [0, 2], Y \mapsto [3, 3], Z \mapsto [4, 4]\}$. The *instant* pointers' targets are set to these values, and SECONDO can execute the condition as a query, returning a result of type *bool*. In our case, the result is *true*, since the difference of the two instants is less than 20 minutes, and thus *conditionsMatch* also returns *true*.

Concerning the runtime complexity of the condition evaluation, we observe that it is linear in c, the number of conditions, and in $\varnothing|V_C|$, the average number of $v.attr$ expressions per condition. Moreover, since for the attributes *time* and *labels*, not only one or two but also possibly all units have to be accessed, the computation cost for the assignment of values must be considered linear in m. Consequently, the worst-case runtime for one invocation of *conditionsMatch* is in $O(c \cdot \varnothing|V_C| \cdot m)$. For the average case, however, none of the first two values can be expected to be large, and the factor m is dropped out for most configurations.

6.4. Requirements for a Linear Runtime of Matches

As a whole, the operator **matches** has a runtime of $O(m(p + n + p^2) + cm^{e-1} \cdot \varnothing|V_C| \cdot m) = O(m(p+n+p^2)+cm^e \cdot \varnothing|V_C|)$. Now we analyze the requirements that are necessary for the runtime of **matches** to be linear in m, the size of the symbolic trajectory. Obviously, this is the case if e equals 1 or even 0. However, as we are interested in nontrivial pattern specifications, consider the formula's nonlinear part $cm^{e-1} \cdot \varnothing|V_C| \cdot m$, where m^{e-1} is the maximal number of different bindings and m represents the value assignment cost.

The first option for a linear value is linearizing the number of bindings, which is successful if w, the number of the wildcard and regular expression items $*$ and $+$ occurring in the pattern, is at most two. This is due to the fact that for $w = 0$, a matching can only occur if the number of pattern elements equals m, which is unlikely, whereas $w = 1$ grants a realistic probability for a matching, which then is unique since

the binding of the respective sequence variable depends on the remaining pattern. For $w = 2$, however, we may obtain up to $m + 1$ different bindings. In that case, the use of a *time* or *labels*/*places* attribute in a condition may lead to an evaluation time linear in m, so the total runtime would be quadratic. Hence, if w equals 2, the computation cost is linear only for a restricted set of patterns.

For the second approach, the conditions are freely configurable, while w may only equal 0 or 1, resulting in exactly one binding (or none at all).

As stated in Section 6.2, the computation cost of **matches** is always linear in m for a pattern without conditions.

6.5. Rewriting a Symbolic Trajectory

If the operator **rewrite** is called and the result of Algorithm 1 is positive, our objective is to find every possibility of rewriting the applied symbolic trajectory according to the parsed assignments. Hence, discovering one binding that fulfills the conditions, as done previously, does not suffice; instead, we need to find all bindings satisfying the conditions. Consequently, we apply an adjusted version of that algorithm, which returns the first condition-fulfilling binding, starting from a certain position inside A, the two-dimensional integer set array. As **rewrite** returns a stream of trajectories, the current positions—along with the partial bindings—are pushed on a stack, so the computation can be continued from there.

With this binding, the symbolic trajectory M is rewritten as follows. First, we loop over the assignment objects, each represented by one variable in the results section of the pattern. Inside this loop, we assign new values to the results if necessary, similarly to Section 6.3.3. That is, the assignment objects contain one operator tree for each $:=$ operation, having a pointer for each $v.attr$ expression on the right side of the assignment symbol. If any parts of the necessary information for a result variable are missing in the assignments section, they are collected from the original symbolic trajectory according to the binding. By this means, a new unit is created for each result variable (or a sequence of units, for a sequence variable) and added to the result trajectory M'. After the end of the outer loop, M' is written to the output stream.

We now consider a rewrite operation for M_0 and P_1. The first binding that fulfills the condition is found as in Table IV, while the position data i and j are stored on a stack. For every backtracking action, that is, when i does not increase from one line to the next, the current binding is reset according to Algorithm 3, lines 13 through 15. The binding $\{X \mapsto [0, 2], Y \mapsto [3, 3], Z \mapsto [4, 4]\}$ along with the given assignments results in the following symbolic trajectory:

```
< ( (2013-01-17-09:02:30, 2013-01-17-09:13:48, T, F), "start of trip"),
  ( (2013-01-17-09:13:48, 2013-01-17-09:18:44, T, F), "Welbeck St") >
```

Starting from the bottom of Table IV, backtracking only one level, that is, $i = 3$ and $j = 2$, and choosing the element 1 lead to the next binding $\{X \mapsto [0, 2], Y \mapsto [3, 4]\}$. The corresponding result reads

```
< ( (2013-01-17-09:02:30, 2013-01-17-09:13:48, T, F), "start of trip"),
  ( (2013-01-17-09:13:48, 2013-01-17-09:18:44, T, F), "Welbeck St"),
  ( (2013-01-17-09:18:44, 2013-01-17:09:20:10, T, F), "Queen Anne St") >
```

where the last two units belong to (the sequence variable) Y. By backtracking until $i = 2$, $j = 0$, we obtain the binding $\{X \mapsto [0, 3], Y \mapsto [4, 4]\}$ and the symbolic trajectory

```
< ( (2013-01-17-09:02:30, 2013-01-17-09:13:48, T, F), "start of trip"),
  ( (2013-01-17-09:18:44, 2013-01-17-09:20:10, T, F), "Queen Anne St") >
```

The computation cost for rewriting a trajectory, given a certain binding, is linear in m, since the size of a resulting symbolic trajectory cannot exceed m, and each unit of the result is created in constant time—either a unit m_i from M is copied, or some data from m_i are processed, or m_i is not considered at all. Note that the number of expressions of the form $v.attr$ on the right side of the assignment symbol is regarded as constant. Thus, we obtain a total computation cost of $O(m(p+n+p^2)+cm^{e+1}\cdot\varnothing|V_C|)$ for rewriting a symbolic trajectory.

6.6. Classification of a Symbolic Trajectory

The purpose of the operator **classify** is to distribute a set of symbolic trajectories into not necessarily disjoint subsets that represent categories. Along with the trajectory collection, the user needs to specify a set of patterns (with or without conditions), where each pattern must be annotated with a category description. For example, assume we have three trajectories M_0, M_1, and M_2 of a person, two of them from home to work (M_0 on a Monday, M_1 on a Tuesday) and one from work to a restaurant (M_2 on a Monday). Then we could associate the categories "Monday" to M_0 and M_2, "home to work" to M_0 and M_1, and "leisure trip" to M_2.

First, the operator reads the patterns and stores them along with their categories. Instead of computing a separate NFA function for each pattern, we build one multi-automaton for all patterns. Let $n_0, n_1, \ldots, n_{l-1}$ be the number of states for each of the l patterns. Hence, the multiautomaton has $\sum_{i=0}^{l-1} n_i$ states. During the multi-NFA construction, a mapping from the final states to the respective pattern number is stored.

Subsequently, a modification of Algorithm 1 is applied to the first trajectory of the collection. For a multiple pattern processing, the initial set of active states must contain l values instead of one, namely, $\{0, n_0, n_0 + n_1, \ldots, \sum_{i=0}^{l-2} n_i\}$. After the main loop, the set of active states determines the set of patterns matching the processed symbolic trajectory. For each of the remaining patterns, Algorithm 2 is invoked to check the conditions, and finally, the categories of the accepted patterns are attached to the trajectory. This procedure is repeated for every symbolic trajectory from the collection.

6.7. Applying a Trajectory Index

So far in this section, each of the described algorithms requires every unit of a symbolic trajectory to be considered. In a follow-up paper [Valdés and Güting 2014], we introduce a pattern matching technique supported by two indexes, one for the time intervals of a trajectory collection and one for the labels/places.

We implemented the operator **createtrie,** which processes a relation containing a symbolic trajectory attribute, storing the tuple identifier and unit position of each label/place into a trie. Subsequently, this trie is converted into a persistent structure and can be used as a SECONDO database object. The construction cost for the index is linear in the total number of labels. Similarly, the operator **createunitrtree** expects a stream of tuples with a symbolic trajectory attribute and builds a one-dimensional R-tree, where each time interval (converted into an interval of real numbers) is associated with a precise position inside the collection.

If both indexes are available, the operator **indexmatches** can be applied for an index-supported pattern matching, in order to avoid the linear trajectory scan. The operator executes NFA transitions by looking up pattern contents in the indexes and holds certain information for each trajectory that is still active, so a trajectory scan is not necessary anymore. Hence, its runtime is linear in the number of trajectories and depends on other factors like the pattern selectivity, but it is almost independent from the size of the trajectories.

An additional operator named **indexclassify** calculates the same result as **classify** with the help of the twofold trajectory index. Similarly to the previous paragraph, it avoids linear trajectory scans.

7. EXPERIMENTAL EVALUATION

This section is devoted to a series of SECONDO queries carried out with the operators detailed in the previous section. All experiments were conducted on an AMD Phenom II X6 3.3GHz processor running openSUSE 13.2, with 8GBytes of main memory. From this environment, SECONDO was assigned one processor core and half of the available memory. In the first part, we present runtime graphs of the operators **matches**, **rewrite**, **classify**, and **indexclassify** in order to analyze and visualize the impact of certain parameters on the time consumption. For that purpose, a synthetical dataset was created. The second part details several approaches of executing matching tasks on a more realistic dataset generated with BerlinMOD [Düntgen et al. 2009], a benchmark for spatiotemporal database management systems.

All runtimes were computed by running each query four times and taking the median value of the durations.

7.1. Experiments with Synthetic Datasets

In order to obtain symbolic trajectories with comparable properties (i.e., having certain sizes and labels from a static limited collection with similar repetition frequencies), we decided to generate synthetic data. All symbolic trajectories applied in this section represent random walks. More exactly, the labels correspond to the names of the 12 main districts of the city of Dortmund, Germany. For the first series of experiments, we also created random walks through the 413 counties of Germany, in order to increase the number of different labels in the trajectories. As the time intervals are irrelevant for the runtime, each unit has a random duration between 1 and 24 hours, and each trajectory starts at midnight, January 1, 2015. Such synthetic symbolic trajectories with a user-defined size and even relations containing arbitrarily many of them can be produced by the operators **createml** and **createmlrel**, respectively.

The patterns in this section are listed in Table V. Note that the patterns P0, P1, ..., P9 are applied with the trajectories through Dortmund (12 different labels), while the random walks through the German counties (413 different labels) are queried with P0', ..., P5'.

*7.1.1. Operator **Matches**.* In Figure 6, we present the performance evaluation of the operator **matches** with respect to symbolic trajectories of increasing length, several patterns, and different numbers of possible labels per trajectory. The two upper diagrams refer to patterns without conditions, while the bottom graphs resulted from patterns that contains at least one condition. The results on the left-hand side are based on symbolic trajectories with 12 different labels, and the trajectories we applied for the right-hand side diagrams have 413 different labels. Note that we did not add an experiment with more trajectories for this operator, since the effect is a linear dependency as for **rewrite**, in Figure 7 (right).

First, we consider the two diagrams on top of Figure 6, both visualizing that the runtime of the operator **matches** is linear in m if no conditions are specified, according to our computations concerning Algorithm 1. Since the patterns P0 and P0' cause an early mismatch (at the third unit, at the latest), the computation cost is constant. For P1 and P2 as well as for P1' and P2', we observe a difference in the slopes, which is because P1 and P1' have two wildcard atoms, so more transitions are executed after every trajectory unit. The runtime for patterns without conditions is hardly affected if the number of different labels is changed. We notice that the runtime for P1' is

Table V. These Patterns Were Applied for the Experimental Evaluation

```
P0 :  (2015-01-01 "Innenstadt-Ost") (_ "Hörde")
P0':  (2015-01-01 "Mülheim an der Ruhr") (_ "Duisburg")
P1 :  * [(february "Hombruch") | (january "Eving")] *
P1':  * [(february "Hamburg") | (january "Hannover")] *
P2 :  * (2014 "Scharnhorst")
P2':  * (2014 "Freiburg")
P3 :  X * Y (2015-01 "Innenstadt-West", "Innenstadt-Nord", "Innenstadt-Ost") Z *
      // (day_of(Y.start) mod 3) = 0
P3':  X * Y (2015-01 "Kassel", "Schwalm-Eder-Kreis", "Hersfeld-Rotenburg") Z *
      // (day_of(Y.start) mod 3) = 0
P4 :  () A + B [(_ "Lütgendortmund") | (_ "Eving") (_ "Mengede")]+ C *
      // A.card > C.card, get_duration(B.time) > duration(1 0)
P4':  () A + B [(_ "Berlin") | (_ "Stuttgart") (_ "Ludwigsburg")]+ C *
      // A.card > C.card, get_duration(B.time) > duration(1 0)
P5 :  X + Y (_ "Innenstadt-Ost") * (_ "Hörde") Z + // X.start > Z.end
P5':  X + Y (_ "Frankfurt am Main") * (_ "Frankfurt (Oder)") Z + // X.start > Z.end
P6 :  X () Y * Z () => Z // Z.start := Y.start
P7 :  A * B * => A X // X.time := B.time, X.label := "begin of trip"
P8 :  G * H (_ "Eving") I * J (_ "Hörde") K * => H
P9 :  A * B [(_ "Brackel") | (_ "Hombruch")] C * => B
```

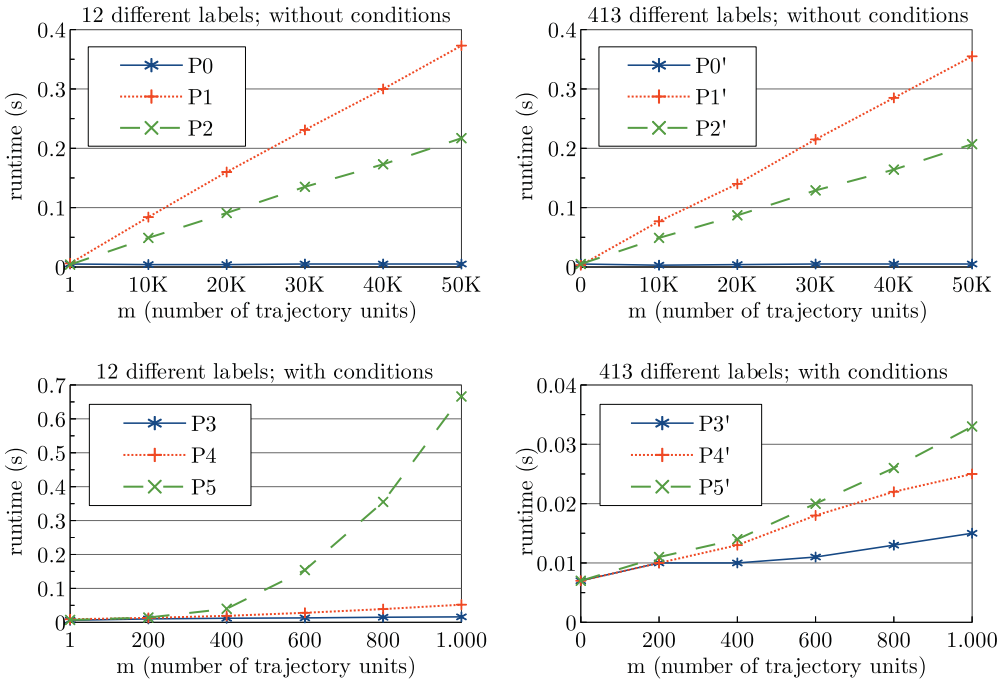


Fig. 6. Runtime of the operator **matches** for 12 (left) and 413 (right) different labels, with (bottom) and without (top) conditions.

slightly faster than for P1, which is because the second pattern element matches more trajectory units if there are fewer different labels in the trajectory, so more transitions have to be conducted.

The graph for pattern P5 in the bottom left diagram confirms the worst-case runtime complexity of Algorithm 2. Since the condition of P5 is never fulfilled, every possible

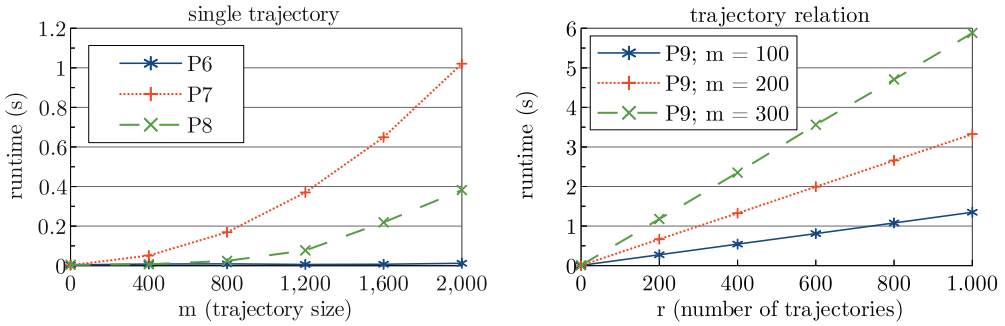


Fig. 7. Runtime of the operator **rewrite** for a single trajectory (left) and a trajectory relation (right, conducted with pattern P9).

binding has to be computed and evaluated, and the runtime is quadratic in m since P5 contains three wildcards. Also for P5' (bottom right diagram), the condition is always evaluated to false, but due to the higher number of different labels, there are fewer possible bindings, so the runtime is still quadratic but clearly lower than for P5. Concerning the patterns P3, P3', P4, and P4', the runtime is linear in m. Among these four patterns, the computation cost is lower for P3 and P3', since their condition is an easy one. The effect of the number of different labels is minor for these patterns but still perceivable.

*7.1.2. Operator **Rewrite**.* The subsequent test series is conducted with the operator **rewrite** and analyzes the runtime for processing single trajectories of different sizes as well as trajectory relations with different numbers of tuples. The corresponding results are depicted in the left and in the right diagram, respectively.

As expected, the graph resulting from the pattern P8 in the left-hand diagram of Figure 7 has a quadratic shape due to the three wildcards. Although the pattern P7 contains only two wildcards, we obtain a quadratic runtime for P7 too, since there are no filtering elements and one of the sequence variables (A) also occurs in the results section. Due to a higher number of result trajectories (i.e., $m + 1$ for P7 compared to less than $\frac{m}{20}$ for P8) induced by the filtering unit patterns inside P8, the graph of P7 shows a higher slope. Finally, as there is only one wildcard in P6 and no sequence variable in the results section, the computation cost for the **rewrite** operation is linear in m.

On the right-hand side, the runtime behavior of **rewrite** is depicted for relations containing an *mlabel* attribute. We varied the number r of tuples of the relations—along the abscissa—and the size of the trajectories, while the applied pattern set, consisting of the pattern P9, remained invariant. Inside a relation, all trajectories have the same number of units. Unsurprisingly, the computation cost is proportional to r, apart from a fractional parsing overhead. Concerning the different trajectory sizes, the rise is quadratic in m because of the two wildcards, similar to the graph of P7 on the left.

*7.1.3. Operators **Classify** and **Indexclassify**.* The final test, whose results are depicted in Figure 8, reviews the performance of the operator **classify** with regard to the quantity and size of the examined symbolic trajectories. Again, the efficiency is optimized with the help of an index. The classification task is conducted with the three patterns `(_ "Hörde") * (_ "Brackel")`, `(_ "Innenstadt-West") (_ "Lütgendortmund") *`, and `* (_"Aplerbeck") * (_"Lütgendortmund") * (_"Eving") *`.

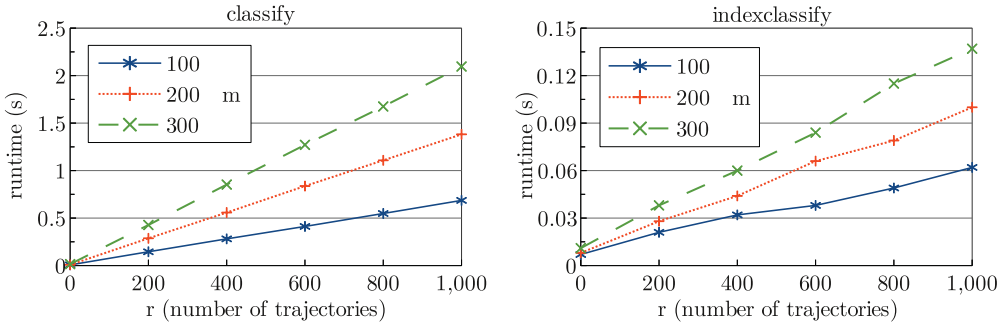


Fig. 8. Runtime of the operator **classify.**

From the left plot, we deduce that the runtime function of the **classify** operator is nearly proportional to the number of trajectories as well as to their sizes, which could be expected, since only Algorithm 1 is executed. Hence, the construction of the multiautomaton, being independent from the set of moving labels, is efficient. In fact, the operator consumes approximately 7 microseconds for processing one unit (for this pattern combination) and an overhead of a few milliseconds for the automaton.

Applying the trajectory relation index, we are able to reduce the runtime to a high extent, that is, by a factor of approximately 15. The runtime is linear in r and less than proportional to m.

7.2. Experiments with BerlinMOD Data

In order to obtain a more realistic dataset, we applied the database benchmark system BerlinMOD with a scale factor of 1.0. By this means, a relation with 293,000 tuples, each containing an *mpoint* attribute, was created. These trips, consisting of 56 million *upoint*s (point units) in total, refer to the movement data of 2,000 objects collected during a period of 28 days inside the city of Berlin. Since half of the moving points are stationary (i.e., they contain only one point unit), we removed them and conducted the experiments with the remaining 145,000 trips.

In the following, we present six different approaches of solving a certain task on these trajectories. While the first one processes the raw *mpoint* data and the second one applies network-constrained data [Güting et al. 2006], the remaining four are based on symbolic trajectories. More exactly, each of the subsequent methods is applied to identify the trajectories passing the street Bundesallee before passing at least one of the three streets Leipziger Str., Hohenzollerndamm, and Steglitzer Damm. In addition, the duration of the trip segment between Bundesallee and one of its three successors must be longer than the time spent after the latter. This query has a selectivity of approximately 1%. We assume that for all six alternatives, the different types of trajectories, that is, raw, network constrained, and symbolic, are present. The symbolic trajectories are clearly shorter than the raw trips, since for the former, an additional unit is necessary only if the street changes. On the other hand, a unit of the raw trajectories of BerlinMOD has a duration of 2 seconds. To be precise, the memory consumption of the symbolic trajectories amounts to 282MBytes, whereas the associated raw trajectories require 10.8GBytes, as well as the network-constrained data.

The executed queries as well as corresponding explanations can be obtained from Appendix B. A summary of the index construction times and the runtimes is presented in Table VI. Note that spatiotemporal pattern queries [Sakr and Güting 2011] are not suitable for this evaluation, since the specified condition cannot be expressed. We provide an overview of the applied methods:

Table VI. Result Overview for Processing 145,000 BerlinMOD Trajectories

Applied Approach	Runtime (Seconds) for Building Index(es)	Query
Based on raw geometric data (10.8GBytes)		
1 : Raw Trajectories and Geometric Indexes	2,249.72	1,276.41
Based on network-constrained moving objects (10.8GBytes)		
2 : Network-Constrained Moving Points	3,870.03	180.13
Based on symbolic trajectories (282 MBytes)		
3 : Linear Scan Without Pattern Language		10.06
4 : Pattern Matching		10.96
5 : Index Support Without Pattern Language	213.91	0.58
6 : Index-Supported Pattern Matching	213.91	0.59

(1) *Raw Trajectories and Geometric Indexes*: We created two B-trees over the streets relation (for the routes and their bounding boxes) and an R-tree over the trajectory units. The result was computed via index results and auxiliary functions.
(2) *Network-Constrained Moving Points*: We built a B-tree over the street network and an R-tree over the network trajectory units. With the help of index results and additional functions, we obtained the result.
(3) *Linear Scan on Symbolic Trajectories Without Pattern Language*: Again, auxiliary functions are required. We performed a linear scan of all symbolic trajectories and applied several filters.
(4) *Pattern Matching on Symbolic Trajectories*: The task was translated into a pattern with condition, so we could use one filter and the **matches** operator. No further functions or objects are necessary.
(5) *Exploiting a Symbolic Trajectory Index Without Pattern Support*: First, two indexes for symbolic trajectories were created. Then we conducted a join of index results for the specified streets and applied additional functions.
(6) *Index-Supported Pattern Matching on Symbolic Trajectories*: The operator **index-matches** was executed with the same pattern as for Approach 4.

7.2.1. Summary of BerlinMOD Experiments. According to Table VI, using symbolic trajectories instead of their raw or network-constrained counterparts offers substantial advantages regarding the execution of queries as well as the disk space occupation and the creation of indexes. For all these parameters, gains in efficiency of several orders of magnitude can be observed, especially for the index-supported approaches.

Among the methods based on symbolic trajectories, the techniques without index (3, 4) and the index-based methods (5, 6) should be considered separately. In both cases, the pattern-assisted version is slightly less efficient. However, for the approaches without pattern support (3, 5), the formulation of queries is much more difficult and lengthy, and SECONDO expert knowledge is required to understand them (see the queries in Appendix B). At the same time, the pattern language enables the user to express sophisticated tasks as short and elegant patterns, which can be discussed, exchanged, and adjusted without profound database skills.

8. RELATED WORK

The notion of symbolic trajectory relates to diverse research areas. In what follows, we tie in our work with the state of the art, focusing in particular on the following two key features: the symbolic data model and the pattern matching and manipulation language.

Semantic Trajectories. The first stream of related research regards the modeling of semantic trajectories. Broadly speaking, semantic trajectories are geometric

trajectories annotated with supplementary information. Annotations can regard, for example, places, transportation means, and activities. Indeed, semantic trajectories is a broad and lively area of research that has progressively grown over the last years thanks to the contribution and convergence of independent research works, especially those carried out in the projects GeoPKDD [Giannotti and Pedreschi 2008] and Modap [Renso et al. 2013] in Europe, in the project Geolife [Zheng et al. 2010b] in Asia, and in the early work by Liu et al. [2006]. A prominent research direction on semantic trajectories is geared toward the definition of general concepts and methods for the extraction and representation of semantic information. Early work on the conceptual modeling aspects defines a semantic trajectory as a sequence of alternating stops and moves, where a stop is a suspension of the movement at the chosen level of abstraction [Spaccapietra et al. 2008]. Methods for the discovery of stops and moves from geometric trajectories are presented in, for example, Palma et al. [2008], Rocha et al. [2010] and Yan et al. [2011]. In Spaccapietra et al. [2013], a generalized conceptualization subsuming the stop-and-move model is presented based on the notion of *episode*, denoting a portion of trajectory that is semantically homogeneous, for example, the movement in proximity of a point of interest. This notion is at the basis of the semantic trajectory discovery process presented in Yan et al. [2013] and Yan and Chakraborty [2014]. The notion of semantic trajectory is much richer than that of symbolic trajectory; however, that definition is not directly usable or sufficiently detailed as a database model because it is a mere conceptualization that does not specify how to access and query semantic trajectories. A possible approach to the definition of a semantic trajectory database model relying on the use of data types is outlined in Pelekis and Theodoridis [2014]. We are not aware, however, of any further specification and implementation.

A different research direction, orthogonal to the construction of a generic trajectory database, investigates efficient access and query processing methods for narrower classes of trajectories. For example, the work in Zhang et al. [2014] defines a semantic trajectory as a sequence of timestamped places wherein a place is described by a spatial location and a semantic label (e.g., office). Semantic trajectories can represent, for example , the sequence of users' checkins in Foursquare and Facebook Places. The research problem is to extract from a database frequent sequential patterns where a sequential pattern is a sequence of temporally bounded transitions from one or more places to another group of places. Places are also the components of the *activity trajectories* [Zheng et al. 2013] defined as sequence of spatial points with an associated number of activities (e.g., shopping). The research problem is to efficiently extract the n trajectories that best match a set of activities, based on a specific notion of matching. A similar type of query relying on a slightly different notion of matching is discussed in Cong et al. [2012]. The notion of place is also available in symbolic trajectories, where we can specify not only the spatial object associated with the label but also the period or duration of the stay. Moreover, matching criteria on the textual and spatial components can be flexibly defined using conditions.

Pattern Matching and Manipulation Language. Pattern matching languages for querying sequential data in relational databases have been proposed in, for example, Sadri et al. [2004] and Agrawal et al. [2008]. The Simple Query Language for Time Series (SQL-TS) [Sadri et al. 2004] adds to SQL constructs for specifying patterns as sequences of variables, for example, (X, Y, Z), that are bound to tuples or series of tuples satisfying conditions on the attributes, for example, $X.att > Y.att$. Tuples are ordered based on the value of an attribute. Whenever the ordering is based on time, the notion of sequence data is similar to that of time series, described next.

Time series are sequences of data points (often real numbers) spaced at strictly increasing times [Gao and Wang 2009] that can be seen as vectors of a high-dimensional space. Time intervals between data points are usually, but not always, of equal size. The main query type is similarity search (matching either the whole or a subsequence) based on many different kinds of distance measures [Faloutsos et al. 1994; Ding et al. 2008]. Compression of huge time-series data warehouses is also a major concern [Korn et al. 1997]. Even for symbolic representations such as SAX (Symbolic Aggregate ApproXimation), their lower bounding properties for distance (i.e., similarity) computations are found essential [Ding et al. 2008].

In contrast, in a symbolic trajectory, symbolic values are associated with time intervals, not instants of time. Time intervals have widely varying size, depending on how long the property holds. There may be gaps, which is not a problem as in time series but is the normal case (consider the example of roe deer in Section 4.2). The symbolic values are semantically meaningful; therefore, it makes sense to mention particular values in queries (e.g., transitions from train to taxi in transportation modes, see Section 4.1). This is not the case in time series. Moreover, the symbolic trajectories can handle places, which is obviously not available in time series. All of this leads to very different requirements in querying.

Mobility pattern matching in spatiotemporal databases is the research line more closely related to our work. The basis for the present work originates from du Mouza and Rigaux [2005]. In particular, du Mouza and Rigaux introduce the notion of mobility pattern for moving objects relative to a hierarchical partitioning of the plane into regions, where each region is identified by a symbol. An object's trajectory is defined by the sequence of symbols denoting the successive regions crossed by the object. The trajectories obtained in this way can be interrogated using a pattern matching language supporting variables. The expressiveness of the language is, however, limited. In particular, variables can be only bound to symbols and not to time; moreover, the language does not allow the specification of conditions. The sequence of region labels does not include time information, so the symbolic model is not suitable for trajectory annotation.

The assumption of a space partitioned into regions is also at the basis of the work by Vieira et al. [2010, 2011]. The idea is again to support pattern matching over trajectories, but in this case trajectories are geometric and not symbolic; that is, a sample query is to find the trajectories that go from region A to region B. The sequence of region labels with timestamps occurs only as a data structure within the implementation. The pattern language is rich and includes not only symbols and variables but also conditions, such as spatial and temporal conditions. Technically, the use of variables is different from our approach as here variables represent regions, whereas in our approach they represent units or subsequences of a symbolic trajectory. While their use of variables allows for an easy specification of the same region being visited twice or repeatedly (just use the same variable), it is in general less expressive than our approach. For example, it is not possible to access attributes of variables describing time intervals of a subsequence or the number of intermediate symbols. Again, this approach is restricted to partitioned space and does not offer a general solution for trajectory annotations. A precursor to the work of Vieira et al. [2010, 2011] is Hadjieleftheriou et al. [2005]. Here, more general geometries are considered than just partitions of the plane into regions. On the other hand, this approach focuses on range and nearest-neighbor queries, does not introduce any symbolic representation, and does not have variables or regular expressions and so is only remotely related.

Our pattern manipulation language does not impose any constraint on the possible annotations. It allows the specification of mobility patterns in terms of regular expressions with variables denoting symbols, time intervals, and subsequences. Moreover,

the language is embedded into the SECONDO platform, which offers a rich and extensible repertoire of data types that can be used for formulating a variety of conditions on pattern variables. Further, we recall that the language not only supports pattern matching but also provides two additional operators: *classify* to categorize trajectory sets through multipattern matching and *rewrite* to let users extract and even change trajectory labels in order to enrich the description with further information. To the best of our knowledge, the expressiveness, flexibility, and variety of operations provided by our query language are unrivaled.

9. CONCLUSIONS AND FUTURE WORK

In this article, we have proposed annotated trajectory databases as a new research direction, to study representation and querying of semantic trajectories from a database perspective. As a first step, we have defined a data model for symbolic annotations (or symbolic trajectories manipulated independently) in the form of four abstract data types to represent time-dependent (sets of) labels and places. They can represent in a generic way any relevant annotation known from the literature.

The new data types are similar to data types for moving objects that have been proposed before (e.g., $\underline{mpoint} = \underline{moving}(\underline{point})$) [Güting et al. 2000], so they exhibit "little novelty." This is not a bug but a feature of the approach. It would be easy to develop new models from scratch, but this would be an entirely bad idea as they would require their own system implementation and fit together with nothing else. In contrast, we have carefully integrated the new types into the framework of Güting et al. [2000] and into the SECONDO system implementation; they could be equally easily integrated into the other moving objects DBMS and Hermes. As a result, the user can query symbolic trajectories with the same mechanisms as raw trajectories or their related data types.

Moreover, a pattern-based language has been proposed to retrieve symbolic or annotated trajectories. Whereas previous languages of this kind were restricted to the one special case of regions traversed, it is now possible to use pattern-based queries on generic annotations of any kind and so retrieve annotated trajectories based on any kind of symbolic features. Furthermore, the pattern-based language treats the temporal dimension in full generality. It provides rewriting, needed to retrieve matching parts, and classification, not present in earlier work. The pattern-based query language has been rigorously defined with respect to syntax and semantics, and an efficient implementation in a DBMS context has been provided, as demonstrated in a comprehensive set of experiments.

The complete implementation of model and language is publicly available with a new release of SECONDO and so can be used for practical applications.

Annotated trajectory databases offer a lot of exciting research possibilities, for example:

—*Querying hybrid and multidimensional trajectories.* By hybrid trajectories we mean pairs consisting of one geometric and one symbolic trajectory, and by multidimensional trajectories objects (tuples) having one geometric and several symbolic (or numeric, e.g., altitude) trajectories. The pattern language so far is restricted to the symbolic dimension. It would be interesting to develop pattern-based mechanisms working across several dimensions.

—*Query optimization in moving objects databases, using symbolic trajectories.* Symbolic trajectories could in fact be used in a way entirely transparent to the user, just as a tool for query optimization. For example, suppose in a vehicle database there are frequent queries to retrieve vehicles that went at some particular speed, for example, faster than 130km/h. Similar to creating an index, the database administrator might

derive symbolic trajectories with a speed classification (e.g., slow, moderate, fast, very fast). The query optimizer may then translate the speed query to a filter and refine the strategy, retrieving first symbolic trajectories with entries fast or very fast.
—*Grouping/clustering by symbolic trajectories (equality).* Clustering so far tries to put together geometric trajectories that are in some way similar. A new possibility to make trajectories similar is to let them have the same symbolic representation.
—*Similarity of symbolic trajectories.* Furthermore, similarity measures for symbolic trajectories (including place annotations) may be studied.
—*Join over symbolic trajectories.* Efficient join techniques may be developed based on equality or similarity.
—*Benchmarks.* Design scalable benchmark data generators and queries.
—*Applications and datasets.* Make real datasets available for research and study related applications.

In summary, we feel annotated trajectory databases and symbolic trajectories can be a major step forward in the handling of semantics of trajectories and offer exciting research prospects in the future.

APPENDIX

A. FORMALIZATION OF REWRITE RULES AND THEIR SEMANTICS

This appendix is provided in electronic form.

B. DETAILS OF THE EXPERIMENTAL EVALUATION

In this section, the queries for the experiments from Section 7.2 are listed. As seen in Section 4, pattern matching queries on symbolic trajectories can easily be expressed in SQL. However, the other approaches require sophisticated commands that can only be conducted in SECONDO executable language. Therefore, we decided to use the latter for all queries, in order to enable a fair comparison.

Approach 1: Raw Trajectories and Geometric Indexes

First, we create a relation `strassen2` containing the names and the bounding boxes of all streets from the original relation `strassen`.

```
let strassen2 = strassen feed sortby[Name]
  groupby[Name; Bbox: bbox(group feed projecttransformstream[GeoData] collect_line[TRUE])]
  consume
```

Over the most recently created relation, we build a B-tree for faster access to the bounding box of a street.

```
let strassen2_btree = strassen2 feed addid createbtree[Name]
```

Since some of the streets in the `strassen` relation are separated in different lines, we unite them with the operator **collect_line** and store them in a new relation.

```
let strassen_united = strassen feed sortby[Name]
  groupby[Name; Line: group feed projecttransformstream[GeoData] collect_line[TRUE]]
  consume
```

We create another B-tree to enable fast access to the precise course of any street.

```
let strassen_btree = strassen_united feed addid createbtree[Name]
```

The subsequent function returns the bounding box of the course of the street that corresponds to the specified street name. Due to the B-tree, this function is very efficient.

```
let getbboxfromname = fun(SName: string)
  strassen2_btree strassen2 exactmatch[SName] extract[Bbox]
```

We create an R-tree containing the bounding boxes of all units of the 145,000 raw trajectories. The operator **units** transforms an *mpoint* into a stream of *upoint*s. Applying the **bbox** operator to such a unit yields a three-dimensional cuboid (two spatial and one temporal dimension), and with **rectproject**, we keep only the spatial information.

```
let units_rtree_2d = SymTrips feed addid projectextendstream[TID; Bbox: units(.MP)]
  replaceAttr[Bbox: rectproject(bbox(.Bbox), 1, 2)] sortby[Bbox] bulkloadrtree[Bbox]
```

The following function checks whether an *mpoint* passes a certain street. First, the trajectory is transformed into a stream of units. The start and end point of every unit are derived by the operators **initial** and **final** (extracting an *ipoint* value, i.e., an *instant* along with a *point*) and **val** (reducing an *ipoint* to its *point*). Finally, the function checks whether there is a unit whose start and end point have a distance of less than 30cm to the *line* that belongs to the specified street name. The corresponding line is returned from the B-tree.

```
let filterendpoints = fun(Trip: mpoint, Name: string)
  units(Trip) transformstream
  projectextend[; Sp: val(initial(.Elem)), Ep: val(final(.Elem))]
  filter[distance(.Sp, strassen_btree strassen_united exactmatch[Name] extract[Line]) < 0.03]
  filter[distance(.Ep, strassen_btree strassen_united exactmatch[Name] extract[Line]) < 0.03]
  head[1] count > 0
```

Similarly to the previous function, we define another function returning the position of the first occurrence of a certain street name in an *mpoint*. If the street is not passed by the *mpoint*, the result is undefined.

```
let firstposMP = fun(Trip: mpoint, Name: string)
  units(Trip) transformstream addcounter[No, 1]
  filter[tostring(val(initial(.Elem))) = Name]
  filter[distance(.Sp, strassen_btree strassen_united exactmatch[Name] extract[Line]) < 0.03]
  filter[distance(.Ep, strassen_btree strassen_united exactmatch[Name] extract[Line]) < 0.03]
  extract[No]
```

The subsequent function yields the last occurrence of a specified street name in an *mpoint*, if existing, and an undefined value otherwise.

```
let lastposMP = fun(Trip: mpoint, Name: string)
  units(Trip) transformstream addcounter[No, 1]
  extend[Sp: val(initial(.Elem)), Ep: val(final(.Elem))]
  filter[distance(.Sp, strassen_btree strassen_united exactmatch[Name] extract[Line]) < 0.03]
  filter[distance(.Ep, strassen_btree strassen_united exactmatch[Name] extract[Line]) < 0.03]
  tail[1] extract[No]
```

This function verifies whether a certain unit of an *mpoint* (retrieved by the operator **getunit**) occurs completely on a specified weekday. Precisely, we check the weekdays of the first and the last instant of the unit, and in addition, the duration of the unit has to be less than 1 day (the **duration** operator is invoked with a number of days and a number of milliseconds).

```
let checkWeekdayMP = fun(Trip: mpoint, Pos: int, Day: string)
  (weekday_of(inst(initial(getunit(Trip, Pos)))) = Day) and
```

```
(weekday_of(inst(final(getunit(Trip, Pos)))) = Day) and
((inst(final(getunit(Trip, Pos))) - inst(initial(getunit(Trip, Pos))))< (duration (1 0)))
```

The last function of this paragraph returns a Boolean value indicating whether the temporal duration spent between two units of a certain *mpoint* (given by their two positions) is longer than the time interval that remains after the second given position. The second duration is computed in lines 2 and 3: if the given position represents the end of the trajectory, the duration is 0. Otherwise, it equals the difference between the final instant of the trajectory and the first instant of the first unit behind the given position. The lines 4 and 5 yield the duration of the first time interval. Either it is 0 (if both units are adjacent) or it equals the difference between the last instant of the predecessor of the second unit and the initial instant of the successor of the first unit. Finally, both durations are compared by the $<$ operator.

```
let matchesTempCondMP = fun(Trip : mpoint, BY : int, BZ : int)
  ifthenelse(no_components(Trip) = BZ, duration(0 0),
    inst(final(getunit(Trip, no_components(Trip)))) - inst(initial(getunit(Trip, BZ + 1))))
  < ifthenelse(BZ = (BY + 1), duration(0 0),
    inst(final(getunit(Trip, BZ - 1))) - inst(initial(getunit(Trip, BY + 1))))
```

In the main query, we first retrieve all trajectories having a unit whose bounding box intersects the bounding box of the street Bundesallee. In the second line, we apply the more precise filter step by endpoints. Then we compute the final position of Bundesallee and perform the weekday check. This stream of tuples is named `ba` for a later join operation. Subsequently, we retrieve the first occurrences of the three other streets, concat the resulting tuples, and name them `nx`. With a hashjoin, both streams are connected. At the end, we check whether the position of Bundesallee precedes the position of the other streets, and we verify the temporal condition.

```
query units_rtree_2d windowintersectsS[getbboxfromname("Bundesallee")]
  sortby[Id] rdup Trips gettuples filter[filterendpoints(.MP, "Bundesallee")]
  projectextend[Tripid, MP; BApos: lastposMP(.MP, "Bundesallee")]
  filter[checkWeekdayMP(.MP, .BApos, "Wednesday")]{ba}
  (units_rtree_2d windowintersectsS[getbboxfromname("Leipziger Str.")]
  sortby[Id] rdup Trips gettuples filter[filterendpoints(.MP, "Leipziger Str.")]
  projectextend[Tripid, MP; NXpos: firstposMP(.MP, "Leipziger Str.")]
  units_rtree_2d windowintersectsS[getbboxfromname("Hohenzollerndamm")]
  sortby[Id] rdup Trips gettuples filter[filterendpoints(.MP, "Hohenzollerndamm")]
  projectextend[Tripid, MP; NXpos: firstposMP(.MP, "Hohenzollerndamm")] concat
  units_rtree_2d windowintersectsS[getbboxfromname("Steglitzer Damm")]
  sortby[Id] rdup Trips gettuples filter[filterendpoints(.MP, Steglitzer Damm")]
  projectextend[Tripid, MP; NXpos: firstposMP(.MP, "Steglitzer Damm")] concat) {nx}
  hashjoin[Tripid_ba, Tripid_nx]
  projectextend[; MP: .MP_ba, BApos: .BApos_ba, NXpos: .NXpos_nx]
  filter[.BApos < .NXpos]
  filter[matchesTempCondMP(.MP, .BApos, .NXpos)] count
```

Runtime for final query: 21 minutes, 16 seconds.

Approach 2: Network-Constrained Moving Points

The operator **tonetwork** converts a raw trajectory (type *mpoint*) into a network-constrained trajectory (type *mgpoint*).

```
let TripsJNet = Trips feed projectextend[Tripid; Pos: tonetwork(JBNet, .Str)] consume;
```

We build an R-tree over the (spatial) bounding boxes of the network-constrained trajectories.

```
derive TripsRtree = TripsJNet feed extend[TID: tupleid(.)]
  projectextend[TID; Box: rectproject(bbox(.Pos), 1, 2)] sortby[Box asc] bulkloadrtree[Box]
```

The original streets from the street relation are converted into network-constrained *jline*s.

```
let strassen_jline = strassen feed extend[JLine: tonetwork(JBNet, .GeoData)] consume
```

Over the names of the recently created network-constrained streets, we create a B-tree.

```
let strassen_jline_btree = strassen_jline feed addid createbtree[Name]
```

This function retrieves the last position of a street name inside a network-constrained trajectory behind a specified position. More exactly, if the trajectory passes the network-constrained line retrieved from the B-tree, we transform it into units, append a counter to them (operator **addcounter**), and exclude the units that occur before the position `Pos`. Then we create an *mjpoint* from each remaining *ujpoint* and check whether it passes the network-constrained line. Finally, the function returns the maximum remaining unit position or an undefined value if the trajectory does not pass the *jline*.

```
let lastposjnet = fun(Trip: mjpoint, Name: string)
  ifthenelse(Trip passes strassen_jline_btree strassen_jline exactmatch[Name]
    extract[JLine], units(Trip) transformstream addcounter[No, 1]
    projectextend[No; SingleMJP: createmjpoint(.Elem)]
    filter[.SingleMJP passes (strassen_jline_btree strassen_jline exactmatch[Name]
    extract[JLine])] tail[1] extract[No], [const int value undef])
```

Similarly to the previous one, the subsequent function computes the first occurrence of a street name in a network-constrained trajectory after a specified position.

```
let firstposafterjnet = fun(Trip: mjpoint, Name: string, Pos: int)
  ifthenelse(Trip passes strassen_jline_btree strassen_jline exactmatch[Name]
    extract[JLine], units(Trip) transformstream addcounter[No, 1]
    filter[.No > Pos] projectextend[No; SingleMJP: createmjpoint(.Elem)]
    filter[.SingleMJP passes (strassen_jline_btree strassen_jline exactmatch[Name]
      extract[JLine])] extract[No], [const int value undef])
```

As in the previous approach, we define a function that checks whether a certain unit of a network-constrained trajectory occurs on a specified weekday.

```
let checkWeekdayjnet = fun(Trip: mjpoint, Pos: int, Day: string)
  (weekday_of(inst(initial(getunit(Trip, Pos)))) = Day) and
  (weekday_of(inst(final(getunit(Trip, Pos)))) = Day) and
  ((inst(final(getunit(Trip, Pos)))-inst(initial(getunit(Trip, Pos)))) < (duration (10)))
```

This function is also very similar to the function `matchesTempCondMP` from Approach 1.

```
let matchesTempCondjnet = fun(Trip: mjpoint, BY: int, BZ: int)
  ifthenelse(no_components(Trip) = BZ, duration(0 0),
    inst(initial(createmjpoint(getunit(Trip, no_components(Trip)))))
      - inst(initial(createmjpoint(getunit(Trip, BZ + 1)))))
  < ifthenelse(BZ = (BY + 1), duration(0 0),
    inst(initial(createmjpoint(getunit(Trip, BZ - 1))))
      - inst(initial(createmjpoint(getunit(Trip, BY + 2)))))
```

The main query first retrieves all tuples where the *mjpoint* intersects the bounding box of the street Bundesallee and adds the last unit position for it. If the position is undefined, the tuple is removed (line 3). Another filter removes the tuples that do not fulfill the weekday condition. Similarly, we add the first positions of the three other

streets (behind the position of Bundesallee) and keep only the tuples with at least one defined value (line 8). Then we aggregate the three values (from which only one is defined) to one and name the attribute NXpos. Finally, the tuple stream is filtered according to the temporal condition.

```
query TripsRtree windowintersectsS[getbboxfromname("Bundesallee")]
  sortby[Id] rdup TripsJNet gettuples extend[BApos: lastposjnet(.Pos, "Bundesallee")]
  filter[isdefined(.BApos)]
  filter[checkWeekdayjnet(.Pos, .BApos, "Wednesday")]
  extend[LSpos: firstposafterjnet(.Pos, "Leipziger Str.", .BApos),
    HDpos: firstposafterjnet(.Pos, "Hohenzollerndamm", .BApos),
    SDpos: firstposafterjnet(.Pos, "Steglitzer Damm", .BApos)]
  filter[isdefined(.LSpos) or isdefined(.HDpos) or isdefined(.SDpos)]
  projectextend[Tripid, Pos, BApos; NXpos: ifthenelse(isdefined(.LSpos), .LSpos,
    ifthenelse(isdefined(.HDpos), .HDpos, .SDpos))]
  filter[matchesTempCondjnet(.Pos, .BApos, .NXpos)] count
```

Runtime for final query: 180.1 seconds.

Approach 3: Linear Scan on Symbolic Trajectories without Pattern Language

This function retrieves the first occurrence of a street name in a symbolic trajectory (type *mlabel*) after a specified position—if existing—or an undefined value.

```
let firstposafter = fun(ST: mlabel, Name: string, Pos: int)
  units(ST) transformstream addcounter[No, 1]
  filter[tostring(val(initial(.Elem))) = Name]
  filter[.No > Pos] extract[No]
```

The next function verifies whether a certain unit of a symbolic trajectory occurs on a specified weekday. Its semantics is analogous to the respective functions of the previous approaches.

```
let checkWeekday = fun(ST: mlabel, Pos: int, Day: string)
  (weekday_of(inst(initial(getunit(ST, Pos)))) = Day) and
  (weekday_of(inst(final(getunit(ST, Pos)))) = Day) and
  ((inst(final(getunit(ST, Pos))) - inst(initial(getunit(ST, Pos)))) < (duration (1 0)))
```

We also need a function for checking the temporal condition for a symbolic trajectory. For details, please refer to the corresponding functions defined before.

```
let matchesTempCond = fun(ST: mlabel, Pos1: int, Pos2: int)
  ifthenelse(no_components(ST) = Pos2, duration(0 0),
    inst(final(getunit(ST, no_components(ST)))) - inst(initial(getunit(ST, Pos2 + 1))))
  < ifthenelse(Pos2 = (Pos1 + 1), duration(0 0),
    inst(final(getunit(ST, Pos2 - 1))) - inst(initial(getunit(ST, Pos1 + 1))))
```

In the first filter step, we keep only the tuples where the symbolic trajectory passes Bundesallee. Then we extend the tuples by the first position of Bundesallee and filter them according to the weekday condition. The remainder of the query is similar to the main query of Approach 2.

```
query Trips feed filter[.Str passes tolabel("Bundesallee")]
  projectextend[Str; BApos: firstposafter(.Str, "Bundesallee", -1)]
  filter[checkWeekday(.Str, .BApos, "Wednesday")]
  projectextend[Str, BApos;
    LSpos: firstposafter(.Str, "Leipziger Str.", .BApos),
    HDpos: firstposafter(.Str, "Hohenzollerndamm", .BApos),
    SDpos: firstposafter(.Str, "Steglitzer Damm", .BApos)]
  filter[isdefined(.LSpos) or isdefined(.HDpos) or isdefined(.SDpos)]
```

```
projectextend[Str, BApos; NXpos: ifthenelse(isdefined(.LSpos), .LSpos,
  ifthenelse(isdefined(.HDpos), .HDpos, .SDpos))]
filter[matchesTempCond(.Str, .BApos, .NXpos)] count
```

Runtime for final query: 10.06 seconds.

Approach 4: Pattern Matching on Symbolic Trajectories

In the pattern language, the alternative (Leipziger Str., Hohenzollerndamm, or Steglitzer Damm) can be expressed by square brackets and the symbol |. The temporal constraint of passing Bundesallee on a Wednesday is handled inside one atomic pattern element. The sequence of units between Bundesallee and one of the three other streets is associated to the variable X, and the remainder of the symbolic trajectory after passing one of the three streets is named Y. Hence, we can formulate the condition including these two variables, which is fulfilled if and only if the temporal duration of the unit sequence belonging to X is greater than the temporal duration of the units associated with Y.

```
query Trips feed filter[.Str matches '* (wednesday "Bundesallee") X *
  [(_ "Leipziger Str.") | (\_ "Hohenzollerndamm") | (\_ "Steglitzer Damm")] Y *
  // get_duration(X.time) > get_duration(Y.time)'] count
```

Runtime: 10.96 seconds.

Approach 5: Exploiting a Symbolic Trajectory Index Without Pattern Support

In this approach, we use a trie `TripsStrTrie` containing all labels from the collection of symbolic trajectories along with their exact position in the collection (i.e., tuple id and unit position). Queried with the operator **searchWord**, the trie yields all occurrences of the street Bundesallee. We append the corresponding tuples (operator **gettuples2**) and check whether Bundesallee was passed on a Wednesday. Then we retrieve all index results for the three other streets and combine them with the Bundesallee results via a hashjoin, where the position of Bundesallee must be smaller than the other position (line 7). Finally, we keep only the required attributes and exclude the tuples that do not fulfill the temporal condition.

Note that the auxiliary functions `checkWeekday` and `matchesTempCond`, defined in Approach 3, are reused here. Hence, this approach is more cumbersome than it appears.

```
query TripsStrTrie searchWord["Bundesallee"]
  projectextend[Tid, WordPos; TID: .Tid] Trips gettuples2[TID]
  filter[checkWeekday(.Str, .WordPos, "Wednesday")]{ba} (
  TripsStrTrie searchWord["Leipziger Str."]
  TripsStrTrie searchWord["Hohenzollerndamm"] concat
  TripsStrTrie searchWord["Steglitzer Damm"] concat) {nx}
  hashjoin[Tid_ba, Tid_nx] filter[.WordPos_ba < .WordPos_nx]
  projectextend[; BApos: .WordPos_ba, NXpos: .WordPos_nx, Tid: .Tid_ba]
  Trips gettuples2[Tid] filter[matchesTempCond(.Str, .BApos + 1, .NXpos + 1)]
  count
```

Runtime: 0.58 seconds.

Approach 6: Index-Supported Pattern Matching on Symbolic Trajectories

The operator **indexmatches** realizes the index-supported pattern matching approach. It is invoked with the name of the symbolic trajectory attribute (`Str`), the name of the trie containing all labels and their exact positions (`TripsStrTrie`), and the R-tree for the time intervals and their exact positions (`TripsStrRtree`). The applied pattern is the same as for Approach 4.

```
query Trips indexmatches[Str, TripsStrTrie, TripsStrRtree, '* (wednesday "Bundesallee")
  X * [(_ "Leipziger Str.") | (_ "Hohenzollerndamm") | (_ "Steglitzer Damm")] Y *
  // get_duration(X.time) > get_duration(Y.time)'] count
```

Runtime: 0.59 seconds.

REFERENCES

J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. 2008. Efficient pattern matching over event streams. In *ACM SIGMOD*. 147–160.

R. Ahas, E. Saluveer, M. Tiru, and S. Silm. 2008. Mobile positioning based tourism monitoring system: positium barometer. In *ENTER*, P. O'Connor, W. Höpken, and U. Gretzel (Eds.). Springer, 475–485.

A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley Longman Publishing Co., Boston, MA.

N. Andrienko and G. Andrienko. 2013. Visual analytics of movement: An overview of methods, tools and procedures. *Inf. Visualization* 12, 1 (2013), 3–24.

L. Chen, M. T. Özsu, and V. Oria. 2005. Robust and fast similarity search for moving object trajectories. In *SIGMOD Conference*. 491–502.

G. Cong, H. Lu, B. Chin Ooi, D. Zhang, and M. Zhang. 2012. Efficient spatial keyword search in trajectory databases. *CoRR* abs/1205.2880 (2012). http://arxiv.org/abs/1205.2880.

M. L. Damiani, H. Issa, and F. Cagnacci. 2014a. Extracting stay regions with uncertain boundaries from GPS trajectories: A case study in animal ecology. In *ACM SIGSPATIAL 2014*, Vol. 22. 253–262.

M. L. Damiani, H. Issa, R. H. Güting, and F. Valdés. 2014b. Hybrid queries over symbolic trajectories: A usage scenario. In *MDM*. 341–344.

H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh. 2008. Querying and mining of time series data: Experimental comparison of representations and distance measures. *Proc. VLDB Endow.* 1, 2 (2008), 1542–1552.

C. du Mouza and P. Rigaux. 2005. Mobility patterns. *Geoinformatica* 9, 4 (2005), 297–319.

C. Düntgen, T. Behr, and R. H. Güting. 2009. BerlinMOD: A benchmark for moving object databases. *VLDB J.* 18, 6 (2009), 1335–1368.

N. Eagle and A. Pentland. 2006. Reality mining: Sensing complex social systems. *Pers. Ubiquitous Comput.* 10, 4 (2006), 255–268.

M. Erwig, R. H. Güting, M. Schneider, and M. Vazirgiannis. 1999. Spatio-temporal data types: An approach to modeling and querying moving objects in databases. *GeoInformatica* 3, 3 (1999), 269–296.

C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. 1994. Fast subsequence matching in time-series databases. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*.

L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. 2000. A data model and data structures for moving objects databases. In *SIGMOD Conference*. 319–330.

A. U. Frank. 1996. Qualitative spatial reasoning: Cardinal directions as an example. *Int. J. Geog. Inf. Syst.* 10, 3 (1996), 269–290.

L. Gao and X. S. Wang. 2009. Time series query. In *Encyclopedia of Database Systems*. 3114–3119.

F. Giannotti and D. Pedreschi. 2008. *Mobility, Data Mining and Privacy - Geographic Knowledge Discovery*. Springer.

R. H. Güting. 2009. Moving objects databases and tracking. In *Encyclopedia of Database Systems*. 1770–1776.

R. H. Güting, F. Valdés, and M. L. Damiani. 2013. *Symbolic Trajectories*. Technical Report. FernUniversität Hagen, Informatik-Report 369.

R. H. Güting, T. Behr, and C. Düntgen. 2010. SECONDO: A platform for moving objects database research and for publishing and integrating research implementations. *IEEE Data Eng. Bull.* 33, 2 (2010), 56–63.

R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. 2000. A foundation for representing and querying moving objects. *ACM Trans. Database Syst.* 25, 1 (2000), 1–42.

R. H. Güting, V. T. de Almeida, and Z. Ding. 2006. Modeling and querying moving objects in networks. *VLDB J.* 15, 2 (2006), 165–190.

R. H. Güting and M. Schneider. 2005. *Moving Objects Databases*. Morgan Kaufmann.

M. Hadjieleftheriou, G. Kollios, P. Bakalov, and V. J. Tsotras. 2005. Complex spatio-temporal pattern queries. In *Proc. of the 31st International Conference on VLDB*. Trondheim, Norway, 877–888.

J. G. Kie, J. Matthiopoulos, J. Fieberg, R. A. Powell, F. Cagnacci, M. S. Mitchell, J.-M. Gaillard, and P. R. Moorcroft. 2010. The home-range concept: Are traditional estimators still relevant with modern telemetry technology? *Philos. Trans. R. Soc. B* 365, 1550 (2010), 2221–2231.

F. Korn, H. V. Jagadish, and C. Faloutsos. 1997. Efficiently supporting ad hoc queries in large datasets of time sequences. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*.

Z. Li, J. Han, M. Ji, L.-A. Tang, Y. Yu, B. Ding, J.-G. Lee, and R. Kays. 2011. MoveMine: Mining moving object data for discovery of animal movement patterns. *ACM Trans. Intell. Syst. Technol.* 2, 4 (July 2011), 37:1–37:32.

L. Liao, D. Fox, and H. Kautz. 2005. Location-based activity recognition using relational Markov networks. In *Proc. of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05)*. 773–778.

J. Liu, O. Wolfson, and H. Yin. 2006. Extracting semantic location from outdoor positioning systems. In *Proc. of the 7th International Conference on Mobile Data Management*. 73.

Microsoft Geolife. 2015. http://research.microsoft.com/en-us/projects/geolife. (2015).

P. Newson and J. Krumm. 2009. Hidden Markov map matching through noise and sparseness. In *ACM SIGSPATIAL GIS*. 336–343.

L.-V. Nguyen-Dinh, W. G. Aref, and M. F. Mokbel. 2010. Spatio-temporal access methods: Part 2 (2003-2010). *IEEE Data Eng. Bull.* 33, 2 (2010), 46–55.

A. T. Palma, V. Bogorny, B. Kuijpers, and L. Otávio Alvares. 2008. A clustering-based approach for discovering interesting places in trajectories. In *SAC*. 863–868.

N. Pelekis, E. Frentzos, N. Giatrakos, and Y. Theodoridis. 2008. HERMES: Aggregative LBS via a trajectory DB engine. In *SIGMOD Conference*. 1255–1258.

N. Pelekis and Y. Theodoridis. 2014. *Mobility Data Management and Exploration*. Springer.

D. Pfoser, C. S. Jensen, and Y. Theodoridis. 2000. Novel approaches in query processing for moving object trajectories. In *VLDB*. 395–406.

M. A. Quddus, W. Y. Ochieng, and R. B. Noland. 2007. Current map-matching algorithms for transport applications: State-of-the art and future research directions. *Transp. Res. Part C: Emerging Technol.* 15, 5 (2007), 312–328.

S. Reddy, M. Y. Mun, J. Burke, D. Estrin, M. H. Hansen, and M. B. Srivastava. 2010. Using mobile phones to determine transportation modes. *TOSN* 6, 2 (2010), 1–27.

C. Renso, S. Spaccapietra, and E. Zimányi. 2013. *Mobility Data – Modeling, Management, and Understanding*. Cambridge Press.

J. A. M. R. Rocha, V. C. Times, G. Oliveira, L. O. Alvares, and V. Bogorny. 2010. DB-SMoT: A direction-based spatio-temporal clustering method. In *IEEE Conf. of Intelligent Systems*. 114–119.

R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. 2004. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.* 29, 2 (2004), 282–318.

M. A. Sakr and R. H. Güting. 2011. Spatiotemporal pattern queries. *GeoInformatica* 15, 3 (2011), 497–540.

S. Spaccapietra, C. Parent, M. L. Damiani, J. A. de Macedo, F. Porto, and C. Vangenot. 2008. A conceptual view on trajectories. *Data Knowl. Eng.* 65, 1 (April 2008), 126–146.

S. Spaccapietra, C. Parent, C. Renso, G. Andrienko, N. Andrienko, V. Bogorny, M. L. Damiani, A. Gkoulalas-Divanis, J. Macedo, N. Pelekis, Y. Theodoridis, and Z. Yan. 2013. Semantic trajectories modeling and analysis. *Comput. Surveys* 45(4) (2013), 42.

L. Speicys, C. S. Jensen, and A. Kligys. 2003. Computational data modeling for network-constrained moving objects. In *ACM SIGSPATIAL GIS*. 118–125.

L. Stenneth, O. Wolfson, P. S. Yu, and B. Xu. 2011. Transportation mode detection using mobile phones and GIS information. In *ACM SIGSPATIAL GIS*. 54–63.

G. Trajcevski, O. Wolfson, K. Hinrichs, and S. Chamberlain. 2004. Managing uncertainty in moving objects databases. *ACM Trans. Database Syst.* 29, 3 (2004), 463–507.

F. Urbano, F. Cagnacci, C. Calenge, H. Dettki, A. Cameron, and M. Neteler. 2010. Wildlife tracking data management: A new vision. *Philos. Trans. R. Soc. B* 365, 1550 (2010), 2177–2185.

F. Valdés, M. L. Damiani, and R. H. Güting. 2013. Symbolic trajectories in SECONDO: Pattern matching and rewriting. In *Database Systems for Advanced Applications*. 450–453.

F. Valdés and R. H. Güting. 2014. Index-supported pattern matching on symbolic trajectories. In *Proc. of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, Dallas/Fort Worth, TX, November 4–7, 2014. 53–62.

M. R. Vieira, P. Bakalov, and V. J. Tsotras. 2010. Querying trajectories using flexible patterns. In *Proc. of the 13th Int. Conf. on Extending Database Technology (EDBT'10)*. 406–417.

M. R. Vieira, P. Bakalov, and V. J. Tsotras. 2011. FlexTrack: A system for querying flexible patterns in trajectory databases. In *12th International Symposium, SSTD*. 475–480.

W. Wu, Y. Wang, J. Bartolo Gomes, D. T. Anh, J. Decraene, S. Antonatos, M. Xue, P. Yang, G.-E. Yap, X. Li, S. Krishnaswamy, and A. Shi-Nash. 2014. Oscillation resolution for mobile phone cellular tower data to enable mobility modelling. In *MDM*. 321–328.

Z. Yan and D. Chakraborty. 2014. *Semantics in Mobile Sensing*. Morgan & Claypool.

Z. Yan, D. Chakraborty, C. Parent, S. Spaccapietra, and K. Aberer. 2011. SeMiTri: A framework for semantic annotation of heterogeneous trajectories.. In *EDBT 2011*. 259–270.

Z. Yan, D. Chakraborty, C. Parent, S. Spaccapietra, and K. Aberer. 2013. Semantic trajectories: Mobility data computation and annotation. *ACM Trans. Intell. Syst. Technol.* 4, 3 (2013), 49:1–49:38.

C. Zhang, J. Han, L. Shou, J. Lu, and T. F. La Porta. 2014. Splitter: Mining fine-grained sequential patterns in semantic trajectories. *PVLDB* 7, 9 (2014), 769–780.

K. Zheng, S. Shang, N. J. Yuan, and Y. Yang. 2013. Towards efficient search for activity trajectories. In *Proc. of the 2013 IEEE International Conference on Data Engineering (ICDE'13)*.

Y. Zheng, Y. Chen, Q. Li, X. Xie, and W.-Y. Ma. 2010a. Understanding transportation modes based on GPS data for web applications. *ACM Trans. Web* 4, 1 (January 2010), 1:1–1:36.

Y. Zheng, X. Xie, and W.-Y. Ma. 2010b. GeoLife: A collaborative social networking service among user, location and trajectory. *IEEE Data Eng. Bull.* 33, 2 (2010), 32–39.

Y. Zheng and X. Zhou. 2011. *Computing with Spatial Trajectories*. Springer.

Received August 2014; revised March 2015; accepted May 2015